Optimization of Halide Image Processing Schedules with Reinforcement Learning

Marcelo Pecenin[1], **André Murbach Maidl**[2], **Daniel Weingaertner**[1]

[1] Informatics Department – Universidade Federal do Paraná (UFPR)
Curitiba, PR – Brazil

[2]Elastic, Curitiba, PR – Brazil

`{mpecenin,danielw}@inf.ufpr.br, andremm@elastic.co`

Abstract. *Writing efficient image processing code is a very demanding task and much programming effort is put into porting existing code to new generations of hardware. Besides, the definition of what is an efficient code varies according to the desired optimization target, such as runtime, energy consumption or memory usage. We present a semi-automatic schedule generation system for the Halide DSL that uses a Reinforcement Learning agent to choose a set of scheduling options that optimizes the runtime of the resulting application. We compare our results to the state of the art implementations of three Halide pipelines and show that our agent is able to surpass hand-tuned code and Halide's auto-scheduler on most scenarios for CPU and GPU architectures.*

1. Introduction

Image processing libraries devote great amount of their development effort in optimizing algorithms and code in order to achieve good performance. This optimization, for each algorithm (and sometimes even for each image size), has to be custom tailored to specific hardware architectures, if the code is to run close to the theoretical optimum. Besides, "good performance" can be defined in different ways (compute time, energy consumption, memory usage), and thus require specific optimization strategies.

Optimized code development is a very challenging task, with few available skilled programmers. Once the code is optimized, it is usually very intricate and specific. Small modifications in an image processing pipeline or data structure can render all optimization inefficient or even useless, demanding a re-implementation.

Given the great profusion of imaging hardware architectures, specially considering mobile and embedded systems, it is almost impossible to develop and maintain optimized versions of image processing tools that can be kept updated considering the above mentioned variables. Therefore, Halide [Ragan-Kelley et al. 2012] was created as a Domain Specific Language (DSL) aiming to decouple algorithms from schedules, allowing for an easier optimization of image processing pipelines.

We propose an approach based on the Proximal Policy Optimization [Schulman et al. 2017] reinforcement learning agent to semi-automatically generate/optimize schedules for Halide algorithms. We developed an interface between the Halide compiler and the agent so that it can "learn" to choose the scheduling directives guided by a feedback based on the runtime of the generated code. Experiments with tree algorithms were performed on CPU and GPU architectures, and were compared to versions generated by Halide's auto scheduler and hand optimized.

2. Background

2.1. Halide Language

Halide DSL is designed to facilitate the portability of high performance code for image processing in modern heterogeneous architectures, generating code capable of exploiting memory locality, vectorization (SIMD instructions) and parallelism in multicore CPUs and GPGPUs [Ragan-Kelley and Adams 2012]. Halide is implemented as a C++ embedded DSL and distributed as a library. It can be compiled for a broad set of platforms such as: X86, ARM, CUDA, OpenCL and OpenGL on OS X, Linux, Android and Windows.

The innovation of Halide is the decoupling between the definitions of the algorithm (logic) and the organization of how it should be computed (execution schedule), i.e., in Halide the heuristic of the image processing algorithm is separated from the nesting of processing loops, parallelism statements, vectorization, etc. [Ragan-Kelley et al. 2012]. Changing the schedule does not affect the definition nor the results of the algorithms logic, making it easier for programmers to experiment with different schedules in order to achieve a better performance on specific hardware architectures.

Image processing algorithms are defined in Halide as a multistage processing pipeline that performs operations on data consumed from previous stages and produces the result for subsequent stages, until the end of the algorithm. A pipeline can be represented as a graph connecting different stages, coded using a functional syntax, in order to facilitate the understanding and readability of the code.

The execution schedule consists of a series of *scheduling directives* that will be applied to the pipeline stages. These directives are implemented as methods of C++ classes defined by the Halide DSL and drive the Halide compiler's code generation for the corresponding pipeline. Listing 1 presents an image smoothing algorithm (3×3 blur) written in Halide, including the definition of a pipeline (Lines 5 and 6) and a possible execution schedule with directives (Lines 8 and 9). Line 11 compiles the code and generates the corresponding executable through the `realize` operation.

```
1  Image<float> in = load<float>("img.png");
2  Func blur_x, blur_y; Var x, y, xi, yi;
3
4  // Algorithm / Processing pipeline
5  blur_x(x, y)=(in(x,y) + in(x+1,y) + in(x+2,y)) / 3;
6  blur_y(x, y)=(blur_x(x,y) + blur_x(x,y+1) + blur_x(x,y+2))/3;
7  // Execution schedule
8  blur_y.tile(x, y, xi, yi, 8, 4).parallel(y).vectorize(xi, 8);
9  blur_x.compute_at(blur_y,x).vectorize(x,8);
10
11 Image<float> out = blur_y.realize(in.width()-2, in.height()-2);
```

Listing 1. Halide program for blurring an image using a $3{\times}3$ mean kernel.

2.2. Reinforcement Learning

A Reinforcement Learning agent interacts with an environment through actions, and receives a reward as a return for each action taken. From its interactions with the environment, the agent learns to choose future actions that maximize the received reward [Ottoni et al. 2015]. The agent represents a control and optimization heuristic while the environment represents a problem modeled by a Markov Decision Process [Sutton and Barto 1998]. For each interaction, the agent observes the current state of the environment and then chooses an action. This action leads the environment to a new state, returning reward value as a measure of the quality of this state change [Júnior 2012]. This process is illustrated in Figure 1.

The agent maps the states of the environment into actions using the agent policy function, which can be guided by different heuristics. The task of the agent is to find a sequence of actions that produces an optimized policy, i.e., that maximizes the total sum of the received rewards [Silva 2016]. Unlike supervised learning methods in which there are "input/output" pairs to be used in training, the agent needs to gain experience (knowledge) from possible states, actions and rewards without *a priori* knowledge [Júnior 2012].

State-of-the-art reinforcement learning method Proximal Policy Optimization (PPO) [Schulman et al. 2017] supports the application of reinforcement learning in high-dimensional actions scenarios, with actions in the continuous space or with a large amount of discrete actions. The PPO is based on an Actor-Critic architecture, as shown in Figure 1. For each reinforcement learning interaction, the PPO agent uses the value estimated by the critic network to determine a Temporal-Difference (TD) error [Silver et al. 2014]. This information is later used to update the weights of both neural networks, through an Adam optimizer (gradient descent).

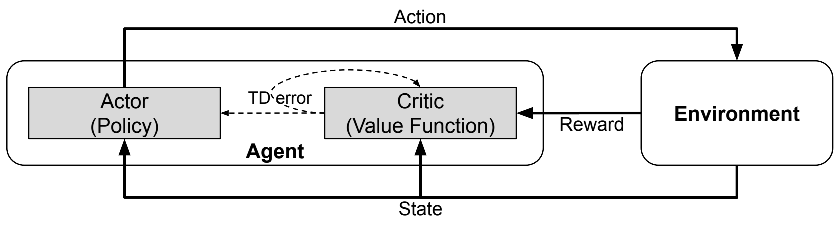


Figure 1. Actor-Critic reinforcement learning architecture. The agent uses two neural networks: the critic network that defines a value function for the environment state, and the actor network that represents the policy to choose actions based on the environment's state. Adapted from [Huang 2018].

3. Related Work

Since the conception of the Halide language, with the decoupling between the definition of the pipeline and the scheduling, efforts have been made to try to automate the generation of optimized schedules. Halide authors show that analyzing and generating individual schedules for each pipeline stage is simple, but complexity raises exponentially when composing these stages. Thus, finding an optimized schedule through exhaustive search for all possible scheduling alternatives is impracticable due to the huge search space. For example, the pipeline of an algorithm with approximately 100 stages has an estimated lower limit of 10^{720} possible schedules [Ragan-Kelley 2014, Ragan-Kelley et al. 2013].

In order to support the creation of schedules, Halide authors proposed an autotuner that uses a genetic algorithm to find an efficient pipeline. However, this approach needs to define specific rules for each schedule being optimized, presents difficulties in overcoming local minimum, and has a very slow convergence time [Mullapudi et al. 2015]. There is also a Halide schedule optimizer based on OpenTuner that has more generalized rules [Ansel et al. 2014]. OpenTuner was able to find efficient scheduling for simple pipelines (8 stages), but failed to converge in more complex pipelines (32, 44 and 49 stages), with generated schedules five to ten times slower when compared to manually optimized.

Another approach [Mullapudi et al. 2016] extended the function domain boundary analysis mechanism of Halide to split the pipeline stages into groups and find an efficient tile for each group by estimating arithmetic cost for each stage. The cost is used to define a tile size that minimizes the number of data loads and the processing loops are ordered to maximize data locality and the parallelism of the outermost loops. This approach is able to generate optimized schedules in short time, but not currently for GPU architecture, and by inspecting the generated code the authors noticed that it could be significantly improved.

4. Halide Schedule Optimization with Reinforcement Learning

Halide schedule optimization refers to the task of finding an execution schedule for a given Halide image processing pipeline and hardware platform that minimizes a cost function. The proposed solution receives as input the Halide image processing pipeline and a possible set of scheduling options for this pipeline. A reinforcement learning agent then takes care of choosing which scheduling options in which order are best for optimizing the pipeline, aiming to minimize the resulting program's runtime.

Learning occurs in an iterative manner, choosing a scheduling option and testing the resulting runtime by executing the program. Gradually, with the accumulated experience, increasingly better scheduling options are selected, and the estimates produced by reinforcement learning help to choose scheduling options that are more likely to perform well. This automated exploration of different scheduling options is possible because Halide ensures that changes in execution scheduling do not change the outcome of the implemented pipeline. An overview of the proposed solution is presented in Figure 2.

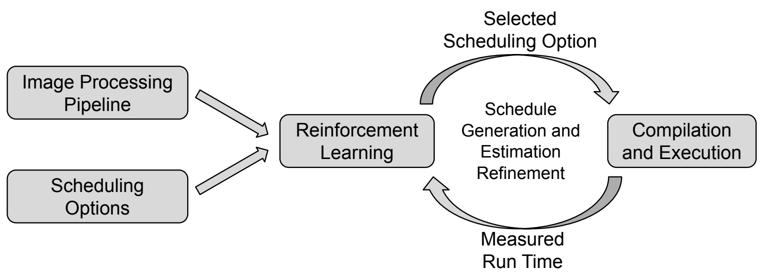


Image Processing Pipeline
Scheduling Options
Reinforcement Learning
Selected Scheduling Option
Schedule Generation and Estimation Refinement
Compilation and Execution
Measured Run Time

Figure 2. The image processing pipeline and the scheduling options are the input for the reinforcement learning agent to generate execution schedules by choosing and executing one scheduling option a time, refining estimates for upcoming choices based on information gathered from previous choices and reward based on runtime.

4.1. Scheduling Options Search Space

Each scheduling option available to the reinforcement learning agent corresponds to a single Halide directive, including the stage of the pipeline the directive should be applied to and all the parameters of that directive. Given

S: the set of pipeline stages of a Halide program (e.g. S ={`blur_x`, `blur_y`});

$\Phi(s)$: a function that given a stage $s \in S$ returns a set D_s of possible scheduling directives for this stage (e.g. s={`blur_y`} $\rightarrow$ D_s={`tile`, `vectorize`,..});

$\Pi(d)$: a function that given a scheduling directive $d \in D_s$ returns its list of parameters P_d (e.g. d={`tile`} $\rightarrow$ P_d={dim_x, dim_y} or d={`vectorize`} $\rightarrow$ P_d={`Var`, SIMD size}); and

$\Theta(p, d, s)$: a function that given a parameter $p \in P_d$, returns a possible value to this parameter when used in directive d for stage s.

Thus, a scheduling option can be expressed as a tuple $< s, d, A >$, where $A = \{a \mid \forall p \in \Pi(d)\ \exists a \in \Theta(p, d, s)\}$. The search space for the scheduling options is given by combinations of S, D_s and A, and can be huge [Ragan-Kelley 2014]. In order to reduce it we propose that the programmer should elicit the elements that make up each of the sets S, D_s, and A, as illustrated in Figure 3. To achieve that, the programmer can rely on her/his professional experience, knowledge of the pipeline in question, as well as of the desired target architecture to elicit only more plausible scheduling options, discarding any options considered inefficient or invalid.

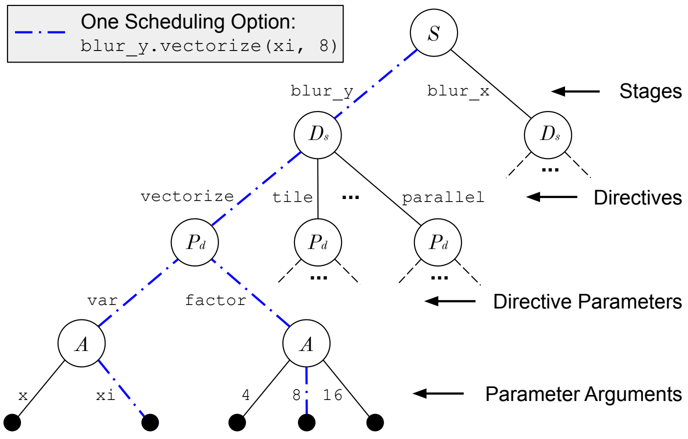


Figure 3. Illustration of the space elicitation of scheduling options for the Blur pipeline. The blue line indicates one possible scheduling option.

Listing 2 exemplifies scheduling options deemed important for the schedule generation of the *Blur* pipeline presented in Listing 1, with their respective set of possible parameters and arguments. These options and a Halide algorithm must be provided by the programmer as input to the PPO agent, as depicted in Figure 2.

4.2. Reinforcement Learning Environment Design

The two main components of reinforcement learning are Agent and Environment, as illustrated in Figure 1. In this work, an implementation of the PPO provided by the OpenAI Baselines library [Dhariwal et al. 2017], in Python, was used as agent, while the environment was developed with the ability to interact with the Halide language and its image processing pipelines. The communication between Python and Halide is done using gRPC/Protobuf synchronous messages.

The state of the reinforcement learning environment is represented by the set of directives already used in the scheduling definition. In this representation, each used

```
void options(HalideScheduleMapper &sm){
   vector<Expr> split_factor = {8, 16, 32, 64, 128, 256, 512};
   vector<Expr> vecto_factor = {4, 8, 16};
   vector<Expr> unroll_factor = {2, 3, 4};
   sm.map(blur_y)
      .bound({y}, {0}, {input.height()})
      .bound({x}, {0}, {input.width()})
      .compute_root()
      .tile({x}, {y}, {xi}, {yi}, split_factor, split_factor)
      .split({y}, {yi}, split_factor)
      .parallel({y})
      .unroll({xi, x}, unroll_factor)
      .vectorize({xi, x}, vecto_factor);
   sm.map(blur_x)
      .store_at({blur_y}, {y})
      .compute_at({blur_y}, {x, yi})
      .unroll({x}, unroll_factor)
      .vectorize({x}, vecto_factor);
}
```

Listing 2. Scheduling options definition code example for Blur pipeline.

element $s \in S$, $d \in D_s$ and $a \in A$ receives a unique numerical identifier so that a Halide execution schedule with several directives is transformed into a vector of numerical data that represents the state of the environment. Every time the same stage s, directive d or parameter argument a is referenced two or more times within the execution schedule, the same identifier code is used, but preserving in the vector the order and position where each element appears in the schedule.

The actions of reinforcement learning are represented by the scheduling directives available for a given input pipeline, elicited by the programmer as described in Section 4.1, with each scheduling option $< s, d, A >$ corresponding to a single action. In this way, performing an action in the environment means applying a scheduling directive to the input pipeline. In addition, it was also added the possibility to perform a special action, called no-operation, which is a command to finalize the current schedule.

The reward is represented by a scalar value returned by the reinforcement learning environment after performing an action, i.e., apply a scheduling option $< s, d, A >$ to the current execution schedule. Positive values mean that the action generated a reduction in the runtime and negative values indicate that it caused some error in the compilation or execution of the pipeline. As shown in Equation 1, the reward r gives more importance to big reductions in time, however, when the execution time increases, the reward is set to zero so it does not penalize actions that may be important to obtain future gains.

$$r = \begin{cases} (rt_{prev} - rt_{curr}) \times r_s & \textbf{if } rt_{curr} < rt_{prev} \\ 0 & \textbf{if } rt_{curr} >= rt_{prev} \\ -1 & \textbf{if } error \end{cases} \tag{1}$$

where $r_s = 100/rt_{init}$ is the normalized scaling factor using the initial execution time rt_{init} of the input pipeline. In this way, the reward will have a similar amplitude for different input pipelines, proportional to the obtained gain, even if they have quite different size and execution times.

5. Experimental Results

The proposed PPO reinforcement learning agent[1] was tested on three image processing pipelines, using different image sizes and CPU and GPU computing architectures.

5.1. Experimental Setup

Experiments were executed on a cluster node from the C3SL Research Group with the configuration listed below. Each process was pinned to a single socket, being able to use only the cores and local memory of that socket. The number of threads that could be used by Halide was limited to the number of cores per socket:

- CPU Intel Xeon E5-4627v2 3.30GHz, 4 Sockets, 8 Cores/Socket, 128GB RAM;
- GPU NVIDIA Tesla K40m, 745MHz, 3.5 CUDA CC, 12GB GDDR;
- Linux x86_64, GCC 5.4.0, Halide 2018/02/15, OpenAI Baselines 0.1.5.

Three Halide image processing pipelines proposed by [Mullapudi et al. 2016] and available in the Halide language repository were used for schedule generation:

Blur: image smoothing filter with a 2-stage pipeline, which receives grayscale images as input and produces output in the same format;

Harris: corner detection filter with a 13-stage pipeline, receiving color images as input and producing grayscale output;

Interpolation: pixel interpolation using pyramids, working with different resolution scales and data dependencies, with a 52-stage pipeline and 10 pyramid levels.

Training of the PPO agent for each pipeline was performed for only one image size, using `image3` with 3848×2568 pixels. Two additional images: `image1` with 962×642 pixels and `image2` with 1924×1284 pixels were also used to evaluate the performance of the generated schedules and for comparison with other approaches.

For the composition of the scheduling options set $< s, d, A >$, provided as input to the reinforcement learning, the following directives and parameters were considered:

`tile, split:` $\{8, 16, 32, 64, 128, 256, 512\}$ sizes for each dimension of processing windows (*e.g.* number of threads and blocks for `gpu_tile`);

`vectorize:` $\{4, 8, 16\}$ sizes for vectorization instructions;

`unroll:` $\{2, 3, 4\}$ loop unroll factor;

`compute_root, compute_at, store_at:` alternative positions for processing intermediate stages and storing partial results, configured according to the dependency of data between the stages of each pipeline.

`parallel:` use multiple threads;

`bound:` setting known limits of dimensions in the domain variables.

Configuration parameters for the PPO agent were based on [Schulman et al. 2017], empirically optimized, and are presented in Table 1.

The proposed PPO agent was used to generate schedules for the three previously mentioned Halide pipelines. For each pipeline and computing architecture, four trials with different random seeds were executed and the best performing schedule was chosen to be compared in terms of performance (runtime), to execution schedules produced using two other generation methods described in [Mullapudi et al. 2016]:

[1]Available at: `https://github.com/mpecenin/wscad-2019`

Table 1. Parameters for the PPO Reinforcement Learning Agent.

Hyperparameter	Value	Hyperparameter	Value
Trajectory segment length	256	Entropy coefficient	0.03
Optimization epochs	4	Annealing type	linear
Minibatch size	64	Iteration limit (episodes)	10000
Adam optimizer stepsize	2.5×10^{-3}	Number of parallel actors	1
Future reward discount (γ)	0.99	Neural network type	MLP/tanh
GAE estimator parameter (λ)	0.95	Hidden layer size	64
Policy ratio clipping (ϵ)	0.2	Number of hidden layers	2

Hand-tuned: schedules were manually optimized by experienced Halide programmers and are available in the Halide language repository.

Auto-schedule: is the automatic schedule generator provided by the Halide language compiler, initially proposed by [Mullapudi et al. 2016]. In the current version of Halide, this mechanism only produces schedules for CPU architecture. Its generation parameters were defined according to the hardware specification.

PPO-schedule: uses the proposed PPO agent. The reinforcement learning was applied separately for each pipeline and hardware architecture. The best schedule found in each case was used in the comparison with the other methods.

A validation test was performed comparing pixels of the output images from all pipelines, and no difference was encountered. The execution time rt used to compare the generation methods was computed according to Equation (2)

$$rt = \min_{1..10}(\sum_{1}^{10} \mathsf{runtime}(pipeline) \times 0.1) \qquad (2)$$

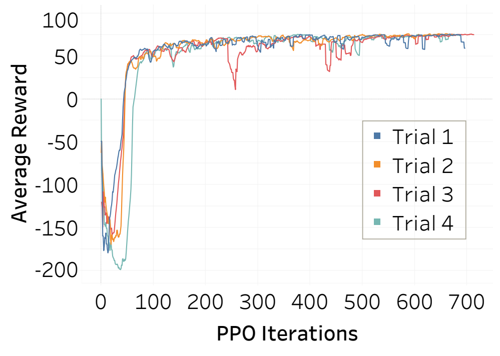
where $pipeline \in \{Blur, Harris, Interpolation\}$ and $\mathsf{runtime}()$ is the execution time, presented in milliseconds, without considering the time spent compiling the Halide program.

5.2. Results and Discussion

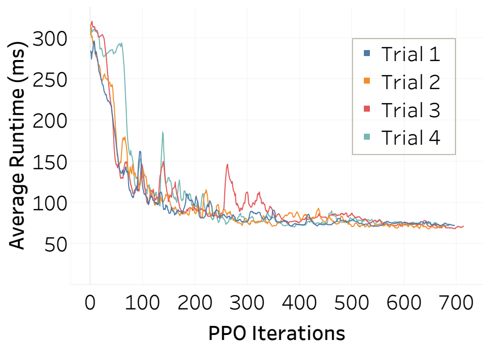
Evolution of the schedules created by the PPO Agent for the *Interpolation* pipeline is depicted in Figure 4. It shows four independent execution trials where, at the beginning of the training, the reward is negative (Figure 4a), indicating that the first schedules do not produce a valid program (1). As the iterations proceed however, reward gets positive and the average runtime of the produced program (Figure 4b) decreases. Similar behaviour is observed for the *Blur* and *Harris* pipelines.

A complete trial of schedule generation with the PPO agent averaged 20h for the *Blur*, 25h for the *Harris* and 130h for the *Interpolation* pipeline. Most of this time, however, was not spent on the execution of the generated program itself, nor the time spent in the updates of the agent's neural networks, but the most time-consuming task is the compilation of the Halide code after each action executed in the environment. Compilation time increases with the size of the pipeline, and it is plausible to consider that the larger the pipeline size, the longer it will take to run a trial of the reinforcement learning agent.

Listings 3 and 4 show the auto-tuned schedules generated for the Blur pipeline allowing a comparison of schedules evolved by the PPO agent, both for CPU and GPU.



(a) Reward



(b) Runtime

Figure 4. Evolution of the Reward (a) and Runtime (b) of schedules generated by the PPO agent for the *Interpolation* pipeline during four independent executions (trials). Negative rewards indicate invalid schedules.

```
// Hand-tuned:
  blur_y.split(y, y, yi, 8).parallel(y).vectorize(x, 8);
  blur_x.store_at(blur_y, y).compute_at(blur_y, yi).vectorize(x, 8);
// PPO:
  blur_y.unroll(x, 4).parallel(y).bound(x, 0, input.width());
```

Listing 3. Comparison of Blur schedules for CPU (Hand tuned x PPO)

```
// Hand-tuned:
  blur_y.gpu_tile(x, y, xi, yi, 16, 16);
  blur_x.compute_at(blur_y, x).gpu_threads(x, y);
// PPO:
  blur_y.gpu_tile(x, y, xi, yi, 128, 8).unroll(yi, 4).unroll(xi, 2);
```

Listing 4. Comparison of Blur schedules for GPU (Hand tuned x PPO)

Comparison between the runtime of Halide programs generated by experienced programmers (*Hand-tuned*), by Halide's *Auto-scheduler*, and our proposed *PPO-schedule* for CPU and GPU architectures is shown in Figure 5 in the form of relative performance bars. Each row indicates one pipeline and each column one image size. The relative performance is based on the ratio between the execution times of the compared schedules with the best program (smallest runtime) valued at 1.0 and the others having their runtime scaled accordingly, allowing for an easy comparison.

The general outcome shows that the *PPO-schedule* is capable of generating competing schedules in all scenarios. It has the best results on GPU for all but one image size on the *Interpolation* pipeline, with up to $\approx 50\%$ faster execution time than hand tuned code (Halide's auto-scheduler is not currently capable of generating GPU schedules). On CPU *PPO-schedule* has the best results with the large images, and performs very close to the best on all but one image size on the *Interpolation* pipeline.

Table 2 lists the absolute values of the execution time in each evaluated scenario, as well as the slowdown[2] of each case, related to the best result within the same hardware architecture, pipeline and input image. In the table, the scenarios that obtained the best

[2]Slowdown: how many times slower was one result compared to another.

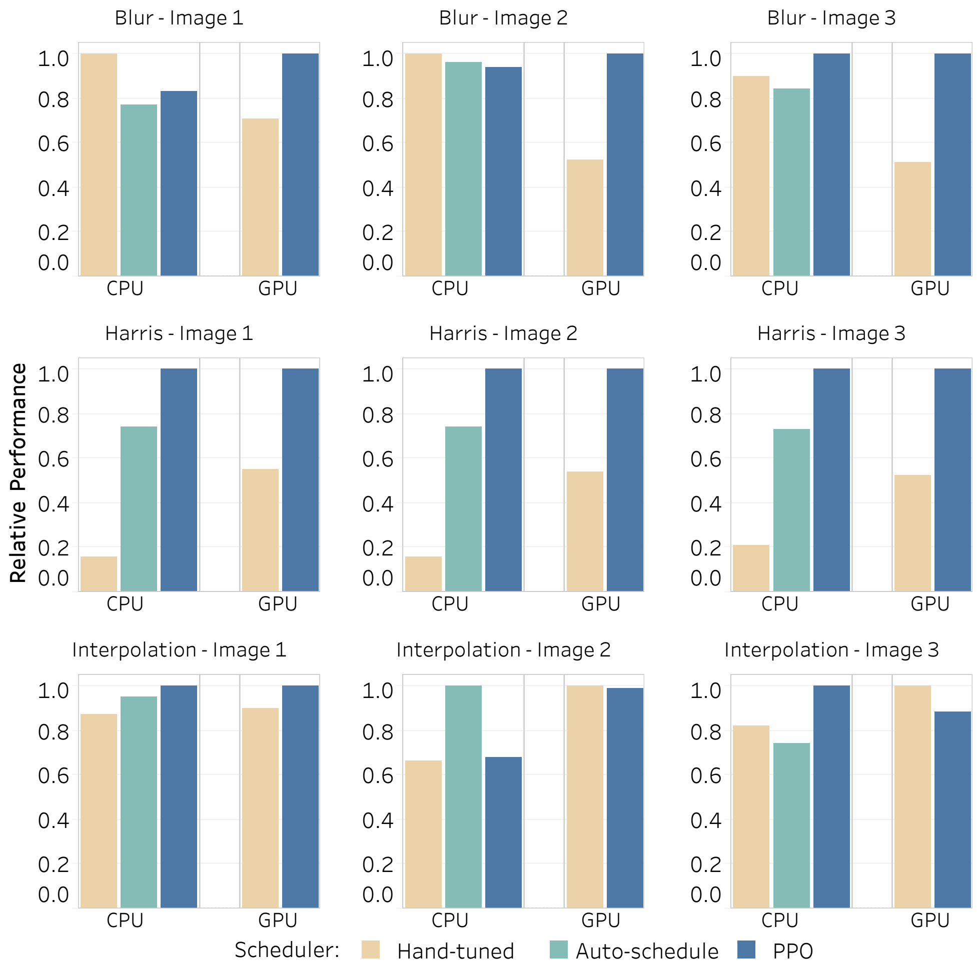


Figure 5. Performance relative to the best result by architecture and scheduling method. The bigger the better. The relative performance is based on the ratio between the execution times of the compared schedules. The best has value 1.0 and the others have lower values.

result, *i.e.*, the shortest execution time, are highlighted in bold, while those with inferior results have their slowdown relative to the best presented.

6. Conclusion

In this work we presented an approach for generation and optimization of Halide schedules using a reinforcement learning technique. The approach utilizes an implementation of a PPO agent that interacts with an environment built within this work. One of the important aspects within the definition of the environment is the calculation of the reward, which represents the most important information that guides the learning of the agent. Another relevant aspect in defining the environment is the composition of the space of actions that will be explored by the agent. This space is defined from the set of scheduling options and entered as input during the initialization of the environment.

In the current implementation the scheduling options are elicited by the Halide programmer for each pipeline to be optimized. An advantage at this point is that it allows the programmer to use her/his professional experience and knowledge about the pipeline to include only plausible options, thus reducing the search space to be explored by the

Table 2. Absolute Execution Time (ms) and Relative Slowdown ($\times$) by Architecture and Scheduling Method.

		CPU						GPU			
		Hand-tuned		Auto-sched.		PPO		Hand-tuned		PPO	
		ms	$\times$	ms	$\times$	ms	$\times$	ms	$\times$	ms	$\times$
Blur	Img1	**0.10**	-	0.13	1.3	0.12	1.2	0.07	1.4	**0.05**	-
	Img2	**0.49**	-	0.51	1.0	0.52	1.1	0.21	1.9	**0.11**	-
	Img3	2.94	1.1	3.15	1.2	**2.66**	-	0.76	1.9	**0.39**	-
Harris	Img1	3.21	6.4	0.68	1.4	**0.50**	-	0.22	1.8	**0.12**	-
	Img2	13.07	6.2	2.82	1.3	**2.10**	-	0.72	1.8	**0.39**	-
	Img3	36.89	4.7	10.71	1.4	**7.78**	-	2.79	1.9	**1.46**	-
Interp.	Img1	4.51	1.2	4.10	1.0	**3.91**	-	3.27	1.1	**2.94**	-
	Img2	17.43	1.5	**11.49**	-	16.88	1.5	**6.22**	-	6.26	1.0
	Img3	77.23	1.2	85.89	1.4	**63.57**	-	**16.23**	-	18.52	1.1

PPO agent. On the other hand, as it requires a programmer's intervention, the developed mechanism is not fully automated, positioning itself as an intermediate model between the manual development of the schedule and the auto-schedule mechanism available in the Halide language. However, the proposed approach is independent of the hardware architecture used. In the present work the approach was evaluated in two architectures, CPU and GPU, but can also be used in other architectures supported by the Halide language.

Results show that the reinforcement learning agent was able to converge to good Halide execution schedules in the evaluated pipelines, on both architectures, although it did not reach the best result in some of the considered scenarios, when compared to the *Hand-tuned* and *Auto-schedule* methods. These results also suggest that the proposed environment, as well as the method of calculating the reward, besides the representation of the states and actions, were effective in representing the problem in a way compatible with the reinforcement learning technique. More test scenarios and a broader set of pipelines need to be evaluated in order to have a comprehensive and reliable indicator.

A point that requires attention is related to the considerable duration of the experiments, which may eventually render the developed solution impracticable for large pipelines. Since most time is spent on compiling each proposed schedule, the use of an agent already trained with one pipeline to optimize the schedule of another pipeline through the use of Transfer Learning techniques [Pan et al. 2010] should be considered. Another enhancement would be to automate the elicitation of scheduling options. A possible implementation could adapt the Halide's available auto-schedule mechanism to use information extracted from the pipeline through the language compiler and generate the set of scheduling options that would then be explored by the reinforcement learning agent.

References

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. (2014). Opentuner: An extensible framework for program autotuning. `http://opentuner.org`. Accessed 2018-01.

Dhariwal, P., Hesse, C., Klimov, O., et al. (2017). Openai baselines. `https://github.com/openai/baselines`. Accessed 2018-09.

Huang, S. (2018). Introduction to various reinforcement learning algorithms, part i: Q-learning, sarsa, dqn, ddpg. `https://towardsdatascience.com/72a5e0cb6287`. Accessed 2018-09.

Júnior, E. P. F. D. (2012). Aprendizado por reforço sobre o problema de revisitação de páginas web. Master's thesis, Pós-Graduação em Informática - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro - RJ.

Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J., and Fatahalian, K. (2016). Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11.

Mullapudi, R. T., Vasista, V., and Bondhugula, U. (2015). Polymage: Automatic optimization for image processing pipelines. *ACM SIGPLAN Notices*, 50(4):429–443.

Ottoni, A. L. C., Oliveira, M. S., Nepomuceno, E. G., and Lamperti, R. D. (2015). Análise do aprendizado por reforço via modelos de regressão logística: Um estudo de caso no futebol de robôs. *Revista Junior de Iniciação Científica em Ciências Exatas e Engenharia*, 1(10):44–49.

Pan, S. J., Yang, Q., et al. (2010). A survey on transfer learning. *IEEE Trans. on Knowledge and Data Engineering*, 22(10):1345–1359.

Ragan-Kelley, J. and Adams, A. (2012). Halide: A language for image processing. `http://halide-lang.org`. Accessed 2018-08.

Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., and Durand, F. (2012). Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530.

Ragan-Kelley, J. M. (2014). *Decoupling algorithms from the organization of computation for high performance image processing*. PhD thesis, MIT.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silva, L. M. D. D. (2016). Proposta de arquitetura em hardware para fpga da técnica q-learning de aprendizagem por reforço. Master's thesis, Pós-Graduação em Engenharia Elétrica e de Computação - Universidade Federal do Rio Grande do Norte, Natal - RN.

Silver, D., Lever, G., Heess, N., Degris, T., et al. (2014). Deterministic policy gradient algorithms. In *International Conference on Machine Learning*.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT press Cambridge.