Optimization of Halide Image Processing Schedules with Reinforcement Learning

Marcelo Pecenin - UFPR
André Murbach Maidl - Elastic
Daniel Weingaertner - UFPR

WSCAD - 2019

Outline

Introduction

- Halide Domain Specific Language
- Reinforcement Learning

Optimization of Halide Schedules

Experimental Results

Conclusion & Future Work

UFPR

Visão Robótica Imagem

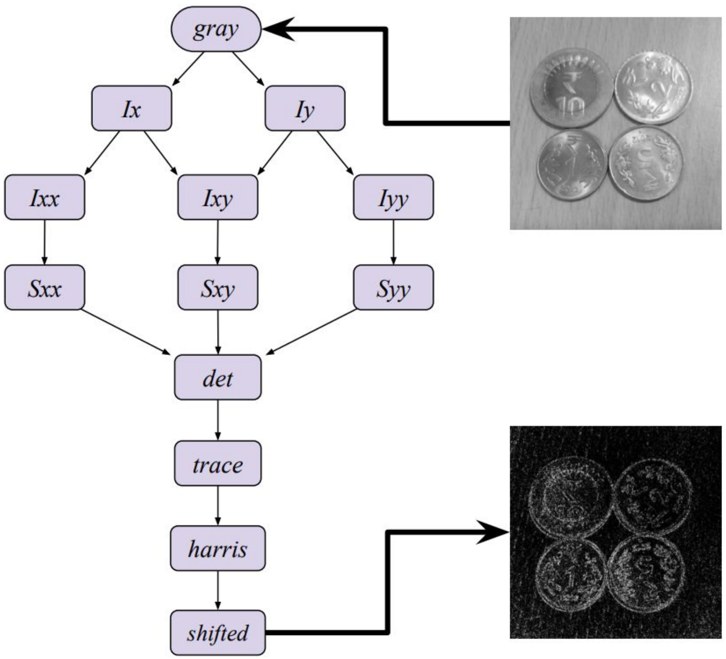
Introduction

Motivation

Image processing pipelines are ubiquitous (cameras, cell phones, streaming)

How to write efficient pipelines considering i.e. different optimization criteria, hardware configuration and image sizes, without reimplementing code?

Introduction



Pipelines have various optimization opportunities

Merging, combining, parallelizing

Generic libraries (OpenCV) are not “pipeline efficient”

Optimized code is hard to maintain

Halide Domain Specific Language

Embedded C++ DSL for image processing

Defines data structures and operations

The algorithm's logic (pipeline) and scheduling are decoupled

- Changing the schedule does not alter the result
- Flexibility to explore different scheduling options

Halide Code Sample - Blur filter

```
void blur_halide() {
  // Initial declarations - Halide data types
  Buffer<uint16_t> input;
  Func blur_x, blur_y;
  Var x, y, yi;

  // Pipeline (Blur Algorithm)
  blur_x(x,y) = (input(x,y) + input(x+1,y) + input(x+2,y)) / 3;
  blur_y(x,y) = (blur_x(x,y) + blur_x(x,y+1) + blur_x(x,y+2)) / 3;

  // Schedule
  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);

  // Compilation and execution
  blur_y.realize(1024, 768);
}
```

Halide’s scheduling directives

Each pipeline stage can have many directives

Each directive can have various parameters

Finding the optimal schedule is a combinatorial problem

Sample Halide scheduling directives:

- `tile(var,var,var,var,num,num)`
- `split(var,var,var,number)`
- `fuse(var,var,var)`
- `reorder(var,var)`
- `compute_root()`
- `compute_inline()`
- `compute_at(func,var)`
- `parallel(var)`
- `vectorize(var,number)`
- `unroll(var)`
- `store_at(func,var)`
- `store_root()`

UFPR

Visão Robótica Imagem

Halide for heterogeneous hardware

Same algorithm but change the schedule for different hardware

CPU

```
harris.tile(x,y,xi,yi,64,64);
harris.vectorize(xi, 8);
harris.parallel(y);
Ix.compute_at(harris, x);
Ix.vectorize(x, 8);
Iy.compute_at(harris, x);
Iy.vectorize(x, 8);
```

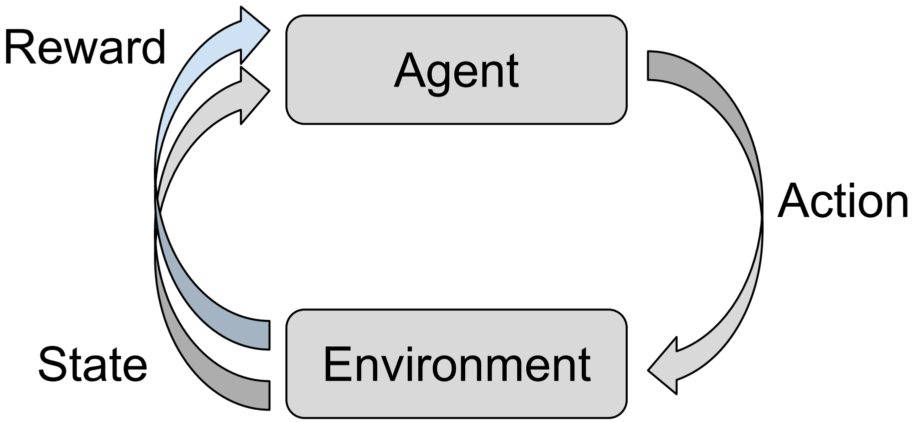
GPU

```
harris.gpu_tile(x,y,xi,yi,14,14);
Ix.compute_at(harris, x);
Ix.gpu_threads(x, y);
Iy.compute_at(harris, x);
Iy.gpu_threads(x, y);
```

Reinforcement Learning

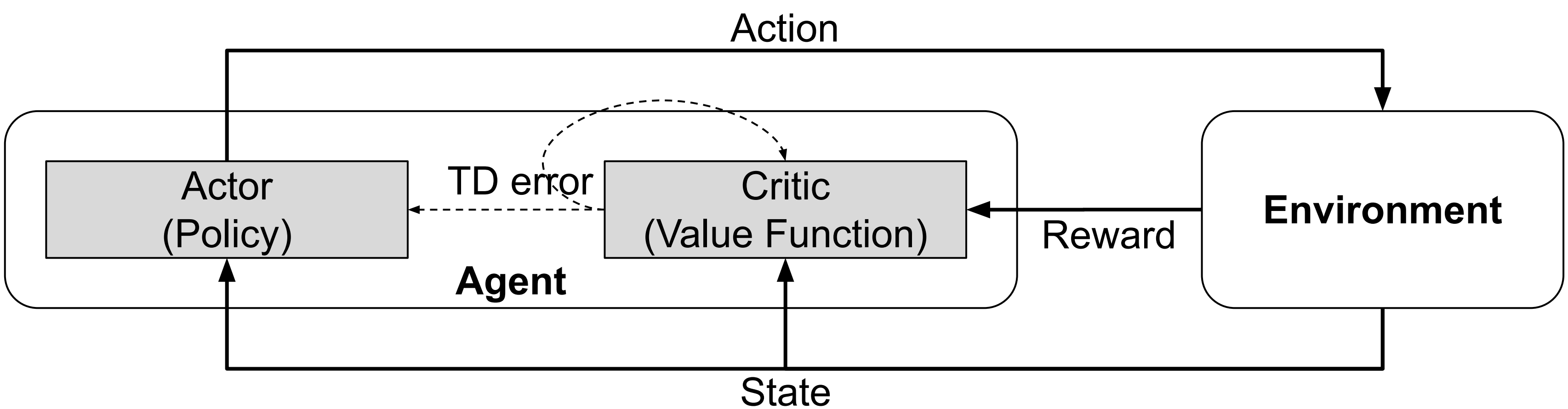
Aims to maximize [future] reward

Takes an action and evaluates the resulting state of the environment and the obtained reward



Proximal Policy Optimization Agent

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



UFPR
Visão Robótica Imagem

Optimization of Halide Schedules

Overview

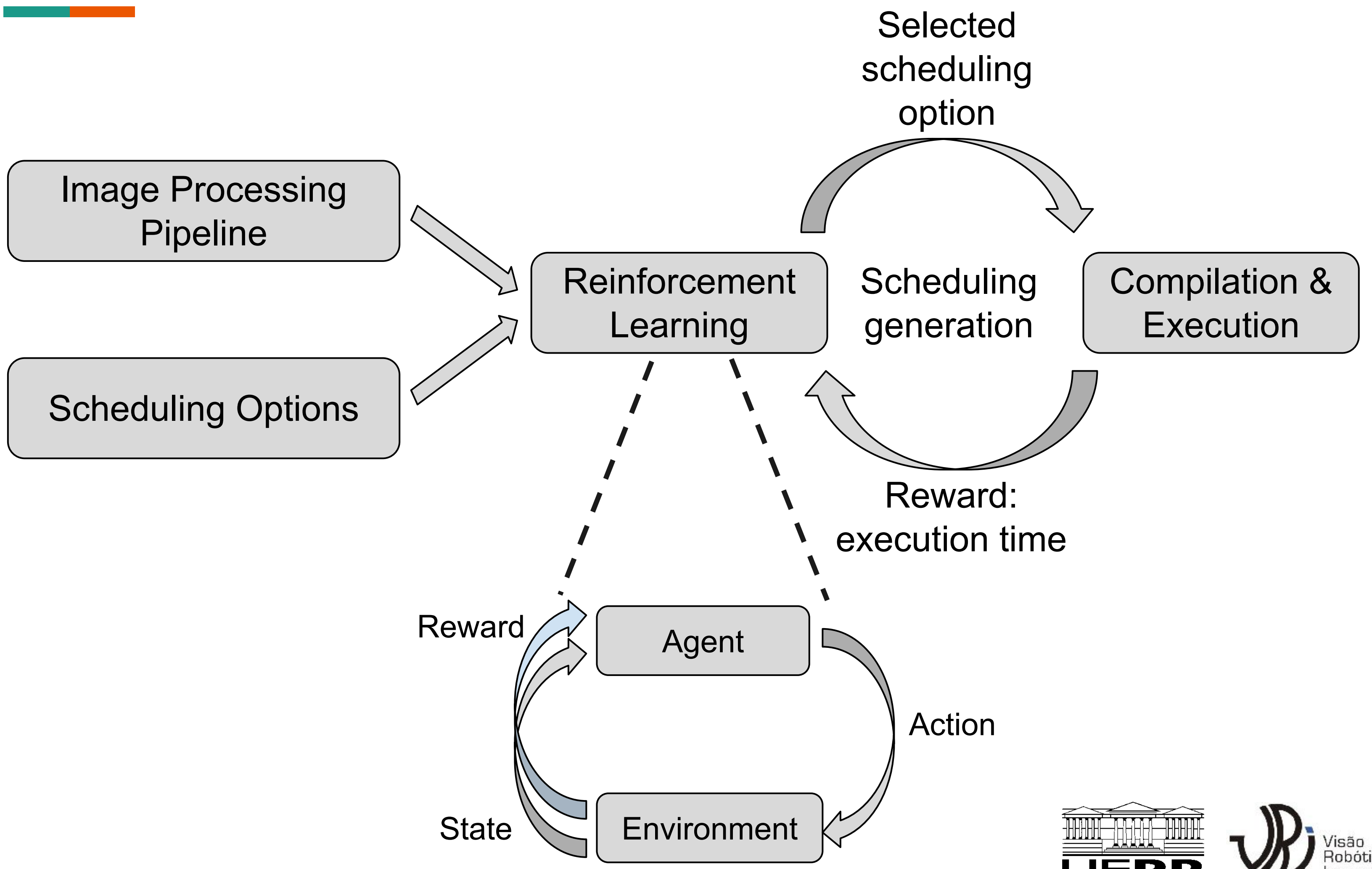


Image Processing Pipeline
Scheduling Options
Reinforcement Learning
Selected scheduling option
Scheduling generation
Compilation & Execution
Reward: execution time
Reward
Agent
State
Environment
Action

Reinforcement Learning

Environment: Halide Pipeline (i.e. Blur filter)

Action: apply a scheduling directive

State: current schedule

Reward: runtime of current schedule

Agent: Proximal Policy Optimization (Schulman et al. 2017)

Mapping the Environment

The programmer has to enlist all possible scheduling directives and their parameters for each pipeline stage

This mapping represents the search space for the PPO agent

Sample environment mapping (Blur)

```
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);
}
```

Experimental Results

Experimental Protocol

Pipelines: Blur (2 stages), Harris corner detection (13 stages) and Piramid interpolation (52 stages)

Comparison to Halide's automatic schedule generator and manual optimization (Mullapudi et al., 2016)

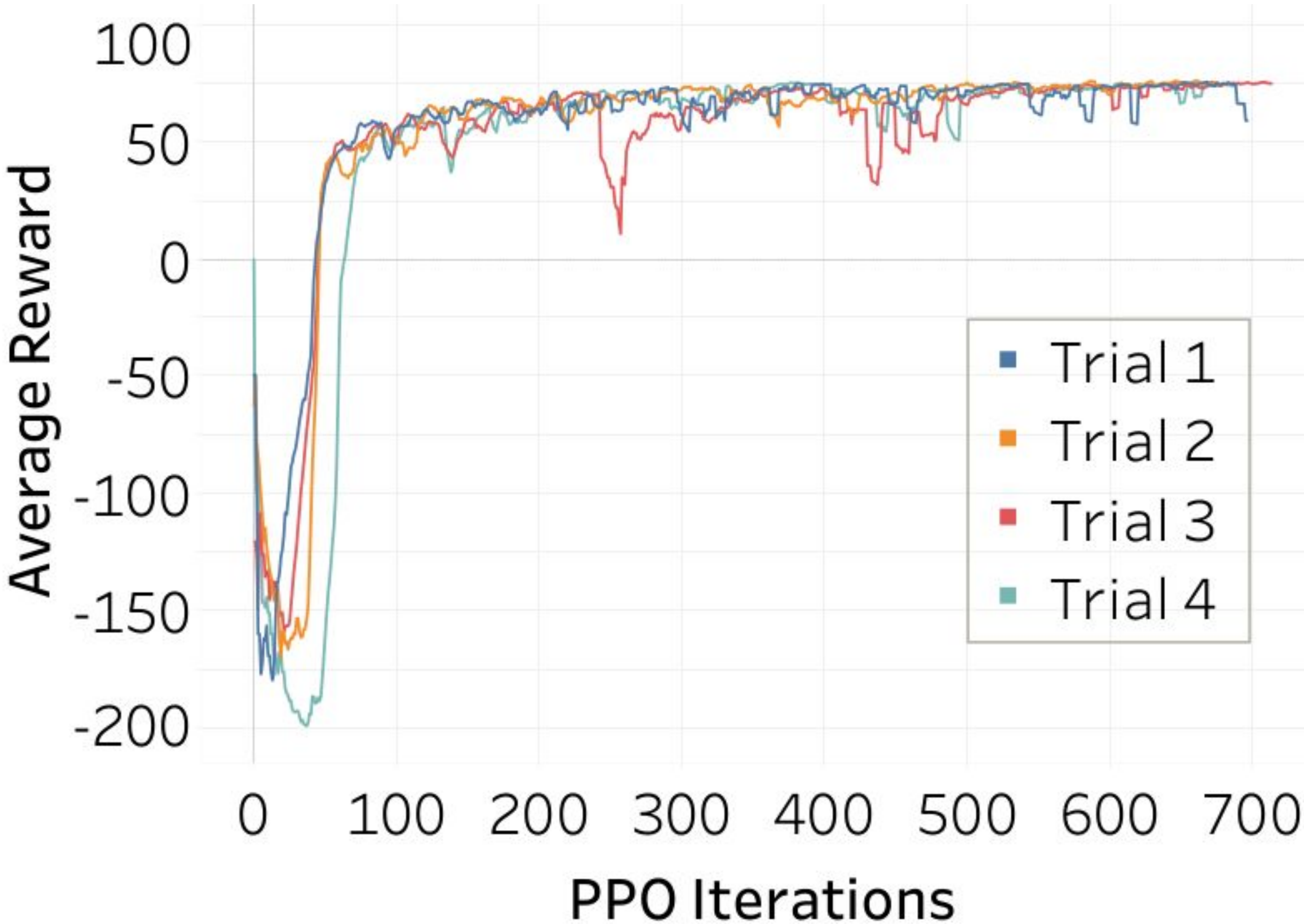
Images of 0.6, 2.4 and 9.8 MPixel

CPU (Intel Xeon) and GPU (Nvidia Tesla V100)

UFPR

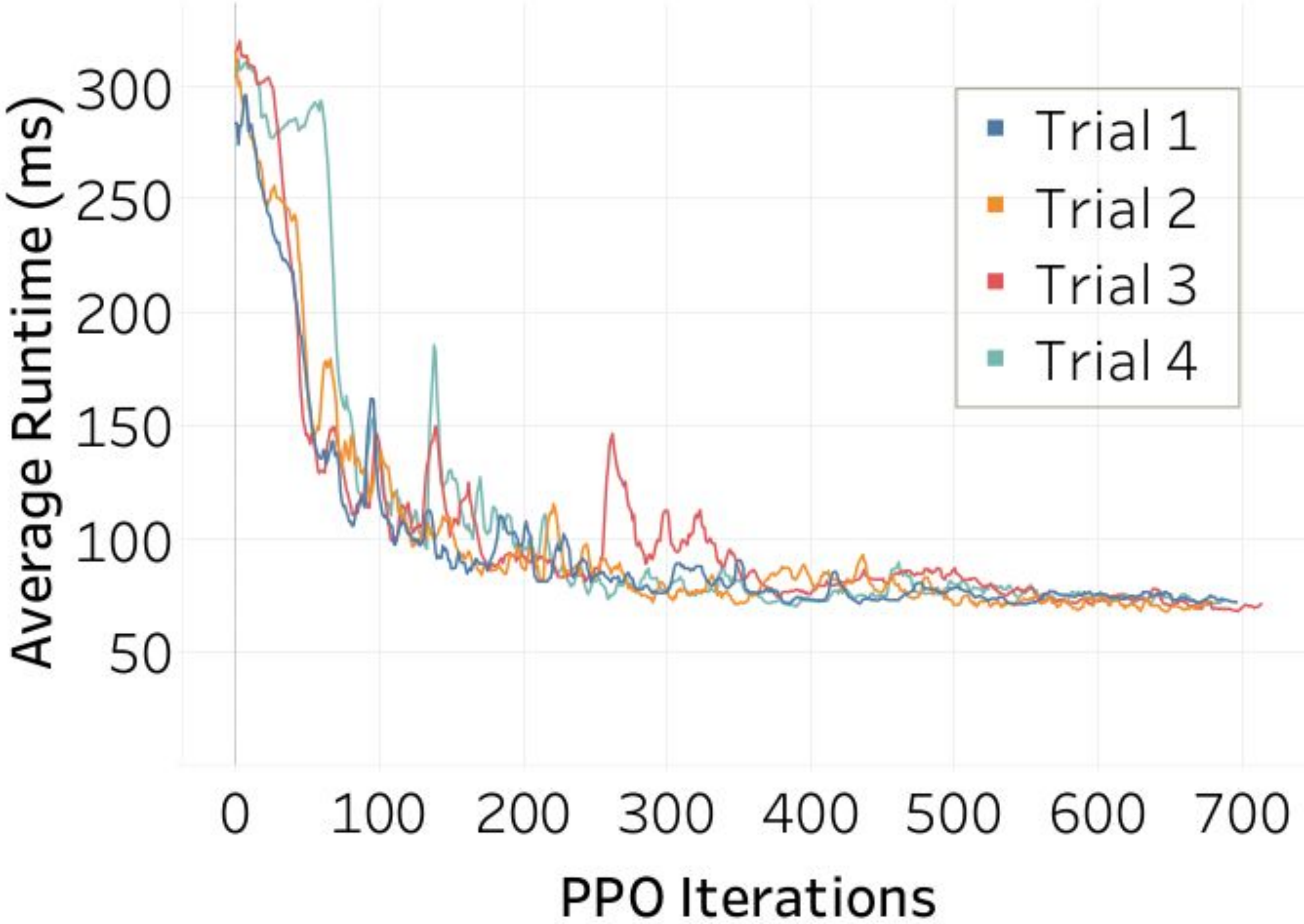
Visão
Robótica
Imagem

PPO Evolution (reward)



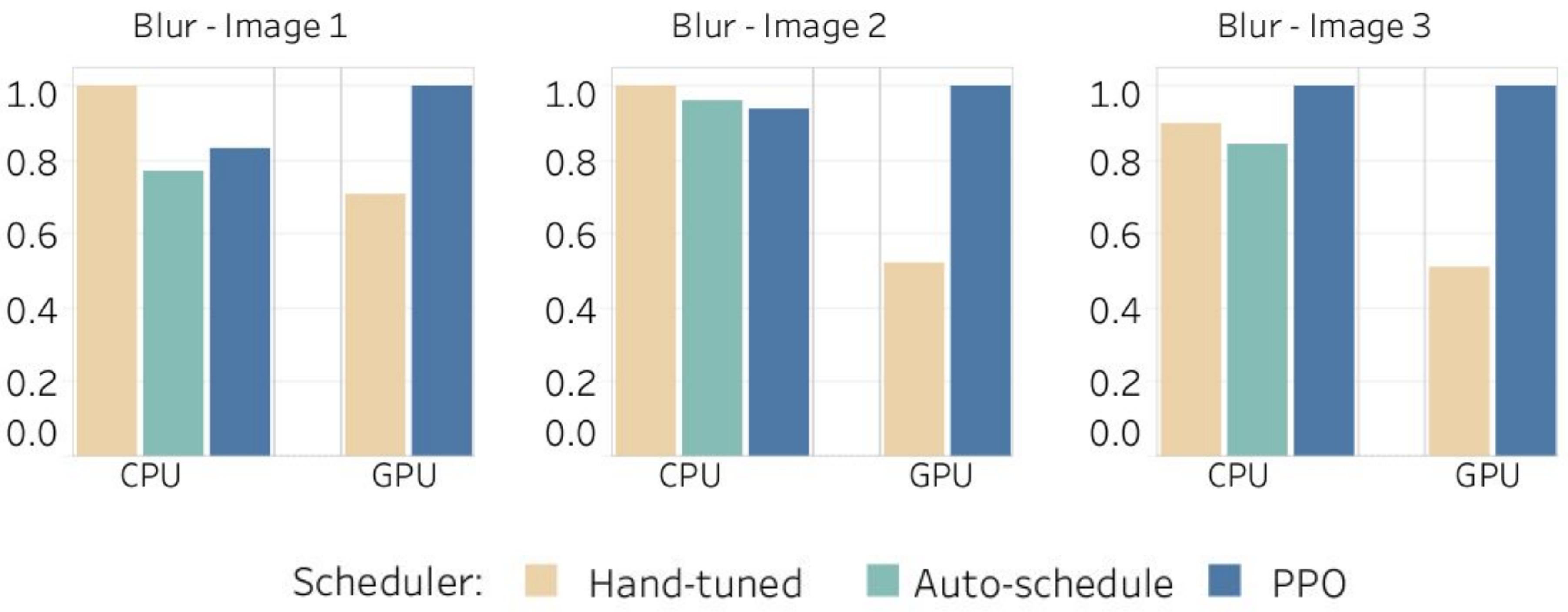
Visão
Robótica
Imagem

PPO Evolution (runtime)



Visão
Robótica
Imagem

Relative Speedup (Blur)



Manual x PPO schedules (Blur)

CPU

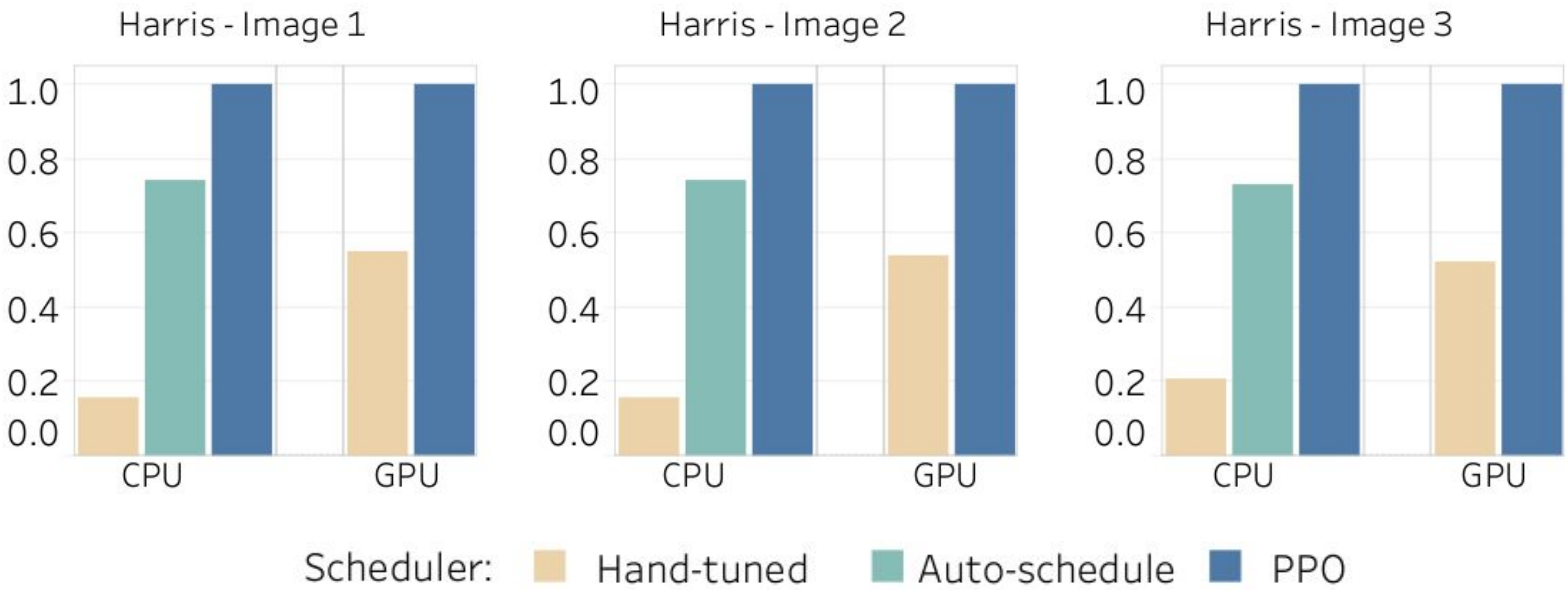
```
// Manual:
blur_y.split(y, y, yi, 8).parallel(y).vectorize(x, 8);
blur_x.store_at(blur_y, y).compute_at(blur_y, yi).vectorize(x, 8);
// PPO:
blur_y.unroll(x, 4).parallel(y).bound(x, 0, input.width());
```

GPU

```
// Manual:
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);
```

UFPR Visão Robótica Imagem

Relative Speedup (Harris)



Manual x PPO schedules (Harris)

CPU

```
// Manual:
shifted.tile(x, y, xi, yi, 128, 128).vectorize(xi, 8).parallel(y);
Ix.compute_at(shifted, x).vectorize(x, 8);
Iy.compute_at(shifted, x).vectorize(x, 8);

// PPO:
shifted.tile(x, y, xi, yi, 512, 8).parallel(y);
gray.compute_at(shifted, x);
Iy.compute_at(shifted, x);
Ix.compute_at(shifted, x);
gray.store_at(shifted, y).unroll(x, 2).unroll(x, 4);
```

Manual x PPO schedules (Harris)

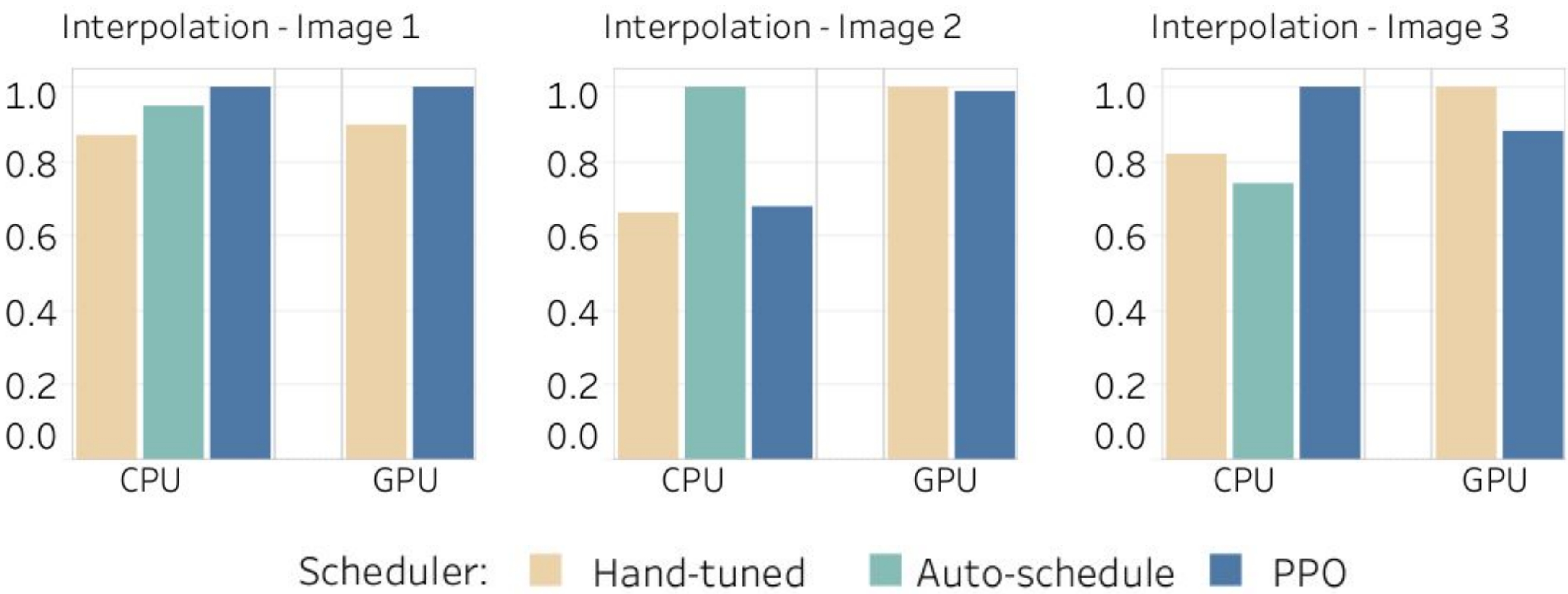
GPU

```
// Manual:
shifted.gpu_tile(x, y, xi, yi, 14, 14);
Ix.compute_at(shifted, x).gpu_threads(x, y);
Iy.compute_at(shifted, x).gpu_threads(x, y);

// PPO:
shifted.compute_root().gpu_tile(x, y, xi, yi, 8, 32);
shifted.bound(y, 0, input.height()).unroll(yi, 2);
gray.compute_at(shifted, x).gpu_threads(x, y);
gray.unroll(y, 2).unroll(x, 3);
```

UFPR Visão Robótica Imagem

Relative Performance (Interpolation)



Comparison

		CPU						GPU			
		Hand-tuned		Auto-schedule		PPO		Hand-tuned		PPO	
		ms	$\times$	ms	$\times$	ms	$\times$	ms	$\times$	ms	$\times$
Blur	Image 1	**0.10**	-	0.13	1.3	0.12	1.2	0.07	1.4	**0.05**	-
	Image 2	**0.49**	-	0.51	1.0	0.52	1.1	0.21	1.9	**0.11**	-
	Image 3	2.94	1.1	3.15	1.2	**2.66**	-	0.76	1.9	**0.39**	-
Harris	Image 1	3.21	6.4	0.68	1.4	**0.50**	-	0.22	1.8	**0.12**	-
	Image 2	13.07	6.2	2.82	1.3	**2.10**	-	0.72	1.8	**0.39**	-
	Image 3	36.89	4.7	10.71	1.4	**7.78**	-	2.79	1.9	**1.46**	-
Interpolation	Image 1	4.51	1.2	4.10	1.0	**3.91**	-	3.27	1.1	**2.94**	-
	Image 2	17.43	1.5	**11.49**	-	16.88	1.5	**6.22**	-	6.26	1.0
	Image 3	77.23	1.2	85.89	1.4	**63.57**	-	**16.23**	-	18.52	1.1

Conclusion and Future Work

Conclusions

The proposed approach was able to generate high performing schedules in a semi-automatic way for different pipelines on CPU and GPU

Environment model was able to represent the problem at hand and appropriate for the chosen reinforcement learning agent

Runtime for generating a good schedule is high and increases with the schedule's complexity

UFPR

Visão Robótica Imagem

Future Work

Use transfer learning to improve convergence on big pipelines and increase the PPO agents generalization capacity

Automatically generate scheduling options

Enrich reward information (cache miss, FLOP, ...)

Thank you!

danielw@inf.ufpr.br