Improved Models for Policy-Agent Learning of Compiler Directives in HLS

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

Abstract—**Acceleration by Field-Programmable Gate Array (FPGA) continues to be deployed into data center and edge computing hardware designs; the tools and integration for accelerating computationally-intensive tasks continue to increase in practicality. In this paper, we build on previous work in applying machine learning to automatically tune the transformation of high-level language (HLL) C code by a High Level Synthesis (HLS) system to generate an FPGA hardware design that runs at high speed. This tuning is done primarily through the selection of code transformations (optimizations) and an ordering in which to apply them. We present more detailed results from the use of reinforcement learning (RL), and improve on previous results in several ways: by developing additional strategies that perform better and more consistently, by normalizing the learning rate to the frequency of new (yet untried) action sequences, and by informing the model from aggregate statistics of optimization sub-orderings.**

Index Terms—**Compiler Optimization, FPGA, High Performance Computing, Machine Learning, High Level Synthesis**

I. Introduction

Custom circuit designs have facilitated high-performance computation since the beginning of electronic computation. Typically a custom circuit is needed to implement functions that are not implemented in a CPU instruction set or any existing coprocessor(s).

Inasmuch as hardware design is difficult for CPU programmers, and cumbersome even for experienced logic designers, high-level synthesis (HLS) was developed, incorporating languages, techniques, and tools that facilitate the conversion of a CPU program into a custom circuit design. Such HLS methods are used for both application-specific integrated circuit (ASIC) and FPGA designs. Being rapidly reprogrammable, FPGAs in particular are exploited for workload flexibility to broaden the range of applications, including use by customers and other third parties. The FPGA is therefore appearing in a greater variety of HLS-applicable use-cases including cloud computing nodes and edge devices. We consider these applications to be particularly amenable to tolerate the raised abstraction level and inevitable reduction in performance (longer run times, and greater energy usage) from using HLS methods. [1]

HLS began with a state-machine driven data flow pipeline model, analogous to a very long instruction word (VLIW) CPU design with operations scheduled as in Tomasulo [1], with the addition of modulo scheduling [2], and appears to have emerged almost seamlessly out of the behavioral style of Hardware Description Languages (HDLs); in its current form it was demonstrated at least as early as 1998 [3]. Modern day tools continue to use this model. Important differences, and greatest opportunities for exploitation of parallelism, exist in the ability to make the pipeline far wider than was possible in any VLIW CPU, along with the deepening, replication, and coupling of those pipelines.

HLS tools are capable of emulating a shared-memory parallel architecture similar to a multi-core processor (CPU or GPU), using multiple kernel processors and interposed data FIFOs. For C programs performing greatly parallelizable computations, previous research [4] found that the multi-kernel design is only an intermediate step in the set of transformations needed to achieve the shortest run time. The final transformed code has a single data flow pipeline with as much "width" as possible for the problem at hand. For example, multiplying a 10×10 matrix by another can produce a pipeline that performs 1000 multiplications simultaneously, if the device size allows for it. HLS tools that use multiple kernels or threads (whether explicitly defined in the C code from the user, generated to fulfill user-supplied pragmas, or inferred in some automatic way by the tools) produce multiple concurrent hardware modules (that correspond to threads in the CPU model, or kernels in a GPU model) with communications overhead lengthening run times.

The primary deployment target for our research is the conversion of a CPU program written in C or C++ via HLS tools into a hardware design for FPGA. The sequential imperative model of those languages restricts such tools to the rules implemented by CPU compilers, to ensure semantic correctness, maintain data dependencies, avoid hazards from address aliasing, etc. HLS tools therefore often use CPU compilers, generating code in an intermediate representation (IR) that can be optimized in all the same ways as CPU assembly language, but for an arbitrary processor design with unlimited registers and combinational logic units, with the result converted into multiple stages of sequential logic with registers between pipeline stages, and all controlled by a state machine. Higher speed is achieved via loop unrolling, vectorization, simultaneous calculation prior to conditional tests (speculative execution), and similar means. It makes sense to apply such transformations more times than would be useful in any single core of a CPU. Furthermore there is no instruction decoding or penalties for taking branches, and no need to predict branches or to maintain a branch target

[1]There are many solutions that generate hardware for specific types of work, such as signal processing, encryption, compression, machine learning kernels, etc.; but these are not universal C compilers and do not address our goals.

cache.

For these reasons this research is directed at the tasks of choosing a set of compiler optimizations and an ordering in which to perform them, allowing for any transformation to be applied multiple times, without the CPU-specific biases inherent in compiler options such as `-O3`. These are heavily researched questions commonly called the *selection* and *ordering* problems; see *e.g.* [5], [6] of which the latter discusses machine learning (ML) approaches. For both problems the number of choices is huge, precluding an exhaustive search, leading to the use of automatic learning techniques. That research addresses CPU-like targets that invariably lack hardware pipelines for many (rare but useful) computation tasks, and CPUs have architectural limitations such as a small number of user registers, the need to decode instructions, a need for branch prediction and a branch target cache, etc. Therefore the research results on the selection and ordering problems serves to inspire but not to answer the present problem. For similar reasons the heuristics embodied in compiler tools, including the orderings invoked by `-O3` and the like, apply to CPUs but not to hardware design in general.

It is therefore necessary to find new machine learning models to generate FPGA-tuned answers for the selection and ordering questions, for any computation task in C. Such is the method of AutoPhase [7], [8] and [9], which we summarize now as they are the most relevant antecedents to the present work:

- In [7], [9], reinforcement learning (RL) is used to train a multilayer perceptron (MLP) model
- The model is allowed to select a sequence of code transformations
- A C program is compiled using the selected optimizations, using HLS tools
- The HLS tools provide an estimated cycle count for the execution of the program by an FPGA
- Models are trained on each of the nine programs marked CHStone and LegUp in Fig.3
- Some sequence of transformations achieves the lowest cycle count; for each program this is compared against previous non-RL results [10]–[12]
- In the 2020 work [8] similar experiments are conducted using a random forests model [13]

Another approach (in progress) is to apply supervised learning approaches; these involve the use of tagged models of the input program and the generation of large input sets (viable randomly-generated code samples that are representative of the work suitable and desirable for an FPGA).

The work in this paper uses RL to discover solutions to the selection and ordering problems for existing single programs. This allows for discovery of what is possible or achievable in the ideal case of a supervised model that manages to achieve the best performance (shortest execution time) for any input of interest.

The contributions of this work, as compared to [7]–[9] are:

- exploration of a greater range of the parameters governing the complexity of the selection and ordering problems

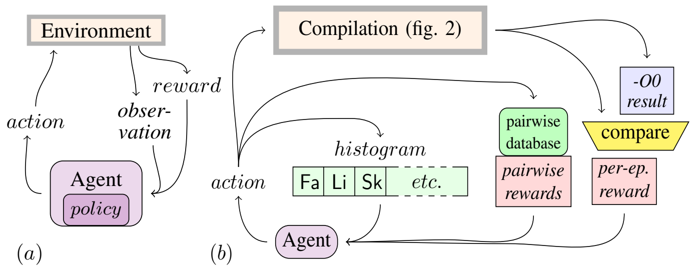

Fig. 1: a: Reinforcement Learning paradigm; b. Prior actions as observation vector, varying rewards; digraphs `Fa`, `Li`, `Sk` are abbreviated optimization names. As described in the text, normalizing to `-O0` and the use of pairwise rewards are new to this work.

- use of pure reinforcement learning to encourage unique selections and orderings for each unique program
- refinement of data analysis to better expose learning rate and speed of output design
- conducting several experiments per configuration to enable standard deviation measures
- evaluating results in relation to the null case of no optimization transformations (`-O0`) rather than the CPU-specific `-O3`
- clarification of earlier work in general and of some of the specific program results
- exploring new variations in agent models, in particular, long-short-term memory (LSTM), Deep-Q Network (DQN), and Asynchronous Advantage Actor-Critic (A3C); and
- augmenting the test suite with additional classes of relevant programs (using floating-point and a structured grid computation)

The remainder of this paper is organized as follows: the next section (II) gives background on the methods of earlier work; Section III introduces the methods of this work; Section IV describes the procedure including details relevant to our contributions; Section V presents results and discusses their significance.

II. Background and Prior Design

A. Reinforcement Learning (RL)

Reinforcement learning is an approach to ML that is well-suited to problems for which open-ended exploration is needed and there is no effective way to determine the correctness of an answer except to try it and observe the result. Refer to Fig.1(a): the RL approach defines an *Agent*, being that part of the system that contains the ML model and parameters (which are often distinguished as the *Policy*), and an *Environment* comprising everything else including in particular that which the agent takes as its input, called the *observation(s)*. The agent takes *actions*, which together with environment completely determine how well the problem is solved. Ideally, the observation(s) should include all information that might influence the agent's ability to make the best choice. This could involve large amounts of processing and analysis—perhaps more than is practical.

For example, if the agent is learning to play chess against a human, the environment includes the private thoughts of the human player, which are unknowable; but also the board position and knowledge of what moves have occurred so far—these could be thoroughly analyzed by playing out every possible game; but that is impractical. RL takes the approach of amassing experience through accumulated data from many observations, actions, and outcomes.

Each observation is given to the agent, which outputs an action; together this is called a *step*; at appropriate times (*e.g.* each step, or at the end of a set of steps called an *episode*) the environment supplies a scalar numeric *reward* to the agent representing the degree of success or failure. In the chess example the episode lasts until the last turn in the game, when the agent gets a positive or negative reward according to who won. The mathematics of Markov decision processes [14] show that this structure works, and a multilayer perceptron [15] can implement the information storage and decision calculations. See e.g. [16], [17] for background and [18] for examples.

B. Environment: Compilation and Performance Estimation

In [8], [9] the environment consists of a specific C program (constant throughout each particular experiment), a C compiler, an HDL code generator, and an evaluation algorithm to compute a run time measured in clock cycles. This is similar to that in Fig.2 though other parts of that figure apply only to the present work. The C compiler is capable of performing code transformations (optimizations) in any desired order, allowing any transformation to occur multiple times or not at all. The HDL code generator creates hardware modules expressed in Verilog, with two modules per C function, one containing the calculations on data and the other containing a finite state machine that implements the function's loops and conditional statements. The run time evaluation profiles the C program on the native CPU to discover how many times each basic block is executed, and combines this with timing information given by the HDL generator, to determine the total number of clock cycles that the hardware design needs to run the C program. This works because the HDL code generator retains the sequential imperative structure of the original C program at all levels above that of the basic block.

C. Agent, Actions, Observation, Reward

In [8], [9] the main results come from an RL agent using PPO (proximal policy optimization), and results are compared to simpler models (the `-O0` and `-O3` orderings, a "greedy" model adding whichever code transformation produces the best improvement over those already applied, a random forests model, and application of just a single code transformation).

The best results of [8] employ an observation that includes both the histogram of actions taken so far and a "feature vector". The histogram of actions is similar to that in Fig.1(b),

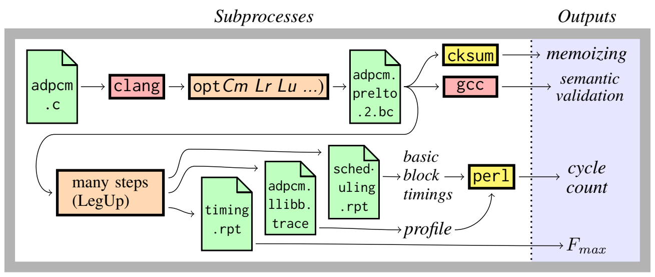

Fig. 2: Compilation and evaluation steps. As described in the text, the use of memoizing is new for this work, and enables most of the steps to be skipped; the use of F_{max} is also new to this work.

though other parts of that figure apply only to the present work. The feature vector is composed of counts of instruction types and data types, like those used in [19], [20].

III. Design and Model in This Work

A. Environment: Compilation, Performance Estimation

The details of the environment and calculation of rewards are illustrated by Fig.2.

Taking a list of code transformations (optimizations) chosen by the agent, the target C program is compiled into IR, which is then evaluated by three different methods:

- A checksum to determine whether the compilation generated a program equivalent to one seen before,
- An executable using the back end suitable for the host CPU, to validate correctness and to conduct profiling,
- HDL code using a hardware-specific back end, along with an F_{max} frequency estimate for a target FPGA, and reports from which a cycle count can be calculated.

As in [8], [9] a combination of profiling and per-module clock cycle calculation is used to calculate an overall cycle count without the need for full simulation in ModelSim or the like. In this work, the F_{max} and cycle count are combined to calculate run time as $cycles/F_{max}$.

B. Agent, Actions, Observation, Reward

Referring now to Fig.1(b): The actions from the agent, used to select code transformations, are also counted in a histogram in which each element of the histogram counts the number of times a certain action (code transformation) was selected. This histogram is a fixed-length vector, given to the agent as its observation.

When the experiment begins, the program is compiled once with the default (null) optimization option `-O0`, and execution time is calculated as $cycles/F_{max}$ for use in comparison to later attempts. Rewards given to the agent are of three types:

- When the agent has taken enough actions to comprise an episode (given the specific episode length being used in the current experiment) the program is compiled, and the reward given is $runtime_{O0} - runtime_{this}$ where $this$ represents this compilation, with each run time calculated as $cycles/F_{max}$

- If it is not the last step in the episode, in second-order training experiments (section IV-B8), a reward is given based on pairs of actions in the current (partial) actions list
- Otherwise, a reward of zero is given.

If enough steps (actions) have been taken to make up an episode, a new episode is begun by clearing the action histogram. The observation vector (action histogram) and reward(s) are given to the RL training framework, which gives the observations to the agent on each step and uses the reward(s) to calculate adjustments to the MLP model (policy) via backpropagation. Together, these influence the agent's choices in subsequent episodes.

IV. Procedure

A. Overview

Twelve benchmark C programs designed specifically for evaluating HLS were used, including all of those in the earlier works cited [7]–[9]. Evaluation of machine learning ability is conducted similarly to [21] and [8], with several methodological improvements listed below. The subject programs are compiled by Clang/LLVM and GCC compilers within the LegUp [22], [23] hardware design system version 4.0, generating both IR and Verilog HDL code. A run time is computed as described earlier. The run time is used to calculate rewards given to an RL agent running within the Ray framework RLLib library [24], [25] within OpenAI Gym [21], [26]. The agent is trained in batches of 4000 steps, with weight backpropagation after each batch. All available LLVM code transformations (of those used for optimization) are considered as choices for the selection problem, and can be used in any order with repeats. As described earlier, the agent's environment is given a vector of the code transformations chosen thus far, up to a finite length corresponding to the action limit for the particular experiment. Each vector element is an integer index into the set of possible optimizations, as in [8], except for the final `-terminate` item that is not an actual LLVM option. When the vector is full, the target C program is compiled and evaluated as described earlier.

B. Optimizations, Data Normalization and Other Adjustments

The results in this paper are obtained with all of the following changes, as compared to earlier work [9]:

1) Multiple Experiments per Configuration: To evaluate the significance of random fluctuations in the learning process, each experiment is run five times using random seeds 3 to 7 inclusive.

2) Semantic Validation: All target programs are modified as needed to check their own results against an expected correct answer and output a pass/fail message. Programs that fail give a negative reward, scaled to be as bad as a working program that is twice as slow as `-O0`. A false "pass" result is possible. Exploration of auto-generated orderings brings up several risks: compilers are not designed for arbitrary nonstandard orderings and often emit semantically incorrect output programs [27]; flawless detection of such output is a "halting problem".

Similar problems arise when trying to determine a candidate output program's run time through static analysis rather than by running it (either in simulation or on some target hardware). This is the second reason why every output program (in IR form) must be executed at least once.

3) Much Wider Range of Episode Lengths: The episode length, corresponding to the maximum number of code transformations that might be used to compile the target program, is varied over six values from 7 to 45 in geometric progression. These six variations appear in six different colors in Fig.4.

4) Learning Rate Normalization by Compilations Performed : The effectiveness of an RL agent to find optimum answers for each target program is evaluated from the trend in achieved run time as a function of a "time-like" progression represented by the horizontal axis in the Fig.4 plots. This "time-like" axis is typically in training steps, episodes, or batches, or can be actual elapsed time when running the training on particular hardware; but careful thought should be given to the effects on agent learning rate that arise from varying the rate of reinforcement feedback (from varying episode length), subject program (complex programs fail semantic validation more often), and other parameters.

To a first approximation, using episodes as the timescale is good because it is only at the end of an episode that the subject program can be compiled, getting new information (the program's run time) from which the agent can learn. As compilation takes up most of the time, this use of timescale closely reflects real-world time. Conversely, counting steps (or equivalently batches of some number of steps e.g. 4000) imposes a bias on the "time-like" axis, in which smaller episode lengths will appear to have a higher training rate because they gain more information per step (or batch) by performing more compilation evaluations.

5) CRC caching and timescale adjustment: Compiling the target program into HDL code and profiling take a much greater amount of time than the RL framework (inference and backpropagation). Therefore, a memoizing technique is used to remember program run times based on a hash of the IR generated early in the HLS-to-HDL compilation process. In Fig.2 the IR file `adpcm.prelto.2.bc` is the result of all the optimizations in the particular sequence that has been generated by the agent's choices (actions). This is the sole input for all later operations, including: compiling to run on a CPU for semantic validation (the box labeled `gcc`); and generating a hardware design (labeled `LegUp`). Before either of these relatively time-consuming tasks is done, a 32-bit hash of the IR file is computed by the ISO/IEC 8802-3:1989 algorithm. If a match is found, the previous $cycles/F_{max}$ value is used.

The probability of a false positive match is about $1/2^{32}$ for any pair of IR files, and a hash collision becomes likely if more than about 2^{16} hashes are generated. When a collision happens, it would cause the agent to receive a reward that does not properly reflect the result for that particular trial compilation, causing a small bias in the agent's training. This is unlikely to happen, as these experiments are ended after about 2^{13} episodes.

Respecting the fact that the agent is often repeating past actions, and therefore not learning as much, a cached result counts 2% as much as a new result for the purpose of deciding how far to move on the horizontal ("time-like") axis. This is reflected in the data plots and learning rate evaluations.

6) Clock Speed Normalization: For some of the target programs, the choice of code transformations causes great differences in the speed at which the HDL design can run. This results from differences in memory access patterns (for those programs that work on data arrays declared in one function and then used in a called function), and differences in depth of combinational logic. To ensure that the agent does not minimize cycle count at the expense of vastly lower frequency, the F_{max} value given by the LegUp back end is combined with the cycle count and converted to a time interval in units of 20ns.

7) Results Normalized to ***-O0****:* Rather than normalizing all run time results from all target programs to a scale of [0..1], the `-O0` run time is set to be the value 1.0, with shorter run times giving numbers above 1.0, *e.g.*, twice as fast is 2.0. These numbers are used for the other statistics and results reported here. To aid in comparison to prior work a `-O3` run time is also computed.

8) Second-Order Training: A large number of experiments were performed on each target program, and every compilation attempt and the resulting run time was recorded. From this data, statistical correlations were calculated and used to generate a database of heuristics that could be used to generate intra-episode rewards in new experiments. This is a pure RL approach, compared with the blended (unsupervised and reinforcement learning) method seen in [8] (their Section 6.2 and Fig.9). Details are in the project repository [35].

C. Specifics of Procedure

1) Choice of Subject Programs: Twelve programs were used as subjects of the compilation-training task. Nine of these are the same as in earlier works [8], [9] allowing for verification of previous results and comparison to the current work. Three more were added to provide better coverage of computationally intensive work in the spirit of [28] [2]. All 12 programs represent emerging tasks that require custom hardware or heterogeneous solutions. Several of the programs perform tasks that are now implemented in hardware, e.g. the encryption algorithms, MPEG, FFT, etc; but all are similar to new tasks that continue to emerge and cannot be performed efficiently by existing CPUs or GPUs etc., thereby encouraging new hardware designs. The three added programs, in particular: `sor-caad` solves a partial differential equation by the Euler method, easily done by GPU, but new structured grid tasks continue to emerge; `iterfl-caad` is a chaotic iterative calculation with unpredictable conditional branches, with many independent inputs that can be evaluated in parallel and so resembles Monte Carlo; and `fft-caad` performs a standard FFT in-memory; all three have modest potential for hardware parallelization.

[2]Patterson et al. [29] soon added to the Colella list, notably including tasks and motifs already well represented in the [30] subject programs used by [8] but also including some not practical for single-node FPGA solution.

name	*source*	*author(s)*	*description*
adpcm	CHStone [30]	SNU-RT [31]	audio codec, sequential error propagation
aes	CHStone	Iwata [32]	encryption
blowfish	CHStone	Young via [33]	encryption
dhry	LegUp [23]	Weicker [34]	integer CPU instruction mix
fft	CAAD [35]	Munafo	non-recursive, float, mem. r/w
gsm	CHStone	Degener via [36]	audio codec, spectral analysis
iterfl	CAAD	Munafo	float multiply+add, mem. reads
mmult	LegUp	Canis [23]	three nested loops, integer
motion	CHStone	MPEG via [36]	find inter-frame motion vectors
qsort	LegUp	Finley [37]	quicksort
sha	CHStone	Hollerbach via [33]	SHA-1 hash
sor	CAAD	Munafo	Euler integration, heat equation on grid

Fig. 3: The benchmark programs

2) Selection Complexity Survey: In the *selection* problem, there is less complexity to the problem when only a small number of code transformations (optimization passes) can be selected; conversely the availability of a large number of code transformations makes the selection problem more difficult for automated learning. Some programs will need a greater number of transformations to achieve the best run time, but an ML model may take a lot longer to discover a sufficient set of code transformations and best ordering thereof.

In order to explore the range of complexity needed for the different subject programs, all programs were used in training experiments across a range of episode length parameter values: 7, 10, 14, 20, 30, and 45. This parameter controls the maximum number of code transformations that may be performed. (Previously, in [9] the values 25, 45 and 100 were used, and in [8] only 45 was used.)

These experiments all were performed using a PPO (proximal policy optimization) agent. The survey of all benchmark programs with all episode lengths is intended to reveal the best choice of episode length for each target program, enabling variations to be explored in a more focused manner, and thereby enabling more variations.

After getting the results of the full survey with a PPO agent, experiments were performed with RL agents using the DQN and A3C methods. Further experiments were performed with an LSTM module in the policy, an option that is provided by RLLib for RL agents.

V. Results and Discussion

A. Initial Complexity Survey

Most experiments were conducted in a way that generated data such as seen in Fig.4. On the vertical axis, 1.0 represents the speed of the program when compiled with no optimizations (using `-O0`) into a hardware design. Higher numbers represent faster execution—for example, if the `-O0` design takes 100μsec, then an optimized result of 80μsec is plotted as 1.25, because $100/80 = 1.25$.

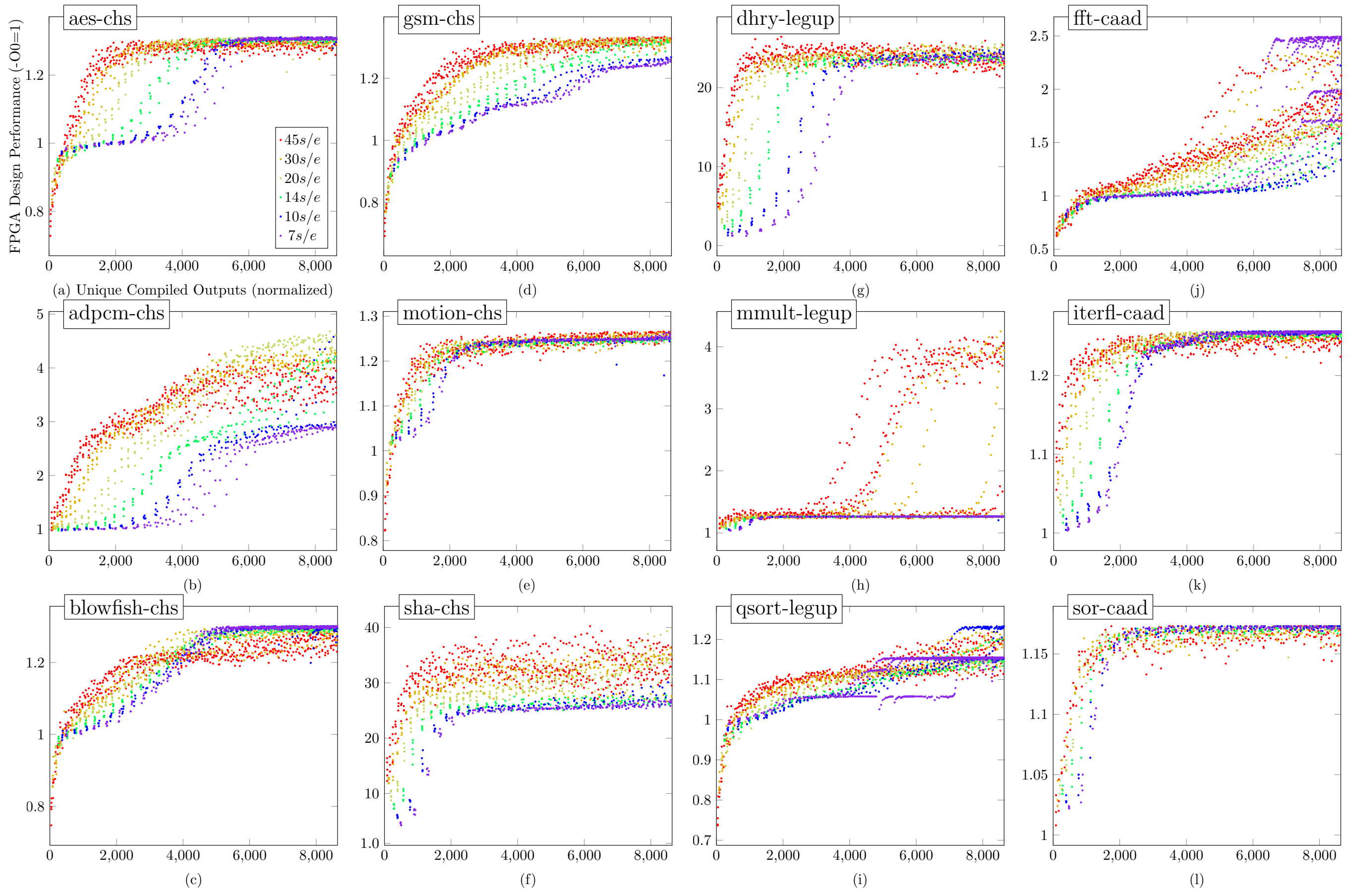

Fig. 4: Selection complexity survey: Speed $vs.$ `-O0`, by steps/episode. All have the same axis labels as (a).

TABLE I: Best Episode Length by Program. *Best* refers to the episode length achieving shortest run time; *mean vs. -O0* and *s.dev.* refer to `-O0` run time divided by shortest run time

program	*best*	*mean vs. -O0*	*s.dev.*	*mean vs. -O3*	F_{max} *range*
adpcm-chs	20	4.501	0.13	1.808	12-12
aes-chs	7	1.307	0.0021	1.007	12-12
blowfish-chs	7	1.298	0.0016	1.231	250-297
dhry-legup	20	24.80	0.48	0.765	12-400
fft-caad	7	2.213	0.31	2.284	12-292
gsm-chs	45	1.324	0.0074	1.101	136-136
iterfl-caad	7	1.255	0.00079	1.255	176-176
mmult-legup	45	2.803	1.3	2.214	297-310
motion-chs	45	1.257	0.0076	0.991	250-250
qsort-legup	10	1.176	0.036	1.047	297-297
sha-chs	20	33.87	2.8	0.853	12-400
sor-caad	10	1.172	0.0012	0.998	250-310
geom. mean(9)	-	3.201	-	1.157	-
geom. mean(12)	-	2.640	-	1.217	-

The colors show the maximum number of code transformations (episode length): red, orange, yellow, etc. for 45, 30, 20, 14, 10, or 7. For some programs it is easy to see one or two best choices of this parameter, but for most the choices depend on what type of result is desired, such as higher mean or lesser standard deviation.

More "noise" (higher standard deviation of the run time measurements) is common for the experiments using a higher episode length; this probably results from the far greater possible variation in compiled results from using more code transformations. We also observe the highest speeds, presumably because some programs require a large number of different transformations to optimize completely. In most experiments the very best run times are intermixed with many relatively poorer run times, and the agent fails to learn to exclude the latter.

Table I summarizes the data by showing, for each program, which episode length resulted in the lowest mean run time after training the RL agent. For example, in the first row `adpcm-chs` was able to run an average of $4.5\times$ faster as compared to `-O0`, when the episode length (and therefore, the number of code transformations in the compilation) was 20. The $s.dev.$ column shows a standard deviation of 0.13 for the data points from that program and episode length. For the mean and standard deviation, the final 20 samples were used, or all samples from the final 5% of training "time" when available (which primarily happened when the cached results hit rate was high, producing more data points per normalized "time"). The final two rows give geometric means for the 9 programs in [8] and for all 12 respectively.

B. F_{max} Results

Table I gives minimum and maximum F_{max} values for each program. Some give the same F_{max} for all compilation attempts; others vary as much as $33\times$. While all programs use arrays, declaration scope varies; some programs pass pointers to functions that modify array contents, presenting an aliasing hazard. Not all programs perform memory writes. Some have easily inlined leaf functions, allowing pointer parameters to become local variable-indexed array references. Some perform as many as eight reads per innermost loop iteration from different addresses. All of these variations can interact with the choice of compiler optimizations and algorithms used in the HDL back end to decide whether arrays are in static RAM and how to arbitrate multiple accesses. The raw cycle counts are partly based on critical path timing of basic blocks, which are lengthened when needed to accommodate possible memory controller delays.

As it is used as a baseline by prior work, `-O3` is shown for reference (column 5 of the table). It can be seen for which programs the CPU-specific optimization ordering of `-O3` happens to function well when generating a custom synthesized hardware design; these are the programs for which column 5 has low values. In some cases `-O3` falls far short of what is possible, and there are two programs (`fft-caad` and `iterfl-caad`) for which `-O3` fails to outperform `-O0`. This is easily understood, because as stated earlier, there are many limitations and idiosyncrasies of CPU architecture, severely constraining what a compiler might do to improve run time, and `-O3` intentionally yields to these constraints.

It is useful to consider how the results of [8] might change by choosing $cycles/F_{max}$ as a performance metric rather than just $cycles$. We found that three of the five programs achieving highest mean speeds (lowest run times) in relation to `-O3` did so with the benefit of an F_{max} greater than 200 MHz. This frequency, cited in [8], is commonly used for collaborative designs and in FPGA design research. Scaling to 200 MHz would prevent these programs from getting their best performance; and would also prevent the `-O3` result from benefiting from a higher F_{max}, which may have comparatively inflated the results presented in that work. [3]

C. Variations on Agent Learning Model

The DQN and A3C experiments were performed with programs that had presented problems as seen in Fig.4. Each of the programs `adpcm-chs`, `aes-chs`, `blowfish-chs`, `dhry-legup`, `gsm-chs`, `mmult-legup`, `motion-chs`, `qsort-legup`, and `sha-chs` was used with one or both agent types, repeating the choice(s) of episode lengths that seemed most promising from the full survey.

[3]Using all our data, for all observed cycle counts irrespective of F_{max}, the means vs. `-O3` (last two rows of Table I col. 5) rise to 1.317 and 1.405.

Unfortunately, all achieved poorer results: lower performance estimates (longer run times) after an equal amount of training (under the normalization described in Section IV-B4) or achieving very nearly the same run times but with a considerably longer training time required, and in all cases having a greater standard deviation in the achieved run times after training.

The experiments employing long-short-term memory (LSTM) also failed to produce better results.

D. Second-Order Training

From detailed logs of all experiments of Section V-A, all pairs of optimizations and associated reward values are normalized and averaged to create a database of *pairwise rewards* (see Fig.1). Care is taken to correct for episode lengths, and not to count (A, B, A, B) as three instances of $A..B$. To prevent the second-order training from learning to replicate bad orderings generated in the earliest batches of the original experiments, this process omits data from compilations that failed to produce a program better than `-O0`. A PPO agent is trained with low-magnitude rewards on each step in an episode, determined by the pairs present in the agent's actions thus far in that episode, and scaled in proportion to the correlated reward weights in the database. The intent is to encourage agents to develop a mild preference for sequences containing known good subsequences, and thereby possibly discover the better full sequences more quickly. These experiments were performed with `adpcm-chs` and `qsort-legup`; varying the overall magnitude of these mini-rewards across a $60\times$ range; results were no better (but also no worse) than in Section V-A.

VI. Conclusions; Future Work

We conclude by underscoring and sharpening the finding of [9] that these methods perform well for some programs, but not for all; and recommending that machine learning be used together with domain-specific knowledge and expert design work, to take maximum advantage of configurable hardware platforms for those computationally intensive tasks that can benefit. To that end, it will be best to pursue further research in the analysis of HLL programs to identify characteristics that can helpfully assist an artificial model in identifying when the best performance is possible and how to achieve it.

Furthermore, as machine learning provides a vast and continually expanding range of methods with many variations and adjustable parameters, there is always more work of this type to be pursued. There are other open-source HDL generation tools, and some of these enable arbitrary optimization orderings; the methods presented here can be used to evaluate their potential.

Acknowledgments

This work was supported, in part, by the NSF through awards CCF-1919130, CNS-1925504, and CCF-2151021; by a grant from Red Hat; and by AMD and Intel both through donated FPGAs, tools, and IP.

References

[1] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[2] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," *ACM SIGMICRO Newsletter*, vol. 12, no. 4, pp. 183–198, 1981.

[3] M. B. Gokhale and J. M. Stone, "NAPA C: Compiling for a hybrid RISC/FPGA architecture," in *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No. 98TB100251)*. IEEE, 1998, pp. 126–135.

[4] A. Sanaullah, R. Patel, and M. Herbordt, "An empirically guided optimization framework for FPGA OpenCL," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 46–53.

[5] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[6] A. H. Ashouri, G. Palermo, J. Cavazos, and C. Silvano, *Automatic tuning of compilers using machine learning*. Springer, 2018.

[7] A. Haj-Ali and Q. Huang, "CS294-112 & CS262A project report AUTOPHASING: Learning to optimize compiler passes with deep reinforcement learning," https://people.eecs.berkeley.edu/ kubitron/courses/cs262a-F18/projects/reports/project2_report_ver3.pdf, 2018, student project report, University of California Berkeley.

[8] Q. Huang, A. Haj-Ali, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek, "AutoPhase: Juggling HLS phase orderings in random forests with deep reinforcement learning," *arXiv preprint arXiv:2003.00671*, 2020.

[9] 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 *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 2022, pp. 13–22.

[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 *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2013, pp. 89–96.

[11] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[12] F.-A. Fortin, F.-M. De Rainville, M.-A. G. Gardner, M. Parizeau, and C. Gagné, "Deap: Evolutionary algorithms made easy," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 2171–2175, 2012.

[13] A. Cutler, D. R. Cutler, and J. R. Stevens, "Random forests," in *Ensemble machine learning*. Springer, 2012, pp. 157–175.

[14] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957.

[15] L. Noriega, "Multilayer perceptron tutorial," *School of Computing. Staffordshire University*, vol. 4, no. 5, p. 444, 2005.

[16] D. P. Bertsekas *et al.*, "Dynamic programming and optimal control 3rd edition, volume ii," http://web.mit.edu/dimitrib/www/dpchapter.pdf, 2011, belmont, MA: Athena Scientific.

[17] R. A. Howard, "Comments on the origin and application of markov decision processes," *Operations Research*, vol. 50, no. 1, pp. 100–102, 2002.

[18] T. R. T. (ray.io), "Rllib examples," https://docs.ray.io/en/latest/rllib/rllib-examples.html.

[19] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 455–466.

[20] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–28, 2017.

[21] C. Cummins, H. Leather, B. Steiner, H. He, and S. Chintala, "CompilerGym: A reinforcement learning toolkit for compilers," https://github.com/facebookresearch/CompilerGym/, 2020.

[22] 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, pp. 1–27, 2013.

[23] A. Canis, J. Choi, B. Fort, B. Syrowik, R. L. Lian, Y. T. Chen, H. Hsiao, J. Goeders, S. Brown, and J. Anderson, *Legup high-level synthesis*. Springer, 2016, ch. 10, pp. 175–190.

[24] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica, "Rllib: Abstractions for distributed reinforcement learning," in *International conference on machine learning*. PMLR, 2018, pp. 3053–3062.

[25] A. Inc., "RLlib: Industry-grade reinforcement learning," https://docs.ray.io/en/latest/rllib/index.html, 2021.

[26] OpenAI, "OpenAI/Gym: A toolkit for developing and comparing reinforcement learning algorithms," https://github.com/openai/gym, 2020.

[27] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[28] P. Colella, "Defining software requirements for scientific computing," slides from a presentation to the DARPA HPCS, 2004.

[29] D. Patterson, K. Asanovic, and K. Keutzer, "Computer architecture is back-the berkeley view of the parallel computing research landscape," in *Stanford EE Computer Systems Colloquium, January*, vol. 31, 2007, p. 167.

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

[31] S. S. Lim, "SNU real-time benchmarks suite for worst case timing analysis," http://archi.snu.ac.kr/realtime/benchmark, 1996.

[32] A. Iwata and M. Sato, "AES advanced encryption standard (source code)," http://www-ailab.elcom.nitech.ac.jp, 2006.

[33] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.

[34] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.

[35] Software developed for this work is available at https://github.com/mrob27/impalcdhls.

[36] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 1997, pp. 330–335.

[37] D. R. Finley, "quicksort, public-domain c implementation," http://alienryderflex.com/quicksort/.