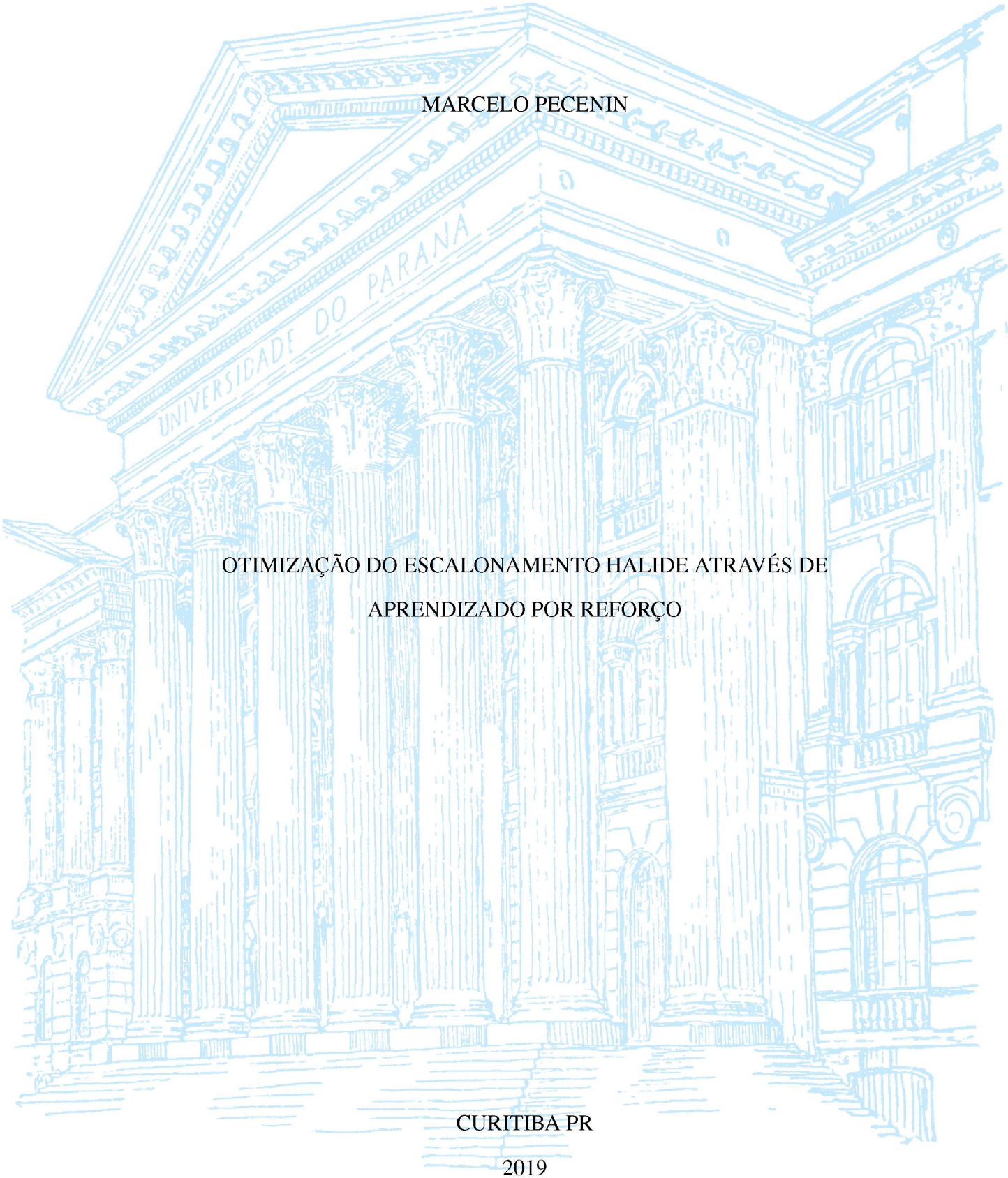
UNIVERSIDADE FEDERAL DO PARANÁ

MARCELO PECENIN




# OTIMIZAÇÃO DO ESCALONAMENTO HALIDE ATRAVÉS DE APRENDIZADO POR REFORÇO

CURITIBA PR

2019

MARCELO PECENIN

# OTIMIZAÇÃO DO ESCALONAMENTO HALIDE ATRAVÉS DE APRENDIZADO POR REFORÇO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação.*

Orientador: Daniel Weingaertner.

Coorientador: André Murbach Maidl.

CURITIBA PR

2019

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

P365o Pecenin, Marcelo
Otimização do escalonamento Halide através de aprendizado por reforço [recurso eletrônico] / Marcelo Pecenin – Curitiba, 2019.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática.
Orientador: Daniel Weingaertner.
Coorientador: André Murbach Maidl.

1. Halide – Linguagem de programação (Computadores). 2. Código de otimização (informática). I. Universidade Federal do Paraná. II. Weingaertner, Daniel. III. Maidl, André Murbach. IV. Título.

CDD: 005.275

Bibliotecária: Roseny Rivelini Morciani CRB-9/1585

MINISTÉRIO DA EDUCAÇÃO
SETOR SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA - 40001016034P5

# TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **MARCELO PECENIN** intitulada: **Otimização do Escalonamento Halide através de Aprendizado por Reforço**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua APROVADO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 27 de Fevereiro de 2019.

DANIEL WEINGAERTNER
Presidente da Banca Examinadora (UFPR)

ANDRE MURBACH MAIDL
Avaliador Externo (ELASTIC)

RODRIGO MINETTO
Avaliador Externo (UTFPR)

MARCOS DIDONET DEL FABRO
Avaliador Interno (UFPR)

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

*À minha família, aos professores e amigos que contribuíram para minha formação.*

## AGRADECIMENTOS

A Deus, por me abençoar com saúde e dar-me força para prosseguir.

Aos orientadores, Daniel e André, pelo apoio e orientação na elaboração deste trabalho.

Ao Departamento de Informática da UFPR e todos seus professores, Fundação PTI e Itaipu, por viabilizarem a realização deste programa de mestrado.

A todos amigos e colegas que percorreram juntos esta jornada.

# RESUMO

Escrever programas que usufruam da melhor maneira a capacidade de processamento oferecida por uma determinada arquitetura de hardware não é uma tarefa trivial e torna-se cada vez mais trabalhosa à medida em que aumenta a diversidade de equipamentos e plataformas. Em uma linguagem de programação de propósito geral esta tarefa depende muito do compilador e do conhecimento e experiência do programador. Neste trabalho foi avaliado o potencial que uma linguagem de domínio específico chamada Halide oferece para facilitar a implementação de programas de processamento de imagens otimizados para diferentes arquiteturas. A partir de uma característica desta linguagem, chamada escalonamento de execução, é possível empregar mecanismos automatizados para produzir uma implementação com melhor desempenho usufruindo de características específicas do hardware, sem afetar a definição lógica e resultados do programa. Trabalhos anteriores empregaram algoritmos genéticos e diversas técnicas de busca para tentar gerar escalonamentos otimizados, porém não chegaram a uma solução efetiva. No presente trabalho foi desenvolvido uma abordagem de geração do escalonamento de execução empregando aprendizado de máquina, especificamente aprendizado por reforço. O problema foi modelado como um processo de decisão de Markov e incorporado a um ambiente de aprendizado por reforço utilizando um agente que agrega funções de aproximação para espaços de grandes dimensões, baseadas em redes neurais, chamado PPO (*Proximal Policy Optimization*). Experimentos foram realizados em duas arquiteturas de hardware (CPU e GPU), utilizando programas de processamento de imagens com diferentes níveis de complexidade. Os escalonamentos produzidos pelo agente PPO foram comparados com versões implementadas manualmente por programadores Halide e com soluções geradas pelo recurso de escalonamento automático suportado na linguagem Halide. Os resultados mostram que nos cenários avaliados a abordagem desenvolvida foi capaz de produzir escalonamentos de execução competitivos com as demais soluções comparadas, superando o desempenho em vários casos. Contudo, o modelo desenvolvido ainda não é totalmente automatizado, necessitando da intervenção do programador para mapear quais diretivas de escalonamento serão exploradas pelo agente de aprendizado por reforço.



Palavras-chave: Aprendizado por Reforço. Otimização de Código. Linguagem Halide.

## ABSTRACT

Writing programs that best utilize the processing power offered by a particular hardware architecture is not a trivial task and becomes increasingly laborious as the diversity of equipment and platforms increases. In a general-purpose programming language, this task relies heavily on the compiler and the knowledge and experience of the programmer. In this work, we evaluate the potential that a specific domain language called Halide offers to facilitate the implementation of optimized image processing programs for different architectures. From a feature of this language, called execution scheduling, it is possible to employ automated mechanisms to produce a better performing implementation, taking advantage of the specific hardware characteristics, without affecting the logical definition and the program results. Previous works has employed genetic algorithms and several search techniques to try to generate optimized schedules, but they have not reached an effective solution. In the present work was developed an approach of execution scheduling generation using machine learning, specifically reinforcement learning. The problem was modeled as a Markov decision process and incorporated into a reinforcement learning environment using an agent that adds approximation functions for large spaces, based on neural networks, called PPO (Proximal Policy Optimization). Experiments were performed on two hardware architectures (CPU and GPU), using image processing programs with different levels of complexity. The schedules produced by the PPO agent were compared to versions implemented manually by Halide programmers and with solutions generated by the automatic scheduling feature supported in the Halide language. The results show that in the evaluated scenarios the developed approach was able to produce competitive execution schedules compared with the other solutions, surpassing the performance in several cases. However, the developed model is not yet fully automated, requiring the intervention of the programmer to map which scheduling directives will be explored by the reinforcement learning agent.



Keywords: Reinforcement Learning. Code Optimization. Halide Language.

## LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ACRÔNIMOS

| | |
|---|---|
| ASIC | *Application Specific Integrated Circuit* |
| CPU | *Central Processing Unit* |
| CUDA | *Compute Unified Device Architecture* |
| DAG | *Directed Acyclic Graph* |
| DQN | *Deep Q-Network* |
| DSL | *Domain Specific Language* |
| DT | Diferença Temporal |
| Expr | Expressão de Funções Halide |
| FPGA | *Field Programmable Gate Array* |
| Func | Função/Estágio do *Pipeline* Halide |
| GAE | *Generalized Advantage Estimation* |
| GPU | *Graphics Processing Unit* |
| LLVM | *Low Level Virtual Machine Compiler Infrastructure* |
| MDP | *Markov Decision Process* |
| MLP | *Multilayer Perceptron* |
| OpenCL | *Open Computing Language* |
| OpenCV | *Open Source Computer Vision Library* |
| PPO | *Proximal Policy Optimization* |
| Rdom | *Reduction Domain* |
| RPC | *Remote Procedure Call* |
| SIMD | *Single Instruction Multiple Data* |
| Tanh | Função de Ativação Tangente Hiperbólica |
| Var | Variável/Domínio de Funções Halide |

# LISTA DE SÍMBOLOS

| | |
|---|---|
| $r$ | Recompensa imediata de uma ação |
| $R$ | Recompensa futura acumulada |
| $t$ | Passo de tempo discreto |
| $a$ | Ação escolhida pelo agente |
| $s$ | Estado do ambiente |
| $\gamma$ | Fator de desconto da recompensa futura |
| $\epsilon$ | Probabilidade de uma ação randômica |
| $Q$ | Função de valor do estado-ação |
| $\pi$ | Política de escolha de ações |
| $V$ | Função de valor do estado |
| $\delta$ | Erro de diferença temporal |
| $A$ | Função de vantagem |
| $r'$ | Razão de probabilidade da política no PPO |
| $\epsilon'$ | Limiar da razão de probabilidade da política no PPO |

## SUMÁRIO

# 1 INTRODUÇÃO

A busca por melhor desempenho e capacidade de processamento gera ao longo do tempo uma diversidade de plataformas e soluções de hardware, cada qual com recursos e características próprias, que podem eventualmente ser integrados ou operar em conjunto para desempenhar uma determinada tarefa. Um exemplo disso é a computação heterogênea (Stringhini et al., 2012), que emprega diferentes aceleradores a um mesmo equipamento para dotá-lo de uma capacidade superior de processamento. Nesse contexto diversificado e heterogêneo o desenvolvimento de software com portabilidade para plataformas com diferentes capacidades e características tem sua importância realçada, pois desenvolver uma implementação própria, otimizada para cada novo tipo de hardware, torna-se uma tarefa muito onerosa e assim inviável em muitos casos (Sarkar e Alavani, 2018). Tem-se então um desafio: como desenvolver software portáveis que sejam capazes de aproveitar adequadamente toda a capacidade oferecida por diferentes arquiteturas de hardware (Ravishankar et al., 2015).

Uma das áreas que atualmente demandam grandes capacidades de processamento é a área de processamento de imagens, a qual é utilizada direta ou indiretamente em diferentes áreas de conhecimento, como reconhecimento de padrões, automação, jogos, medicina, pesquisas espaciais, realidade aumentada, e muitas outras. Nesse contexto, Ragan-Kelley et al. (2012) desenvolveu uma linguagem de domínio específico (DSL - *Domain Specific Language*)(Hudak, 1997), chamada Halide, voltada para processamento de imagens, cuja representação alto nível do código fonte visa aumentar a produtividade e facilitar a portabilidade no desenvolvimento de programas desse domínio. Pela sua estrutura e domínio específico, a linguagem também consegue abstrair grande parte dos detalhes necessários para gerar um código executável otimizado para diferentes arquiteturas de hardware, usufruindo do paralelismo, hierarquia de memória, e de instruções de baixo nível que conseguem acessar recursos avançados específicos de cada arquitetura suportada.

Halide é uma das linguagens estado-da-arte em processamento de imagens, usada em produtos de grande abrangência como o Google Photos e em filtros do aplicativo Câmera para Android Nexus 5 (Denniston, 2016; Ragan-Kelley, 2014), sendo também usada na Google para facilitar a programação para o coprocessador Pixel Visual Core que está integrado em todos os aparelhos Google Pixel 2 (Shacham e Reynders, 2017).

Uma das características mais distintas da linguagem Halide é o desacoplamento entre a lógica funcional do programa e sua organização para execução, chamada na linguagem de escalonamento de execução. Esse desacoplamento é viabilizado pela DSL do Halide que limita as operações e estruturas de dados que o programador pode utilizar. Com essa característica de desacoplamento é possível fazer toda a implementação da lógica do algoritmo e, depois, sem afetar o resultado gerado, desenvolver e experimentar organizações totalmente diferentes de execução (Ragan-Kelley, 2014). Justamente neste aspecto reside um desafio para o programador, pois precisa encontrar para cada arquitetura de hardware uma organização otimizada, que proporcione o melhor desempenho durante a execução. O problema nesse ponto é que à medida que o tamanho dos programas cresce a quantidade de organizações possíveis também cresce significativamente, tornando-se uma tarefa difícil encontrar bons escalonamentos de execução, mesmo para programadores experientes (Mullapudi et al., 2016).

A existência dessa dificuldade aliada à característica da linguagem que permite desenvolver o escalonamento sem afetar a parte lógica do programa, conduz ao anseio por tentar automatizar esse processo. Algumas abordagens para automatizar a geração do escalonamento

de execução já foram experimentadas na literatura, entre elas a geração do escalonamento usando algoritmo genético (Ragan-Kelley et al., 2013), outra usando conjunto de técnicas de busca (Ansel et al., 2014a), e uma mais recente, ainda em evolução, chamada de escalonamento automático (Mullapudi et al., 2016), que sugere um escalonamento a partir de análises da definição do programa. Entretanto, Li et al. (2018) mencionam que a geração automática do escalonamento Halide de maneira geral ainda é um problema não resolvido, pois apesar de terem obtido bom desempenho nos cenários abordados no trabalho, citam que ao inspecionar o código gerado notaram que ainda havia grande espaço para melhorias adicionais no escalonamento.

O presente trabalho aplicou uma abordagem de aprendizado de máquina, especificamente aprendizado por reforço, para o problema de gerar escalonamentos de execução otimizados para programas escritos em Halide.

## 1.1 OBJETIVO DO TRABALHO

O objetivo do trabalho é desenvolver um sistema baseado em agentes de aprendizado por reforço para a geração e otimização automática de escalonamentos de execução para programas da linguagem Halide. Os objetivos específicos são:

- Criar um mecanismo de representação do escalonamento de execução de programas Halide para integração com o ambiente de aprendizado por reforço.
- Desenvolver um ambiente (*environment*) de aprendizado por reforço para interagir com programas Halide.
- Treinar um agente de aprendizado por reforço na geração dos escalonamentos de execução.
- Comparar o desempenho dos escalonamentos gerados com o estado-da-arte.

## 1.2 CONTRIBUIÇÕES DO TRABALHO

Em consonância com o objetivo apresentado, as principais contribuições derivadas do presente trabalho são:

- Uma abordagem que permite utilizar o aprendizado por reforço na otimização do escalonamento de programas Halide para duas arquiteturas de hardware: CPU e GPU.
- Um ambiente (*environment*) de aprendizado por reforço para interagir com programas Halide usando interfaces já padronizadas na literatura e assim compatível com diversas bibliotecas e algoritmos existentes de aprendizado por reforço.
- Resultados experimentais da aplicação do aprendizado por reforço para geração e otimização automática do escalonamento de programas Halide.
- Comparação entre os resultados obtidos com a abordagem de aprendizado por reforço e outras abordagens disponíveis.

## 1.3 ORGANIZAÇÃO DO TRABALHO

No capítulo 2 é apresentado uma introdução à linguagem Halide e seus principais elementos, incluindo exemplos de escalonamentos de execução. Em seguida, no capítulo 3 são descritos conceitos relacionados ao aprendizado por reforço e detalhado características de agentes de aprendizado por reforço baseado em rede neural. No capítulo 4 são apresentadas abordagens utilizadas em trabalhos relacionados. No capítulo 5 é descrita a abordagem desenvolvida neste trabalho. No capítulo 6 são listadas as configurações, programas e cenários de teste utilizados nos experimentos realizados. Por fim, no capítulo 7 são apresentados os resultados obtidos e, no capítulo 8, a conclusão e sugestões de trabalhos futuros.

# 2 LINGUAGEM HALIDE

Halide é uma linguagem de programação projetada para facilitar a escrita de código de alto desempenho para processamento de imagens em máquinas modernas, de arquiteturas mais recentes, gerando código capaz de aproveitar a localidade de memória (*memory locality*), vetorização (instruções SIMD) e paralelismo em CPUs e GPUs multicore (Ragan-Kelley e Adams, 2012). Halide é implementada como uma DSL (Linguagem de Domínio Específico) embarcada na linguagem C++ e distribuída como uma biblioteca. A linguagem Halide permite compilar para várias plataformas, como X86, ARM, CUDA, OpenCL e OpenGL nos sistemas OS X, Linux e Windows.

DSL se refere a uma linguagem de programação que foi criada com o objetivo de solucionar problemas em um domínio específico de aplicação, que no caso do Halide refere-se ao processamento de imagens. Ao contrário das linguagens de propósito geral, uma DSL possui expressividade limitada, direcionando o usuário a utilizar estruturas que modelam de forma clara e concisa funcionalidades específicas de um determinado domínio, aumentando o nível de abstração e reduzindo detalhes de implementação. Halide é um tipo de DSL categorizada como interna, ou embarcada, pois utiliza a infraestrutura de uma linguagem de programação existente, também chamada de linguagem nativa, ou hospedeira, sendo assim implementada como bibliotecas e componentes que estendem a linguagem nativa, no caso, a linguagem C++, criando novos tipos de dados, rotinas, procedimentos, etc (Fowler e Parsons, 2010).

A principal inovação da linguagem Halide é o desacoplamento entre as definições do algoritmo (lógica) e da organização de como este deve ser executado (escalonamento), ou seja, no Halide o código que implementa a heurística do algoritmo de processamento de imagem fica separado do código que define o aninhamento de laços de processamento, instruções de paralelismo, vetorização, etc (Ragan-Kelley et al., 2012). Esse desacoplamento foi viabilizado justamente pela construção da DSL Halide, que limita e expressividade da linguagem permitindo ao programador utilizar apenas operações e estruturas de dados previamente existentes na linguagem, específicas para o domínio de processamento de imagem, provendo assim condições para que o compilador da linguagem Halide analise e ajuste o código executável que será gerado. Normalmente essas definições (lógica e escalonamento) são intercaladas e acopladas entre si, de modo que, por exemplo, mudanças no escalonamento de execução requer mudanças na lógica do algoritmo. No Halide, mudanças no escalonamento não afetam a definição da lógica do algoritmo, facilitando ao programador experimentar diferentes escalonamentos e escolher o melhor.

Os algoritmos de processamento de imagens podem ser implementados como um *pipeline* de processamento composto por vários estágios que realizam operações matemáticas sobre dados recebido de estágios anteriores e disponibilizam novos dados para estágios seguintes, até concluir todo o processamento do algoritmo. Um *pipeline* pode ser representado como um grafo conectando diferentes estágios, no qual cada estágio pode conter operações *stencil* (máscara/janela), aritméticas, lógicas, dependência de dados de outros estágios, etc. A figura 2.1 exibe uma representação de um *pipeline* de processamento de imagem composto por diversos estágios, ilustrados como retângulos, conectados por setas que indicam o fluxo de dados entre eles. O *pipeline* inicia recebendo uma imagem de entrada e termina produzindo uma outra imagem de saída. Neste caso o processamento realizado pelo *pipeline* corresponde a um algoritmo para detecção de cantos presentes na imagem de entrada utilizada.

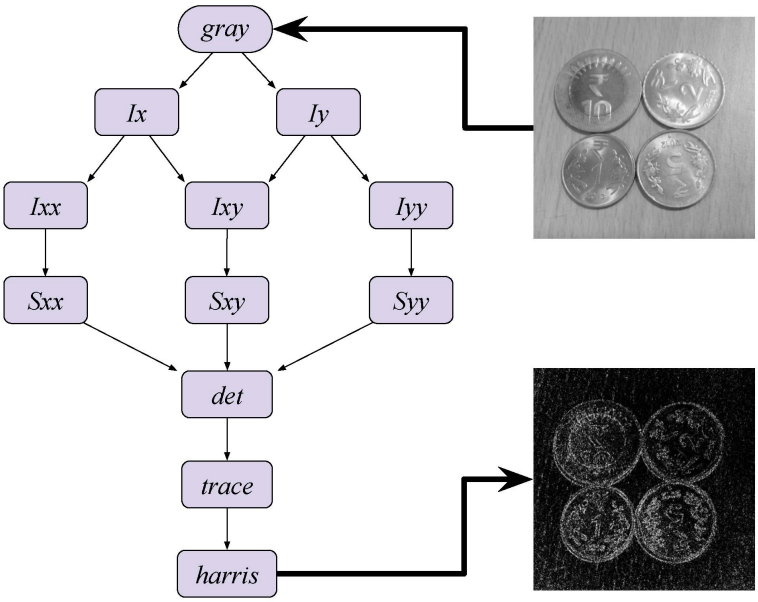



Figura 2.1: Representação de um *pipeline* de processamento de imagens para detecção de cantos. Adaptado de Mullapudi et al. (2015). Os estágios do *pipeline*, ilustrados como retângulos, estão conectados por setas que indicam o fluxo de dados entre eles. Cada estágio realiza operações matemáticas sobre os dados recebidos disponibilizando o resultado para estágios seguintes.

De acordo com o exposto por Ragan-Kelley et al. (2013), implementações eficientes de *pipelines* de processamento de imagens requerem otimizações do paralelismo e da localidade de memória, mas devido à natureza das operações *stencil*, é necessário ponderar a possibilidade de recomputação redundante de valores compartilhados, analisando assim o *trade-off* entre esses três aspectos: paralelismo, localidade e recomputação. Halide analisa esses aspectos usando um modelo sistemático que combina informações das definições do algoritmo e do escalonamento, conforme ilustrado na figura 2.2.

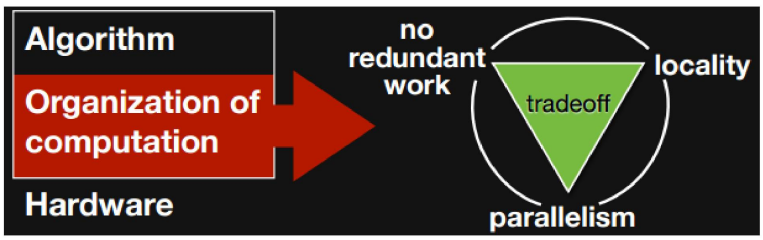



Figura 2.2: Aspectos ponderados para otimização no Halide (Durand, 2015a). São considerados três aspectos: paralelismo, localidade de memória e processamento redundante de valores compartilhados. Esses aspectos são analisados usando informações da definição do algoritmo e da definição do escalonamento de execução para um determinado hardware.

## 2.1 ESTRUTURA DA LINGUAGEM

No Halide o *pipeline* de processamento de imagens é descrito através de uma DSL de modelo funcional. Seus principais elementos segundo Durand (2015a) são:

- **`Func`:** É uma função definida sobre um domínio de inteiros e cada função representa um estágio do *pipeline* de processamento. `Func` é o elemento central do Halide.

- **`Var`:** São variáveis abstratas que representam o domínio sobre os quais as `Funcs` atuam.
- **`Expr`:** São expressões aritméticas compostas por `Funcs` e `Vars` utilizando operadores e funções aritméticas. `Funcs` são definidas por expressões.
- **`Buffer/Image`:** Estrutura multidimensional que representa a entrada e saída de dados (imagens).

A DSL do Halide é implementada como uma biblioteca para C++, assim, esses elementos da linguagem como `Func`, `Var`, `Expr` e `Buffer/Image` são representados usando classes, estruturas e sobrecarga de operadores. A declaração desses elementos no código C++ representa a instanciação da classe ou estrutura correspondente.

A definição de uma função (`Func`) ou expressão (`Expr`) faz uso da sobrecarga de operadores como =, `()`, +, etc. Já a definição do escalonamento de execução (organização da computação) é realizado através de métodos definidos na classe do elemento `Func`, conhecidos como **diretivas de escalonamento**.

A seguir, no código 2.1, é apresentado um exemplo de programa escrito usando a linguagem DSL do Halide.

Código 2.1: Exemplo de Programa Halide.

```
1 Image<float> in = Tools::load_image<float>("images/gray.png");
2 Func input = BoundaryConditions::repeat_edge(in);
3 Func blur_x, blur_y;
4 Var x, y;
5
6 blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y)) / 3;
7 blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1)) / 3;
8
9 Image<float> output = blur_y.realize(in.width(), in.height());
```

O código 2.1 apresentado implementa um algoritmo de suavização de imagens (*blur*). A semântica do código é descrita nos passos a seguir:

- Na linha 1, uma imagem é carregada, usando a função utilitária `load_image`, para um elemento `Image` do Halide, representando cada pixel como ponto flutuante (float).
- Na linha 2, é configurado uma estratégia para controlar o limite das bordas da imagem.
- Nas linhas 3 e 4, são declaradas as funções Halide `blur_x` e `blur_y`, bem como as variáveis Halide `x` e `y`.
- Nas linhas 6 e 7, é definido o corpo das funções `blur_x` e `blur_y`, as quais trabalham sobre o domínio `x` e `y` que, por convenção, no Halide correspondem a "coluna" e "linha" na imagem, nessa ordem.
- No caso da função `blur_x` observa-se que ela depende da entrada `input` em três posições de domínio relativas ao atual valor das variáveis `x` e `y`.
- A função `blur_y`, por sua vez, depende da função anterior `blur_x`, demonstrando assim a relação de dependência que pode ocorrer entre diferentes funções em Halide.

- Na linha 9, o *pipeline* é executado através do método `realize`, o qual também faz a compilação e geração de código de máquina necessário. Os argumentos informados definem o tamanho desejado para o domínio `x` e `y`. Note que nesse caso não foi definido um escalonamento de execução, assim o Halide usa uma definição default, sequencial e inline, não otimizada para o hardware disponível.

Pela característica funcional da linguagem Halide, os laços de processamento são implícitos, ou seja, as iterações necessárias para percorrer todos os valores possíveis do domínio utilizado são automaticamente gerados e controlados pelo Halide. Além disso, as funções Halide não apresentam efeito colateral (Durand, 2015a; Ragan-Kelley et al., 2013), ou seja, não podem alterar ou armazenar valores em outros `buffers` ou funções e, como consequência, a linguagem não suporta utilizar junto de uma função Halide outras estruturas de dados, como por exemplo, uma estrutura de pilha. Funções em Halide apenas mapeiam coordenadas (domínios) de números inteiros para resultados escalares de tamanho estático. Recursão de propósito geral ou resultados de tamanho dinâmico não são permitidos (Ragan-Kelley, 2014).

Entretanto, Halide suporta uma forma limitada de recursão através de um recurso chamado "*update definition*", ou seja, além da definição inicial de uma função Halide, chamada "*pure definition*", é possível definir uma ou mais atualizações da definição, as quais podem redefinir o valor inicial de pontos cujas coordenadas específicas possam ser obtidas por expressões ou variáveis Halide que dependam de valores definidos ou computados previamente (Ragan-Kelley, 2014). Esse recurso, por exemplo, permite expressar operações como Histograma. Halide também não permite a existência de condições de concorrência, mesmo que usando o recurso de *update definition* (Sharlet, 2015). Segundo Ragan-Kelley (2014) a linguagem Halide não é Turing completa, uma vez que os *pipelines* são apenas *feed-forward* e recursões devem ter profundidade delimitada.

No código 2.2 é apresentado um exemplo de como um escalonamento de execução mais otimizado é definido no Halide, para o mesmo *pipeline* do código 2.1 anterior, apenas acrescentando algumas linhas antes da execução, `realize`, porém sem qualquer mudança na lógica do algoritmo.

Código 2.2: Exemplo de Escalonamento Halide.

```
1  Image<float> in = Tools::load_image<float>("images/gray.png");
2  Func input = BoundaryConditions::repeat_edge(in);
3  Func blur_x, blur_y;
4  Var x, y, xi, yi;
5
6  // Algoritmo / pipeline de processamento
7  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y)) / 3;
8  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1)) / 3;
9
10 // Escalonamento de execução
11 blur_y.tile(x, y, xi, yi, 8, 4).parallel(y).vectorize(xi, 8);
12 blur_x.compute_at(blur_y, x).vectorize(x, 8);
13
14 Image<float> output = blur_y.realize(in.width(), in.height());
```

Utilizando de diretivas de escalonamento disponibilizadas pela linguagem Halide, o código 2.2 define o seguinte escalonamento de execução:

- O escalonamento inicia na linha 11, com a diretiva `tile`, que divide a imagem principal em janelas (*tiles*) menores de tamanho 8x4, vinculando as dimensões dessas janelas às novas variáveis `xi` e `yi`. O uso de janelas melhora a localidade de memória.

- Ainda na linha 11, a diretiva `parallel` define uma execução paralela para cada janela.
- No final da linha 11, a diretiva `vectorize` estabelece a utilização de vetorização (instruções SIMD) no processamento de cada linha, assim, 8 posições (colunas) da janela são processadas simultaneamente.
- Na linha 12, a função `blur_x` é configurada para ser processada dentro do domínio da função `blur_y`, vetorizando o processamento de cada linha de `blur_x`.

Na figura 2.3 é apresentado uma ilustração simplificada de como ocorre o processamento do *pipeline*, de acordo com o escalonamento definido anteriormente. Cada *tile* é executado em paralelo, e dentro deste cada função (estágio) é processado conforme necessário para produzir a saída, ocorrendo reprocessamento nas regiões sobrepostas entre os *tiles*.

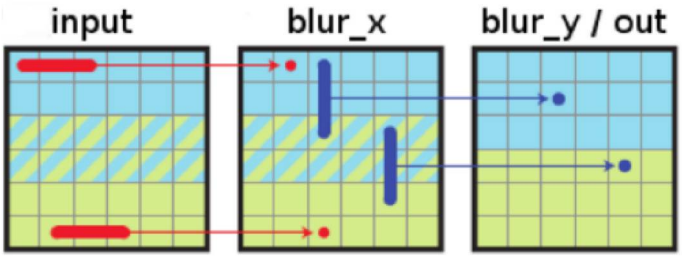



Figura 2.3: Ilustração de um exemplo de escalonamento Halide com processamento paralelo e vetorização usando múltiplos estágios. Adaptado de Ragan-Kelley et al. (2013).

Quando estiver utilizando aceleradores gráficos (GPU), no Halide o *pipeline* continua o mesmo, sem alterações, porém o escalonamento de execução é diferente, devendo utilizar algumas diretivas próprias para GPU, como por exemplo, `gpu_blocks` e `gpu_threads`, conforme apresentado no código 2.3.

Código 2.3: Exemplo de Escalonamento para GPU.

```
1  Image<float> in = Tools::load_image<float>("images/gray.png");
2  Func input = BoundaryConditions::repeat_edge(in);
3  Func blur_x, blur_y;
4  Var x, y, xi, yi;
5
6  // Algoritmo / pipeline de processamento
7  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y)) / 3;
8  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1)) / 3;
9
10 // Escalonamento de execução para GPU
11 blur_y.tile(x, y, xi, yi, 32, 8).gpu_blocks(x, y).gpu_threads(xi, yi);
12 blur_x.compute_at(blur_y, x).gpu_threads(x, y);
13
14 Target target = get_host_target();
15 target.set_feature(Target::CUDA);
16
17 Image output = blur_y.realize(in.width(), in.height(), target);
```

Para ilustrar a diferença entre um código Halide puro e um código C++ equivalente, escrito e otimizado manualmente, é apresentada a figura 2.4 a seguir. No código C++ do exemplo são utilizados diretivas de compilação para definir o paralelismo e funções intrínsecas para vetorização. Além disso, a organização das iterações (laços de processamento) e a lógica do

**Halide** 0.9 ms/megapixel

```
Func box_filter_3x3(Func in) {
  Func blurx, blury;
  Var x, y, xi, yi;

  // The algorithm - no storage, order
  blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
  blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blury.tile(x, y, xi, yi, 256, 32)
       .vectorize(xi, 8).parallel(y);
  blurx.compute_at(blury, x).store_at(blury, x).vectorize(x, 8);

  return blury;
}
```

**C++** 0.9 ms/megapixel

```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
         a = _mm_loadu_si128((__m128i*)(inPtr-1));
         b = _mm_loadu_si128((__m128i*)(inPtr+1));
         c = _mm_load_si128((__m128i*)(inPtr));
         sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
         avg = _mm_mulhi_epi16(sum, one_third);
         _mm_store_si128(blurxPtr++, avg);
         inPtr += 8;
      }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

Figura 2.4: Comparação entre Códigos Halide e C++ Equivalentes (Durand, 2015b).

algoritmo estão acoplados. Conforme apresentado na figura os códigos Halide e C++ tem o mesmo desempenho.

No exemplo da figura 2.4 a entrada `in` corresponde a uma imagem em nível de cinza de 16 bits. O processamento usa paralelismo em janelas de 256x32 (colunas x linhas) com vetorização (instruções SIMD) de 8 posições, ou seja, 128 bits (8x16). Como `blury` depende de `blurx`, um *buffer* auxiliar do tamanho da janela é alocado para armazenar o resultado intermediário. No caso do C++ é possível perceber que a alocação do *buffer* auxiliar utiliza um tipo de dados de 128 bits usado por instruções intrínsecas, capaz de armazenar 8 posições, logo no cálculo do tamanho do *buffer* a quantidade de 256 colunas da janela foi dividida por 8. Por outro lado, na quantidade de linhas foi adicionado 2, pois corresponde a área de sobreposição de `blurx` utilizada por `blury`.

## 2.2 COMPILAÇÃO E GERAÇÃO DE CÓDIGO

As linhas de código Halide (na prática código C++) que observamos nos exemplos anteriores, como no código 2.2, exceto pela instrução `realize`, não fazem o processamento da imagem propriamente. De fato estão apenas montando uma representação em memória da estrutura do *pipeline* de processamento de imagem. É uma meta-programação, ou seja, são apenas instâncias de estruturas tipo `Func`, `Var`, `Expr`, etc organizadas em memória. Concretamente o Halide modela um grafo acíclico dirigido - DAG (*Directed Acyclic Graph*) das funções Halide definidas no *pipeline*, com a raiz do grafo iniciando pela função de saída do *pipeline*. A efetiva computação só acontece na chamada do método `realize`, o qual realiza todo o processo de compilação e geração de código de máquina, bem como a execução propriamente do *pipeline* (Ragan-Kelley, 2014).

Halide é implementada como uma biblioteca C++, a qual também inclui um compilador completo LLVM (*Low Level Virtual Machine Compiler Infrastructure*) utilizado para gerar código de máquina em tempo de execução (Durand, 2015c). O LLVM, proposto por Lattner e Adve (2004), é uma infraestrutura de compilador desenvolvida para programas escritos em linguagens de programação variadas. A infraestrutura provê camadas intermediárias de um compilador, suportando um conjunto de instruções e um sistema de tipos independentes de linguagem, chamado de representação intermediária do LLVM.

Na arquitetura do LLVM, um *frontend* de uma linguagem de alto nível é responsável por analisar e diagnosticar erros no código de entrada e, em seguida, traduzir o código analisado para a representação interna do LLVM. Esta representação pode opcionalmente passar por uma série de análises e otimizações de código, sendo então enviada para um gerador de código *backend* que irá produzir código de máquina nativo (Lattner, 2018). Os *backends* e *frontends* são independentes, assim, um mesmo *backend* específico para uma arquitetura de hardware pode ser utilizado por *frontends* de diferentes linguagens. Na prática, o Halide usufrui dessa infraestrutura, necessitando apenas converter a representação em memória da estrutura do *pipeline* para a representação interna do LLVM, usufruindo então de *backends* já construídos para o LLVM.

Conforme ilustrado no diagrama da figura 2.5, utilizando como entrada as declarações de funções e escalonamento Halide, o processo de compilação realiza sintetizações, transformações e otimizações, mapeando depois para a representação interna do LLVM. A partir da representação em LLVM é gerado o código para a plataforma desejada utilizando *backends* já existentes.

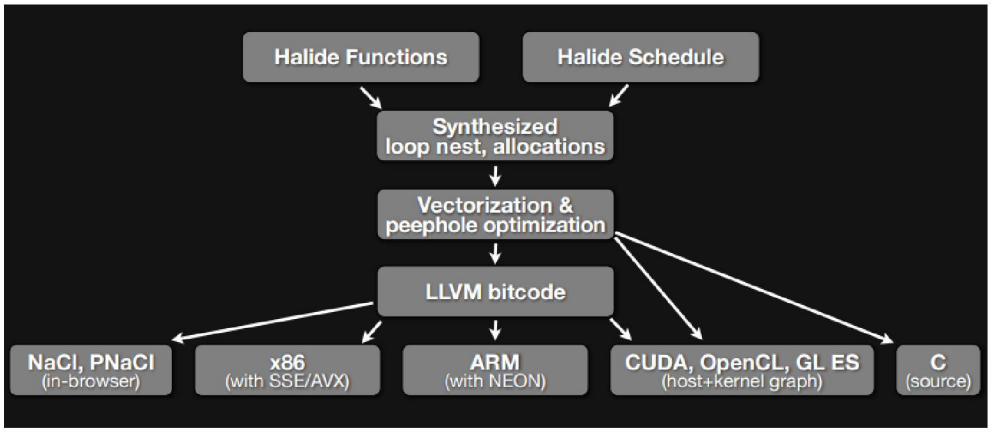



Figura 2.5: Processo de Compilação Halide (Ragan-Kelley et al., 2015).

Ragan-Kelley et al. (2013) explicam que para alguns padrões de vetorização, por ser difícil de representar no LLVM, o Halide faz otimizações e utiliza diretamente funções intrínsecas

específicas da plataforma desejada. Processos similares acontecem na etapa de geração de código para GPU, na qual o código resultante caracteriza um amplo grafo de execução híbrido entre CPU e GPU.

Para analisar a estrutura do código a ser gerado via representação LLVM, especialmente o aninhamento dos laços de processamento definido por diferentes tipos de escalonamentos de execução, o Halide disponibiliza uma representação em pseudocódigo do *pipeline* final (Ragan-Kelley, 2014). Até o momento o Halide ainda não disponibiliza uma representação completa do *pipeline* e do escalonamento diretamente em outra linguagem, como por exemplo, C e C++. Um dos mecanismos para visualizar a estrutura de laços gerados é através do método `print_loop_nest` disponibilizado pela classe de funções Halide. Um outro mecanismo consiste em definir a variável de ambiente `HL_DEBUG_CODEGEN` para visualizar outras informações de cada estágio do processo de compilação do Halide. A seguir serão apresentados alguns exemplos com saídas geradas pelo primeiro mecanismo de análise, obtidas a partir de demonstrações disponíveis nos tutoriais online da linguagem (Ragan-Kelley e Adams, 2012).

```
Func gradient("gradient");
gradient(x, y) = x + y;
gradient.realize(4, 4);
```

```
compute gradient:
  for y:
    for x:
      gradient(...) = ...
```

Figura 2.6: Representação dos Laços de Processamento.

No exemplo da figura 2.6 é possível perceber que no escalonamento default o Halide gera um laço de processamento para cada variável de domínio da função, sendo que a ordem dos laços correspondem a ordem em que as variáveis são utilizadas para definir o domínio. Aquela que for utilizada primeiro, corresponde ao laço mais interno, e assim por diante. Em um exemplo bidimensional, como nesse caso, o laço mais interno corresponde às colunas e o externo às linhas.

```
Func gradient("gradient_col_major");
gradient(x, y) = x + y;
gradient.reorder(y, x);
gradient.realize(4, 4);
```

```
compute gradient_col_major:
  for x:
    for y:
      gradient_col_major(...) = ...
```

Figura 2.7: Processamento Reordenado por Coluna.

Em outro exemplo, na figura 2.7, é apresentado a diretiva de escalonamento `reorder`, que permite alterar a ordem dos laços de processamento associado às variáveis de domínio, ou seja, ao invés de processar o domínio horizontalmente, processa verticalmente. Vale lembrar que para todas as diretivas de escalonamento disponíveis, o Halide evita que o algoritmo (lógica) seja afetado, ou seja, o programa sempre retornará os mesmos resultados, independente do escalonamento utilizado.

```
Func gradient("gradient_split");
gradient(x, y) = x + y;
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 2);
gradient.realize(4, 4);
compute gradient_split:
  for y:
    for x.x_outer:
      for x.x_inner in [0, 1]:
        gradient_split(...) = ...
```

Figura 2.8: Processamento com *Split*.

Uma diretiva de escalonamento bastante interessante, apresentada na figura 2.8, é a `split`, que divide um laço de processamento em dois, usando um fator informado. Caso esse fator de divisão não seja múltiplo do tamanho do domínio, o Halide garante que não haverá acesso além dos limites ou alterações no resultado, pois gera código com tratamentos de controle (Ragan-Kelley, 2014). Essa diretiva sozinha não tem grande impacto, porém quando combinada com outras, como será visto mais adiante, pode impactar bastante a estrutura gerada. De maneira análoga, também existe a diretiva `fuse`, que combina dois laços de processamento em um único.

```
Func gradient("gradient_in_vectors");
gradient(x, y) = x + y;
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.vectorize(x_inner);
// gradient.vectorize(x, 4);
compute gradient_in_vectors:
  for y:
    for x.x_outer:
      vectorized x.x_inner in [0, 3]:
        gradient_in_vectors(...) = ...
```

Figura 2.9: Processamento com Vetorização.

A diretiva `vectorize` usada para gerar vetorização é um exemplo normalmente combinado com `split`. Usando essa diretiva, várias posições são processadas simultaneamente, substituindo assim o que seria o laço mais interno, conforme mostra o pseudocódigo da figura 2.9. Por ser uma operação comum, também existe uma versão da diretiva de vetorização que recebe um fator de divisão e assim combina automaticamente as duas diretivas, `split` e `vectorize`, em uma única, conforme pode ser observado na linha de código comentada na mesma figura 2.9.

Outra diretiva usada para criar janelas de processamento, já comentada anteriormente, é o `tile`, que deriva da combinação entre `split` e `reorder`, conforme ilustrado no exemplo da figura 2.10.

Para definir paralelismo é usada a diretiva de escalonamento `parallel`, que também pode ser combinada com outras, conforme apresentado na figura 2.11. Neste exemplo, cada janela *tile* definida poderá ser executada em uma ordem arbitrária, devido ao paralelismo definido

```
Func gradient("gradient_tiled");
gradient(x, y) = x + y;
Var x_outer, x_inner, y_outer, y_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.split(y, y_outer, y_inner, 4);
gradient.reorder(x_inner, y_inner, x_outer, y_outer);
// gradient.tile(x, y, x_outer, y_outer, x_inner, y_inner, 4, 4);
compute gradient_tiled:
  for y.y_outer:
    for x.x_outer:
      for y.y_inner in [0, 3]:
        for x.x_inner in [0, 3]:
          gradient_tiled(...) = ...
```

Figura 2.10: Processamento com *Tile*.

no laço mais externo, que por sua vez derivou da fusão de dois outros laços. Vale lembrar que o Halide evita a existência de condições de concorrência (*race condition*), logo a ordem arbitrária da execução não afeta o resultado.

```
Func gradient("gradient_fused_tiles");
gradient(x, y) = x + y;
Var x_outer, y_outer, x_inner, y_inner, tile_index;
gradient.tile(x, y, x_outer, y_outer, x_inner, y_inner, 4, 4);
gradient.fuse(x_outer, y_outer, tile_index);
gradient.parallel(tile_index);
compute gradient_fused_tiles:
  parallel x.x_outer.tile_index:
    for y.y_inner in [0, 3]:
      for x.x_inner in [0, 3]:
        gradient_fused_tiles(...) = ...
```

Figura 2.11: Processamento Paralelo.

Normalmente os *pipelines* possuem múltiplos estágios, ou seja, são construídos usando múltiplas funções Halide. Nesses casos as dependências entre os estágios são mapeados usando um conceito de produtor-consumidor. As diretivas de escalonamento apresentadas anteriormente também podem ser aplicadas nestes cenários, individualmente para cada função, porém existem algumas outras diretivas específicas para organizar a relação de dependência, como por exemplo, a diretiva `compute_at`, que permite indicar onde dentro dos laços de processamento uma função deve ser processada. O Halide usa essa informação para decidir quais posições do domínio do "produtor" precisam ser processadas antes de liberar o resultado para o "consumidor".

Por exemplo, quando um estágio produtor processou uma janela, o estágio seguinte consumidor já pode iniciar seu processamento. Esse é o caso do exemplo apresentado na figura 2.12. De maneira análoga, também existe a diretiva `store_at`, que permite indicar onde dentro dos laços de processamento o resultado de um estágio produtor, ou de uma janela já processada do produtor, deve ser armazenado. Essas informações são utilizadas pelo Halide para determinar maneiras otimizadas de alocar *buffer* auxiliares e então armazenar e reutilizar

```
Func producer("producer_tile"), consumer("consumer_tile");
producer(x, y) = sin(x * y);
consumer(x, y) = (producer(x, y) +
                  producer(x, y+1) +
                  producer(x+1, y) +
                  producer(x+1, y+1))/4;

Var x_outer, y_outer, x_inner, y_inner;
consumer.tile(x, y, x_outer, y_outer, x_inner, y_inner, 4, 4);

producer.compute_at(consumer, x_outer);
```

```
compute consumer_tile:
  for y.y_outer:
    for x.x_outer:
      compute producer_tile:
        for y:
          for x:
            producer_tile(...) = ...
      for y.y_inner in [0, 3]:
        for x.x_inner in [0, 3]:
          consumer_tile(...) = ...
```

Figura 2.12: Organização do Processamento Produtor-Consumidor.

resultados intermediários processados ao londo dos múltiplos estágios do *pipeline*, sempre respeitando eventuais dependências de dados existentes entre os estágios.

Também é possível utilizar a diretiva `compute_root`, conforme pode ser observado no pseudocódigo do exemplo na figura 2.13, em situações de dependência na qual é desejado que um determinado estágio produtor do *pipeline* seja inteiramente processado antes de seguir para o estágio consumidor seguinte.

```
Func producer("producer_root"), consumer("consumer_root");
producer(x, y) = sin(x * y);
consumer(x, y) = (producer(x, y) +
                  producer(x, y+1) +
                  producer(x+1, y) +
                  producer(x+1, y+1))/4;

producer.compute_root();
```

```
compute producer_root:
  for y:
    for x:
      producer_root(...) = ...
compute consumer_root:
  for y:
    for x:
      consumer_root(...) = ...
```

Figura 2.13: Processamento Completo do Produtor Antes do Consumidor.

Além das diretivas apresentadas anteriormente, ainda existem outras diretivas disponibilizadas pela linguagem, inclusive para aceleradores gráficos. Essas outras diretivas também podem ser utilizadas para explorar opções de escalonamento de execução, buscando encontrar uma organização que resulte em um desempenho melhor.

## 2.3 CONCLUSÃO

Neste capítulo apresentamos uma introdução à linguagem de domínio específico chamada Halide, a qual é dedicada ao domínio de processamento de imagens. Sua principal característica é a separação entre a definição lógica do algoritmo e sua organização para execução. Através das chamadas diretivas de escalonamento presentes na linguagem, é possível instruir o compilador para gerar código de máquina com diferentes organizações e que usufruem de características específicas de determinadas arquiteturas, sem afetar a definição e resultado do programa. Esta característica da linguagem será explorada neste trabalho em conjunto com técnicas de aprendizado de máquina para tentar encontrar escalonamentos otimizados para uma determinada arquitetura de maneira automatizada.

# 3 APRENDIZADO POR REFORÇO

O Aprendizado por Reforço é uma área de Aprendizagem de Máquina que consiste na representação de um agente interagindo com um ambiente, por meio de ações, e recebendo uma recompensa como retorno de cada ação tomada. A partir de suas interações com o ambiente o agente aprende a escolher ações futuras que maximizem o retorno recebido (Ottoni et al., 2015).

Para cada interação, o agente observa o estado atual do ambiente e então escolhe uma ação. Esta ação altera o ambiente levando-o para um novo estado. O ambiente então retorna um sinal de reforço ou recompensa como medida dessa mudança de estado (Júnior, 2012). Esse processo é ilustrado na figura 3.1.


Figura 3.1: Modelo de Aprendizado por Reforço (Júnior, 2012).

O agente mapeia os estados do ambiente para ações que deve tomar a partir de uma função que é definida como a política do agente, a qual pode ser guiada por diferentes heurísticas. A tarefa principal do agente é encontrar uma sequência de ações que determine uma política otimizada, ou seja, que maximize a soma total dos retornos recebidos (Silva, 2016). Entretanto, ao contrário dos métodos de aprendizado supervisionado que existem pares de "entrada/saída" para serem utilizados no treinamento, o agente precisa obter experiência (conhecimento) a partir dos possíveis estados, ações e recompensas (Júnior, 2012).

O modelo de aprendizado por reforço apresentado na figura 3.1 é formalizado por meio de um Processo de Decisão de Markov (MDP - *Markov Decision Process*).

## 3.1 PROCESSO DE DECISÃO DE MARKOV

Um Processo de Decisão de Markov (MDP) fornece uma estrutura matemática para modelagem de tomada de decisão em situações nas quais os resultados são em parte aleatórios e em parte sob o controle de um tomador de decisão. Segundo Sutton e Barto (1998), de maneira geral, a solução de problemas de decisão de Markov pode ser obtida a partir da utilização do aprendizado por reforço.

O MDP, conforme ilustrado na figura 3.2, é um processo de controle estocástico de tempo discreto. Em cada passo de tempo, o processo está em algum estado e o agente pode escolher qualquer ação que esteja disponível. O processo responde no próximo passo de tempo, movendo-se aleatoriamente para um novo estado e dando ao agente uma recompensa correspondente. A probabilidade do processo se mover para um determinado novo estado é influenciada pela ação escolhida, mas é independente de todos os estados e ações anteriores.

Processos com estas características são ditos satisfazerem a Propriedade de Markov (Lopes e Braga, 2017).

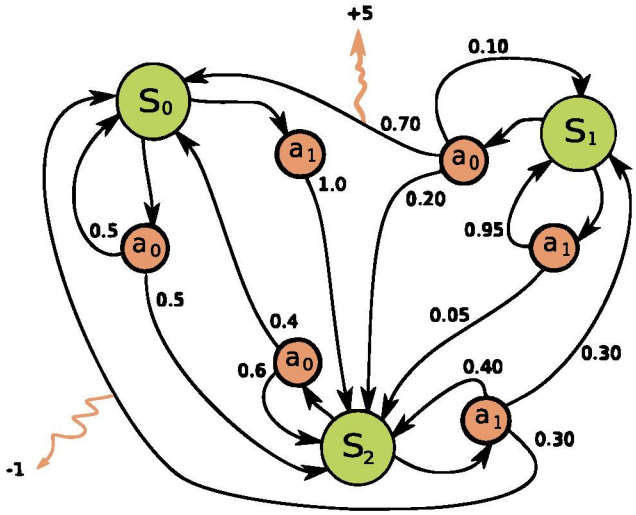



Figura 3.2: Ilustração de um processo de decisão de Markov com três estados e duas ações (Wikipedia, 2018). As setas contínuas indicam a probabilidade de possíveis transições entre estados a partir de ações. As setas curvadas indicam possíveis recompensas pelas transições realizadas.

A propriedade de Markov permite soluções incrementais, como na programação dinâmica, na qual os valores de retorno em um estado seguinte podem ser expressos a partir dos valores correntes, de maneira recursiva, sem a necessidade de todo o histórico anterior (Costa, 2017).

## 3.2 RECOMPENSA FUTURA COM DESCONTO

Matiisen (2015) explica que dado um agente interagindo com um ambiente que satisfaz o MDP, através de uma sequência de ações até alcançar um estado terminal, recebendo uma recompensa imediata $r$ para cada ação executada, o total de recompensas obtidas do ambiente pode ser expresso como na equação 3.1. Essa sequência de ações até alcançar um estado terminal, da primeira até a n-ésima, é chamada de episódio, e o total de recompensas obtidas nessa sequência é chamado de recompensa do episódio.

$$R = r_1 + r_2 + r_3 + ... + r_n \tag{3.1}$$

Para expressar apenas o total de recompensas futuras, ou seja, o total obtido com as ações a partir de um determinado passo de tempo $t$ até alcançar um estado terminal, pode-se utilizar da equação 3.2.

$$R_t = r_t + r_{t+1} + r_{t+2} + ... + r_n \tag{3.2}$$

Uma vez que os ambientes são estocásticos, não é possível ter certeza de quais serão as recompensas obtidas nos passos futuros do processo. Quanto mais no futuro for uma estimativa,

maior pode ser a divergência. Por esse motivo é comum utilizar Recompensas Futuras com Desconto (*Discounted Future Reward*), como representado na equação 3.3.

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{n-1} r_n = \boxed{r_t + \gamma R_{t+1}} \tag{3.3}$$

O $\gamma$ da equação 3.3 é o fator de desconto, entre `0` e `1`. Quanto mais no futuro uma recompensa, menos seu valor será considerado. Por exemplo, uma boa estratégia para um agente interagindo com o ambiente seria sempre escolher ações que maximizem o total de recompensas futuras com desconto.

## 3.3 DILEMA ENTRE EXPLORAR E USUFRUIR

O dilema entre explorar e usufruir (*exploration-exploitation*) se caracteriza pelo fato do agente de aprendizado por reforço precisar usufruir as melhores ações para maximizar as recompensas, mas também precisar explorar o ambiente para encontrar as melhores ações (Li, 2017). Para um agente aprender como lidar de maneira otimizada com todos os estados de um ambiente, é necessário que ele seja exposto ao máximo de estados possíveis. Idealmente uma abordagem deve incentivar a exploração do ambiente até o agente aprender o suficiente para escolher as ações mais adequadas.

Para esse cenário algumas das abordagens possíveis são (Juliani, 2016b):

- **Gulosa (*Greedy*):** Para garantir que a melhor ação estimada seja escolhida em qualquer passo de tempo na interação com o ambiente, basta que o agente escolha a ação com maior estimativa de recompensa (argmax). Essa abordagem provê pouca ou nenhuma exploração do ambiente, consequentemente quase sempre converge para um mínimo local.

- **Randômica:** Trata-se da abordagem oposta à gulosa, pois o critério é sempre escolher uma ação aleatória. É normalmente útil como um ponto de partida para explorar o ambiente e acumular experiência com as transições.

- **$\epsilon$-gulosa ($\epsilon$-*greedy*):** Essa abordagem é um combinação das abordagens gulosa e randômica, sendo uma das estratégias de exploração mais usada. Nessa estratégia o agente escolhe a melhor ação, de maior valor estimado, na maiorias das vezes, porém ocasionalmente escolhe uma ação aleatória. O parâmetro $\epsilon$ determina a probabilidade de escolher uma ação randomicamente. No início do processo de aprendizagem o parâmetro é inicializado com uma alta probabilidade e reduzido progressivamente para um valor constante pequeno.

  Dentro dos agentes de aprendizado por reforço essas abordagens são chamadas de políticas de escolha de ações, as quais definem as regras de como um agente escolhe uma ação para cada estado do ambiente. Todo algoritmo de aprendizado por reforço define de alguma forma uma política de escolha de ações.

## 3.4 *Q-LEARNING*

O *Q-Learning* é um algoritmo de aprendizado por reforço que foi proposto por Watkins (1989), no qual é definido uma função de valor estado-ação $Q(s,a)$ que representa a estimativa do valor ótimo, ou seja, a maior recompensa futura com desconto a partir do estado $s$ executando

uma ação $a$, e mantendo-se a maior a partir desse ponto. Essa função é representada pela equação 3.4.

$$Q(s_t, a_t) = max(R_t) \tag{3.4}$$

A equação 3.4 pode ser entendida como a "maior pontuação" (recompensa) possível até atingir o estado final a partir do estado $s_t$ executando a ação $a_t$, representando assim uma medida da "qualidade" de uma determinada ação em um determinado estado.

Para estimar o total de recompensas até um estado final a partir do estado atual, sem conhecer quais serão os estados, ações e recompensas futuras, usa-se a função de valor estado-ação $Q(s, a)$ e escolhe apenas as ações com maior valor a partir de um determinado estado, conforme representado na equação 3.5.

$$\pi(s) = argmax_a Q(s, a) \tag{3.5}$$

Na equação 3.5 o $\pi$ representa a política, a regra de como escolher uma ação em cada estado. Um aspecto relevante é que a escolha das ações durante o processo de aprendizado da função $Q(s, a)$ pode ser realizada por qualquer método de exploração, até mesmo de forma aleatória (Silva, 2016).

Agora, considerando apenas uma transição $< s, a, r, s' >$, em que o agente executa uma ação $a$, em um estado $s$, recebendo uma recompensa $r$ e indo para um novo estado $s'$, de maneira similar a equação 3.3 usada no cálculo da recompensa futura com desconto, é possível representar o valor de $Q$ do estado atual $s$ em termos do valor de $Q$ do novo estado $s'$, através da equação 3.6.

$$Q(s, a) = r + \gamma max_{a'} Q(s', a') \tag{3.6}$$

Esta equação é chamada de equação de Bellman (Bellman, 1957). A recompensa máxima futura para este estado e ação é a recompensa imediata da ação mais a recompensa máxima futura para o estado seguinte.

A principal ideia no *Q-Learning* é a possibilidade de iterativamente aproximar o valor da função $Q$ usando a equação de Bellman. A implementação da função $Q$, por exemplo, pode ser realizada por meio de uma tabela na qual as linhas são os estados e as colunas as ações.

O algoritmo *Q-Learning* é apresentado a seguir:

**Algoritmo 1** *Q-Learning*

1: inicialize $Q$[num_estados, num_ações] randomicamente
2: observe o estado inicial $s$
3: **repeat**
4: escolha e execute uma ação $a$
5: observe a recompensa $r$ e o novo estado $s'$
6: $Q[s, a] = Q[s, a] + \alpha(r + \gamma max_{a'} Q[s', a'] - Q[s, a])$
7: $s = s'$
8: **until** parar treinamento

No algoritmo 1 o parâmetro $\alpha$ é a taxa de aprendizagem que controla o quanto da diferença entre o valor atual e o novo valor proposto de $Q$ será levada em consideração. Note que o novo valor proposto corresponde a equação de Bellman ilustrada em 3.6. O valor $r$ é o retorno imediato obtido com a última ação, porém, $max_{a'} Q[s', a']$ é apenas uma aproximação dos valores dos estados seguintes, que inicialmente pode estar completamente errada, mas a

aproximação melhora a cada iteração, e foi demonstrado que se executado um número suficiente de vezes os valores convergem para o valor correto de $Q$.

Ainda que o critério de convergência para valores ótimos do *Q-Learning* demande que todos os pares de estado-ação sejam visitados repetidamente inúmeras vezes (Watkins e Dayan, 1992) e o conjunto de estados e ações sejam discretos, na prática, ao executar um número de iterações suficientemente grande (considerando a tarefa a ser aprendida) é possível alcançar valores bastante relevantes (Silva, 2016).

## 3.5 *Q-LEARNING* COM REDES NEURAIS

A estrutura de tabela usada no *Q-Learning* é interessante, mas para problemas mais complexos que possuem um número grande de estados e/ou ações, pode gerar um problema de escalabilidade, pois o tamanho da tabela pode se tornar inviavelmente grande. Para muitos problemas é necessário, de alguma maneira, obter uma representação dos estados e derivar o valor de $Q$ para as ações sem utilizar uma tabela. É nesse contexto que as redes neurais são adicionadas, atuando como uma função de aproximação do valor de $Q$. Matiisen (2015) explica que a rede neural recebe um vetor de representação dos estados possíveis e aprende a gerar saídas com os valores de $Q$ de cada ação possível, de acordo com o modelo ilustrado na figura 3.3.

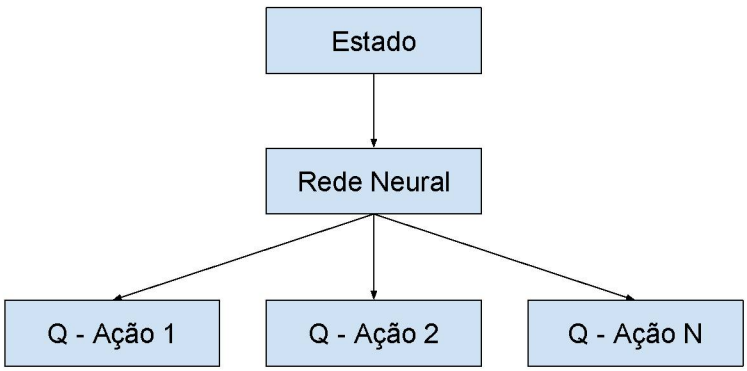



Figura 3.3: Modelo da Rede Neural para *Q-Learning*. Adaptado de Matiisen (2015).

O método de atualização dos valores de $Q$ também é diferente. Ao invés de diretamente atualizar uma tabela, com a rede neural será usado uma função de perda/erro (*loss function*) com *backpropagation*[1] para atualização dos pesos da rede, visando minimizar a perda (Juliani, 2016a). A função de perda, apresentada na equação 3.7 abaixo, utiliza a soma dos quadrados da diferença entre o valor corrente previsto de cada $Q$ e o novo valor de aproximação $Q\text{-}target$ obtido de acordo com a equação 3.6 acima.

$$Perda = \sum (Q\text{-}target - Q)^2 \tag{3.7}$$

No algoritmo *Q-Learning* apresentado anteriormente, a regra de atualização usando tabela precisa ser substituída para trabalhar com a rede neural. Considerando uma transição $< s, a, r, s' >$, a regra de atualização deve seguir os seguintes passos (Matiisen, 2015):

1. Passar pela rede neural o estado atual $s$ para obter os valores $Q$ previsto para cada ação.
2. Passar pela rede neural o novo estado $s'$ e escolher a ação de maior valor $[max_{a'} Q(s', a')]$.

[1]Método usado para calcular a contribuição de cada neurônio da rede no erro obtido.

3. Calcular a perda da ação $a$ atribuindo o valor de $Q$ de acordo com o passo 1 e $Q\text{-}target$ de acordo com a equação 3.6, usando o valor do passo 2 e recompensa $r$. Para as demais ações, atribuir $Q\text{-}target$ para os mesmos valores do passo 1, zerando assim o erro/perda.

4. Atualizar os pesos da rede via *backpropagation* (*gradient descent*).

## 3.6 *DEEP Q-NETWORK*

É conhecido que o aprendizado por reforço é instável ou até diverge quando uma função de aproximação não linear é usada, como no caso de quando uma rede neural é usada para representar os valores das ações (Mnih et al., 2015). Para tratar essa instabilidade, um novo modelo de agente, chamado DQN (*Deep Q-Network*), foi proposto por Mnih et al. (2013) e Mnih et al. (2015), apresentando uma nova variação do *Q-Learning* com Rede Neural, que acrescenta algumas técnicas para melhorar a estabilidade do algoritmo.

O DQN supera a instabilidade na aprendizagem principalmente pelas seguintes técnicas (Mnih et al., 2015; Seno, 2017):

- **Reuso de Experiência (*Experience Replay*):** Trata-se de uma memória de transições (experiências) anteriores a serem usadas de maneira aleatória durante o treinamento. No aprendizado por reforço, uma rede neural fica facilmente especializada nas iterações mais recentes (*overfitting*). Para resolver esse problema o DQN armazena as transições realizadas $< s, a, r, s' >$ e treina a rede utilizando pequenos lotes selecionados randomicamente ao invés das transições mais recentes. Alguns dos benefícios são:

  - Reduz a correlação entre as transições subsequentes, pois a similaridade poderia induzir a rede para um mínimo local.
  - Aumenta a velocidade da aprendizagem devido ao uso de lotes de transições.
  - Evita o esquecimento catastrófico (*catastrophic forgetting*) reutilizando transições anteriores.

- **Rede de Aproximação Separada (*Separate Target Network*):** Utiliza uma rede separada, clone da rede principal, para obter o valor de aproximação $Q\text{-}target$ usado no cálculo da função de perda/erro. O clone da rede é atualizado periodicamente a partir de uma determinada quantidade de transições. Quando o $Q\text{-}target$ varia constantemente, torna o treinamento difícil. Essa técnica melhora a estabilidade da rede e generalização, reduzindo a correlação de dados entre o valor previsto de $Q$ e o novo valor de aproximação $Q\text{-}target$.

- **Recompensa Limitada (*Reward Clipping*):** O valor de recompensa gerado no ambiente é ajustado deixando $+1$ para recompensas positivas, $-1$ para negativas, e mantendo 0 inalterado. Com essa limitação na amplitude do valor, limita a escala de erro e facilita o uso das mesmas taxas de aprendizagem entre diferentes ambientes.

  O algoritmo *Q-Learning* adaptado proposto para o DQN é apresentado a seguir:

**Algoritmo 2** *Q-Learning* para DQN

inicialize a memória de reuso $D$ com capacidade $N$
inicialize a rede principal $Q$ com pesos randomicamente
inicialize a rede clone $\hat{Q} = Q$
observe o estado inicial $s$
**repeat**
    escolha uma ação $a$:  // política $\epsilon$-gulosa usando rede principal
        caso gere uma probabilidade aleatória inferior a $\epsilon$ escolha $a$ randomicamente
        caso contrário escolha $a = argmax_a Q(s,a)$

    execute a ação escolhida $a$
    observe a recompensa $r$ e o novo estado $s'$
    armazene a transição $< s,a,r,s' >$ na memória $D$

    escolha transições aleatórias $< s_1,a_1,r_1,s_1' >$ da memória $D$
    obtenha $q$:  // usando rede principal
        $q = Q(s_1,a_1)$
    obtenha $\hat{q}\text{-}target$:  // usando rede clone
        caso $s_1'$ seja terminal $\hat{q}\text{-}target = r_1$
        caso contrário $\hat{q}\text{-}target = r_1 + \gamma max_{a'} \hat{Q}(s_1',a')$
    treine a rede principal $Q$ (*backpropagation*) usando perda $(\hat{q}\text{-}target - q)^2$

    periodicamente a cada $C$ passos faça $\hat{Q} = Q$  // atualiza clone da rede principal
    $s = s'$
**until** parar treinamento

De acordo com o exposto por Mnih et al. (2015), na prática, o algoritmo apenas armazena as últimas $N$ transições (experiências) na memória de reuso $D$, sobrescrevendo com as transições mais recentes. Isso é aplicado devido ao tamanho finito da memória. Para atualizar a rede principal (online) faz amostragens aleatórias uniformes da memória $D$. Outro ponto é que a rede separada de aproximação do novo $Q\text{-}target$ é atualizada com um clone da rede principal a cada $C$ passos de tempo. Ainda, na atualização dos parâmetros da rede principal, cada gradiente de perda/erro foi limitado ao intervalo de valores $[-1, 1]$ para suavizar as alterações na rede quando o erro for grande, melhorando a estabilidade do algoritmo.

## 3.7 *PROXIMAL POLICY OPTIMIZATION*

Apesar do agente DQN ter alcançado grande sucesso em problemas com espaços de alta dimensionalidade de estados, este agente atua com efetividade em espaços de ações discretas e de baixa dimensionalidade, ou seja, que contenha uma baixa quantidade de ações (Li et al., 2017; Lillicrap et al., 2016; Matiisen, 2015; Mnih et al., 2015). Nesse contexto, o *Proximal Policy Optimization* (PPO) proposto por Schulman et al. (2017) é um tipo de agente que visa aprimorar a aplicação do aprendizado por reforço em cenários com ações no espaço contínuo ou com grande quantidade de ações discretas.

O PPO se baseia em uma arquitetura chamada *Actor-Critic* (Atuação-Avaliação), conforme apresentado na figura 3.4. Nessa arquitetura o agente utiliza duas redes neurais, uma para avaliar o estado atual do ambiente (*critic*) e outra para definir a melhor ação para o estado atual (*actor*), sendo que a partir do erro da estimativa da rede de avaliação são derivadas informações necessárias para atualização de ambas as redes. A estrutura das camadas ocultas e

de entrada são iguais nas duas redes, e seu tamanho é configurado por parâmetros do agente, já as camadas de saída são definidas e específicas de acordo com a finalidade de cada rede.

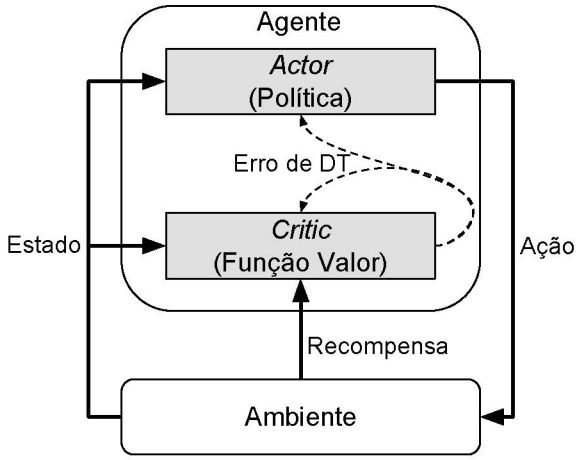



Figura 3.4: Arquitetura *Actor-Critic*. O agente de aprendizado por reforço utiliza duas redes neurais: a rede *critic* que define uma função valor do estado do ambiente, e a rede *actor* que representa a política de escolha de ações a partir do estado do ambiente. A recompensa do ambiente é usada para corrigir o valor estimado pela rede *critic*, cujo erro de diferença temporal (DT) é usado na otimização dos pesos de ambas as redes. Adaptado de Huang (2018).

A rede de avaliação *critic* representa uma função de valor do estado $V(s)$, a qual estima a recompensa futura com desconto a partir de um dado estado $s$ apenas, ou seja, sem a necessidade de considerar uma ação específica, tornando-se assim mais genérica, diferente da função de valor estado-ação $Q(s,a)$, usada no *Q-Learning*, que estima o valor da recompensa futura considerando uma ação específica $a$ aplicada ao estado $s$, e portanto requer de sucessivas iterações de aproximação para todos os valores possíveis de ações, conforme equação 3.6, o que além de dispendioso para grandes quantidades de ações, é inaplicável para ações no espaço contínuo.

Por outro lado, a rede de atuação *actor* representa a política do agente para escolha de ações. Na prática, no PPO esta rede representa o mapeamento do estado $s$ para os parâmetros de uma função gaussiana (espaço contínuo) ou categórica (espaço discreto) de distribuição de probabilidade, da qual será amostrado o valor efetivo da ação usando o valor médio e desvio padrão da distribuição, conforme ilustrado na figura 3.5. Na rede também será aplicado uma penalidade de acordo com um cálculo de entropia para estimular uma melhor exploração do ambiente. Juliani (2016c) explica que quando o nível de certeza da política é baixo resultando em probabilidades similares entre as ações, a entropia será maior, porém a medida que a probabilidade cresce para uma ação específica, a entropia fica menor.

A cada passo de tempo do aprendizado por reforço, o agente PPO utiliza o valor estimado pela rede de avaliação para determinar o erro de Diferença Temporal (DT) e a Vantagem, conceitos estes apresentados a seguir, e então utiliza esses valores para calcular as perdas/erros de aproximação de cada rede. Tais informações são posteriormente utilizadas para atualização dos pesos das redes neurais através do otimizador Adam (*gradient descent*).

O erro de diferença temporal $\delta$ é definido pela equação 3.8 (Silver et al., 2014). Note que a equação é similar ao cálculo de atualização do algoritmo *Q-Learning*, porém não depende da função de valor estado-ação $Q(s,a)$, mas sim da função $V(s)$, estimada pela rede de avaliação e da recompensa imediata $r$ obtida com a última ação. $\gamma$ é o fator de desconto do valor futuro estimado.

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{3.8}$$

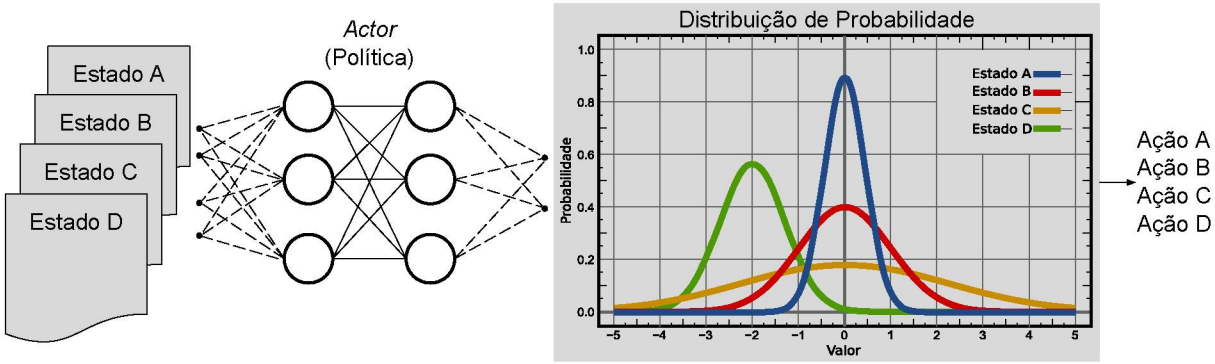



Figura 3.5: Ilustração da política de escolha de ações do PPO. A rede *actor* estima uma função de distribuição de probabilidade para cada estado do ambiente, da qual será amostrado o valor efetivo da ação usando o valor médio e desvio padrão da distribuição. A probabilidade de um determinado valor cresce quando o nível de certeza do agente aumenta. Adaptado de Silva (2018).

Entretanto, o erro de diferença temporal é utilizado como insumo para calcular a vantagem $A(s,a)$, a qual é formalizada pela equação 3.9 (Schulman et al., 2015a). A vantagem pode ser interpretada como um indicador do quanto uma ação é boa ou ruim em comparação ao valor previsto para um estado específico, não ao ganho que a ação ocasionou no ambiente, permitindo ao agente se concentrar onde a previsão da rede está falhando (Juliani, 2016c). Contudo, como não está disponível o valor da função $Q(s,a)$, o PPO utiliza um outro método para estimar a vantagem, chamado *Generalized Advantage Estimation* (GAE), proposto por Schulman et al. (2015b), que aproxima o valor a partir de um conjunto de erros de DT coletados durante as interações com o ambiente.

$$A(s,a) = Q(s,a) - V(s) \tag{3.9}$$

Periodicamente, após uma série de passos coletando dados, o agente inicia um processo de atualização dos pesos das redes, ou seja, da função valor e da política. Na coleta de dados, clones $\theta old$ das redes são utilizados, enquanto que na atualização são usadas as redes online $\theta$, as quais serão novamente clonadas ao final do processo. Similar ao DQN, para calcular a perda na aproximação da função valor é utilizado a equação 3.10 de erro quadrático, na qual $V_{target}$ é calculado adicionando a vantagem ao valor original estimado $V_{target} = V_{\theta old} + A$.

$$Perda = (V_\theta - V_{target})^2 \tag{3.10}$$

Por outro lado, para calcular a perda referente à política, o PPO propôs a equação 3.11 que limita a amplitude da mudança quando a diferença pode levar a atualizações muito grandes na política. Na equação $r_t(\theta)$ denota a razão de probabilidade $\frac{\pi_\theta}{\pi_{\theta old}}$ entre as estimativas da nova política $\pi_\theta$ e da anterior $\pi_{\theta old}$ usada na coleta de dados, tal que $r_t(\theta_{old}) = 1$. $\epsilon$ é um parâmetro da equação que controla os limites de corte (*clip*) da razão de probabilidade usada no segundo termo da equação. Por fim, o valor retornado é o menor entre o primeiro e o segundo termo da equação, ou seja, com ou sem o limiar da razão de probabilidade, caracterizando uma estratégia do tipo *pessimistic bound*.

$$Perda = min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t) \tag{3.11}$$

Vale observar na equação 3.11 que quando a vantagem for negativa o valor adotado para a razão de probabilidade não será menor que $1-\epsilon$ e quando a vantagem for positiva o valor

da razão será limitado a $1 + \epsilon$, removendo assim o incentivo para atualizações na política que desloquem a razão entre a nova política e a anterior para fora do intervalo $[1 - \epsilon, 1 + \epsilon]$.

O algoritmo do agente PPO é apresentado a seguir:

**Algoritmo 3** *PPO*

1: inicialize as redes $\theta_{old} = \theta$
2: **for** iteração = $1, 2, \ldots$ **do**
3:  **for** atuador = $1, 2, \ldots, N$ **do**
4:  execute a política $\pi_{\theta old}$ no ambiente por $T$ passos coletando dados
5:  calcule a vantagem estimada $A_1, \ldots, A_T$
6:  **end for**
7:  otimize os novos pesos $\theta$ por $K$ épocas com lote $M \leq NT$
8:  $\theta_{old} = \theta$
9: **end for**

No algoritmo 3 o agente pode ter, opcionalmente, até $N$ processos atuadores paralelos, executando $T$ passos de tempo em seu ambiente correspondente, quantidade esta conhecida como horizonte, coletando assim segmentos de dados de tamanho fixo e calculando a vantagem obtida em cada passo de tempo. Posteriormente, utilizando o conjunto completo de todos os $NT$ dados coletados, embaralha os dados, calcula a perda e otimiza as novas redes do agente usando lotes de otimização de tamanho $M \leq NT$ até consumir o conjunto de dados completo, então embaralha os dados novamente e repete a otimização até completar $K$ ciclos, chamados de épocas de otimização. As iterações do agente para coleta de dados e otimização terminam após atingir uma condição de parada configurada, que pode ser uma quantidade limite de iterações, episódios, ou passos executados no ambiente.

A figura 3.6 ilustra como os dados coletados durante os passos de tempo são usados em uma iteração de otimização do agente PPO.

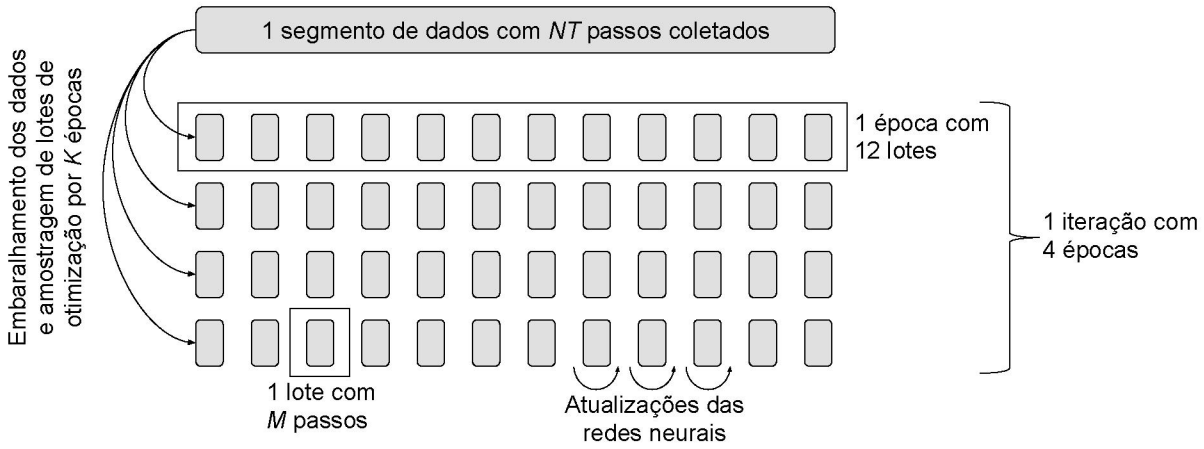



Figura 3.6: Relação entre segmentos de dados, lotes de otimização, épocas e iterações do PPO. As atualizações das redes neurais do agente ocorrem ao final de cada lote processado. Adaptado de Silva (2018).

## 3.8 CONCLUSÃO

Neste capítulo foi apresentado o que é aprendizado por reforço e seus principais conceitos relacionados. Também foi abordado algoritmos associados a essa área da aprendizagem de máquina, ilustrando seus códigos e comentando algumas das principais características, iniciando

por um algoritmo mais tradicional de aprendizado por reforço indo até um algoritmo mais contemporânea. Essas informações ajudaram o leitor a compreender os capítulos seguintes, os quais aplicam esses conceitos e algoritmos ao problema abordado neste trabalho.

# 4 REVISÃO BIBLIOGRÁFICA

Escrever códigos otimizados é uma tarefa desafiadora, e futuras arquiteturas, com a crescente demanda por otimização da estrutura de paralelismo e memória, irão tornar essa tarefa ainda mais difícil. Ragan-Kelley (2014) descreve que essa dificuldade é derivada do grande e complexo espaço de possíveis otimizações e da complexidade do código necessário para representar e experimentar cada ponto nesse espaço.

Segundo Ragan-Kelley (2014), bibliotecas tradicionais, baseadas em composição de subrotinas, não são mais suficientes para programação de alto desempenho, porque o paralelismo e a localidade de memória, pontos chaves para um desempenho eficiente em hardware modernos, são determinados pela estrutura global do programa, não apenas por otimizações pontuais. O trabalho defende que a utilização de linguagens de domínio específico pode ser um sucessor natural para a organização tradicional de bibliotecas, promovendo maior flexibilidade na organização da computação, e que no caso do processamento de imagens, uma representação no modelo funcional, além de mais flexível, também facilita o entendimento e raciocínio.

Van Deursen et al. (2000) comenta que uma DSL pode apresentar características e vantagens relevantes em relação às bibliotecas de subrotinas e tipos de dados. Mullapudi et al. (2015); Hegarty et al. (2014); Ragan-Kelley (2014) exploram algumas dessas características no âmbito do processamento de imagens, e duas podem ser destacadas:

- Transformações globais do programa que necessitam de uma geração extensa de código. Para realizar a otimização do paralelismo e da localidade de memória de um programa de processamento de imagem de maneira global, isto é, do programa como um todo, um sistema eficiente deve analisar a definição completa do *pipeline* (algoritmo), fazer transformações, e depois compilar o resultado para um código eficiente para a arquitetura desejada. Os compiladores são as ferramentas para automatizar essas transformações.

- Controle da implementação da estrutura de dados principal e profundo entendimento das dependências e operações de computação sobre essa estrutura de dados. Esse conhecimento é essencial para viabilizar transformações na estrutura global do programa sem alterar o resultado do processamento. Essas informações chaves são obtidas definindo uma linguagem ciente de uma estrutura particular de dados (domínio específico).

## 4.1 OTIMIZAÇÃO EM HALIDE

Halide, conforme apresentado no capítulo 2, é uma das linguagens estado-da-arte em processamento de imagens, usada em produtos de grande abrangência como o Google Photos e em filtros do aplicativo Câmera para Android Nexus 5 (Denniston, 2016; Ragan-Kelley, 2014). Uma das características mais distintas da linguagem Halide é o desacoplamento simultâneo entre a definição do *pipeline* e a sua organização para computação, com a parte da organização controlada pelo programador através de diretivas explícitas de escalonamento.

Segundo Ragan-Kelley (2014), ainda que na prática programas Halide normalmente foquem em *pipelines* de até cem estágios, não sendo limitado a isso, a análise do escalonamento de cada estágio individualmente é simples, mas a complexidade emerge da composição para globalmente otimizar um *pipeline* inteiro, devido a conflitos de escolha (*trade-off*) entre diferentes opções de escalonamento, como a ordem de alocação, execução e comunicação de cada estágio.

Desenvolver escalonamentos otimizados para arquiteturas modernas requer profissionais que também sejam experientes em técnicas de otimização e nas arquiteturas de hardware. Por exemplo, segundo Mullapudi et al. (2016), na Google, cerca de setenta engenheiros de software escreviam programas de processamento de imagens usando Halide, mas eles dependiam de um número bem menor de engenheiros, especializados no desenvolvimento do escalonamento, para produzir implementações mais eficientes. Esse exemplo evidencia que a elaboração do escalonamento é uma tarefa que exige conhecimentos mais específicos, que não são comuns a todos os desenvolvedores, e que vai além do entendimento da linguagem e do algoritmo de processamento de imagem propriamente.

### 4.1.1 Otimização usando Algoritmo Genético

Ragan-Kelley et al. (2013) aplicou uma técnica de busca estocástica para automaticamente localizar bons escalonamentos para *pipelines* implementados em Halide. O otimizador automático recebe um *pipeline* em Halide e tenta otimizar o tempo de execução buscando pelo escalonamento mais eficiente. O espaço de busca é enorme, tornando inviável uma busca exaustiva por todas as possibilidades. Por exemplo, o *pipeline* para o algoritmo do filtro Laplacian Local contém aproximadamente 100 estágios, sendo estimado um limite inferior de $10^{720}$ escalonamentos possíveis (Ragan-Kelley, 2014; Ragan-Kelley et al., 2013). Esse valor foi derivado rotulando cada estágio com três *tiles* e todas as possibilidades de armazenamento e granularidade de execução permitidas. Um escalonamento otimizado depende da arquitetura da máquina, dimensões da imagem e do código produzido.

O otimizador automático (*autotuner*) proposto no trabalho de Ragan-Kelley et al. (2013) usa algoritmo genético para procurar uma solução aproximada. No algoritmo genético foi usado uma população fixa grande de 128 indivíduos por geração para manter diversidade, construindo novas gerações usando regras de elitismo, cruzamento, mutação e indivíduos randômicos, sendo que nas regras de mutação foi incorporado conhecimento sobre o escalonamento. A heurística incorporada nas regras de mutação foi essencial para convergir quando otimizando *pipelines* mais complexos, entretanto, essas regras são específicas à estrutura do programa sendo otimizado. Além disso, ocasionalmente o *autotuner* ficava preso em mínimos locais, necessitando reinicializações com novos valores randômicos, escolhendo então a melhor entre várias execuções.

O otimizador foi capaz de encontrar boas implementações que produziam desempenho equivalente ao estado-da-arte, levando horas a dias para convergir para bons escalonamentos (Mullapudi et al., 2015). Essa abordagem por algoritmo genético para otimização do escalonamento não é mais mantida e não está mais disponível no Halide. Também foi relatado que o escalonamento gerado, de maneira automática, é pouco afetado por mudanças moderadas na resolução da imagem ou na arquitetura de hardware, porém, grandes mudanças causam alterações significativas no melhor escalonamento. Além disso, experimentos realizados mostraram que o processo apresentava melhor generalização partindo de imagens de baixas resoluções para altas resoluções (Ragan-Kelley, 2014; Ragan-Kelley et al., 2013).

### 4.1.2 Otimização usando Conjunto de Técnicas de Busca

O OpenTuner (Ansel et al., 2014b) é um framework para construir otimizadores multiobjetivo de domínio específico. Ao invés de otimizar um programa diretamente, o desenvolvedor expressa um espaço de busca das implementações e otimizações possíveis. O framework suporta representações customizáveis para domínios específicos e um mecanismo para interagir com o programa a ser otimizado. Uma característica chave do OpenTuner é o

uso simultâneo de um conjunto de técnicas de busca, no qual as técnicas que produzem melhor resultado são dinamicamente alocadas para atuar em uma proporção maior dos testes, enquanto técnicas que não geram bons resultados recebem proporções menores dos testes até serem desativadas (Ansel et al., 2014a). As técnicas compartilham resultados usando uma base comum, assim avanços alcançados por uma técnica podem beneficiar outras.

O framework inclui implementações de diversas técnicas de busca, entre elas, evolução diferencial; várias variantes do método de busca Nelder-Mead (Nelder e Mead, 1965) e do método subida de encosta Torczon (Torczon, 1989); técnicas de mutação evolucionárias; método de busca de padrão; otimização por enxame de partículas; e busca randômica. Segundo Ansel et al. (2014a), essas técnicas abrangem uma série de estratégias e cada uma tende a desempenhar melhor em diferentes tipos de espaços de busca.

Um otimizador para o escalonamento Halide com regras mais generalizadas foi implementado usando o framework OpenTuner. As diretivas de escalonamento do Halide foram codificadas no framework e as informações do *pipeline* como variáveis, funções e a relação entre produtor e consumidor de cada estágio do *pipeline* foram codificadas como parâmetros de entrada do framework. O sistema foi capaz de encontrar escalonamentos eficientes para *pipelines* mais simples, como por exemplo, o filtro bilateral (8 estágios), em aproximadamente uma hora de execução, porém não conseguiu convergir para soluções boas em *pipelines* mais complexos (Mullapudi et al., 2016; Denniston, 2016). O escalonamento produzido pelo OpenTuner para *pipelines* mais complexos como o camera raw (32 estágios), combinação de pirâmides (44 estágios) e interpolação com múltiplas escalas (49 estágios) foi cinco a dez vezes mais lento comparado ao escalonamento otimizado manualmente por programadores experientes na linguagem Halide (Mullapudi et al., 2015).

### 4.1.3 Escalonamento Automático

No trabalho de Mullapudi et al. (2016) foi apresentado uma abordagem estendendo a linguagem Halide para o compilador gerar automaticamente o escalonamento do *pipeline* sem a necessidade de processos de otimização muito custosos baseados na medida do tempo de execução. A solução estende o mecanismo de análise de limites do domínio de funções já existente no compilador da linguagem para automaticamente realizar melhorias globais no paralelismo e localidade de dados do *pipeline*.

A ideia principal é particionar as funções (estágios) do *pipeline* em grupos de funções e determinar um *tile* eficiente, possivelmente diferente, para cada grupo, sendo a comunicação entre cada grupo usando *buffers* em memória. Para cada função é estimado um custo aritmético e então um algoritmo busca um tamanho de *tile* que minimize o número de carga de dados necessários para processar o resultado. Para cada tamanho de *tile* candidato o escalonador automático também estima e compara o custo de mesclar grupos com o custo de não mesclar. No final os laços de processamento dentro de cada grupo de funções são ordenados de maneira a maximizar a localidade de dados e o paralelismo dos laços mais externos de acordo com a arquitetura alvo desejada.

Nas estimativas do custo de mesclar grupos de funções e na escolha do tamanho dos *tiles* são usados alguns parâmetros configurados de acordo com o hardware desejado, como tamanho do *cache* de nível mais alto, quantidade de paralelismo disponível que é relacionada ao número de *cores* existentes, um fator de custo relativo entre carregar um valor da memória principal versus computar um valor nos *cores* disponíveis. Para estimar o tamanho dos *tiles* também é considerado um parâmetro que indica o tamanho das operações de vetorização (instruções SIMD) suportadas no hardware, evitando *tiles* de tamanhos que não podem ser eficientemente vetorizados.

O mecanismo de escalonamento automático da linguagem foi capaz de gerar em segundos escalonamentos para um conjunto de *pipelines* de processamento de imagens usados para testes, apresentando desempenho competitivo com escalonamentos produzidos por programadores experientes na linguagem. Mullapudi et al. (2016) cita que embora o escalonamento automático seja gerado significativamente mais rápido, segundos ao invés de horas ou dias, as alternativas baseadas em processos de otimização por tempo de execução podem encontrar escalonamentos mais eficientes em muitas situações.

A ideia do escalonador automático do Halide foi baseada na técnica de escalonamento com agrupamento e *tile* apresentada na linguagem PolyMage (Mullapudi et al., 2015), entretanto, o Halide é mais abrangente uma vez que no PolyMage dependências e combinações *non-affine* de estágios do *pipeline* estão fora do escopo da linguagem devido ao uso de análise poliedral para determinar a ordem de aninhamento dos laços de processamento.

Outra linguagem de processamento de imagens, também com escalonamento automático, é a linguagem Darkroom (Hegarty et al., 2014), que utiliza uma técnica chamada *line-buffering* para produzir escalonamentos eficientes para dispositivos dedicados e específicos como ASICs (*Application Specific Integrated Circuit*) e FPGA (*Field Programmable Gate Array*), os quais tem suporte nativo a esse tipo de otimização, porém a implementação requer um algoritmo especificado ou transformado para suportar essa técnica de escalonamento, limitando a *pipelines* de operações *stencil* (máscara/janela) de tamanho fixo.

## 4.2 OTIMIZAÇÃO BASEADA EM REDE NEURAL

Falch e Elster (2015) propuseram uma abordagem usando rede neural para otimização de programas escritos com OpenCL (*Open Computing Language*). O OpenCL é uma arquitetura aberta para desenvolvimento de programas portáveis para diferentes plataformas, também usada na programação de algoritmos de processamento de imagens, entretanto, Falch e Elster (2015) explicam que mesmo havendo a portabilidade funcional do programa, normalmente para um bom desempenho é necessário otimização do programa para cada nova plataforma. A proposta apresentada usa um subconjunto do espaço de parâmetros de otimização para construir e treinar um modelo baseado em rede neural, o qual posteriormente é usado para selecionar apenas as partes mais promissoras de todo o espaço de busca, as quais então são avaliadas para identificar aquela que apresenta melhor desempenho. Esse modelo é ilustrado na figura 4.1.

O modelo da figura 4.1 inicia com um programa de entrada estruturado de maneira a receber um conjunto fixo de parâmetros que determinam um espaço de possíveis implementações, isto é, como será gerado o código final do programa. Em seguida, são geradas algumas amostras randômicas com valores para cada parâmetro e então o programa é executado registrando o tempo de execução obtido para cada amostra. Esses dados são usados no treinamento de uma rede neural para prever o tempo de execução de todas as configurações possíveis dos parâmetros.

Cada configuração possível é processada através do modelo da rede neural. No final, são selecionadas apenas as configurações que produziram melhor desempenho de acordo com a previsão gerada, as quais são então submetidas à execução no programa de entrada para determinar a configuração com melhor desempenho real. Se a rede neural estiver suficientemente correta, é esperado que a configuração ótima esteja entre aquelas selecionadas para medir o tempo de execução real no final do processo (Falch e Elster, 2015).

De uma maneira geral, a intenção dessa abordagem é criar uma função de aproximação do tempo de execução para usá-la como uma função substituta (*surrogate function*) e assim reduzir a quantidades de execuções reais do programa, entretanto, o modelo proposto requer que todas as configurações possíveis sejam previamente conhecidas e estruturadas no programa,

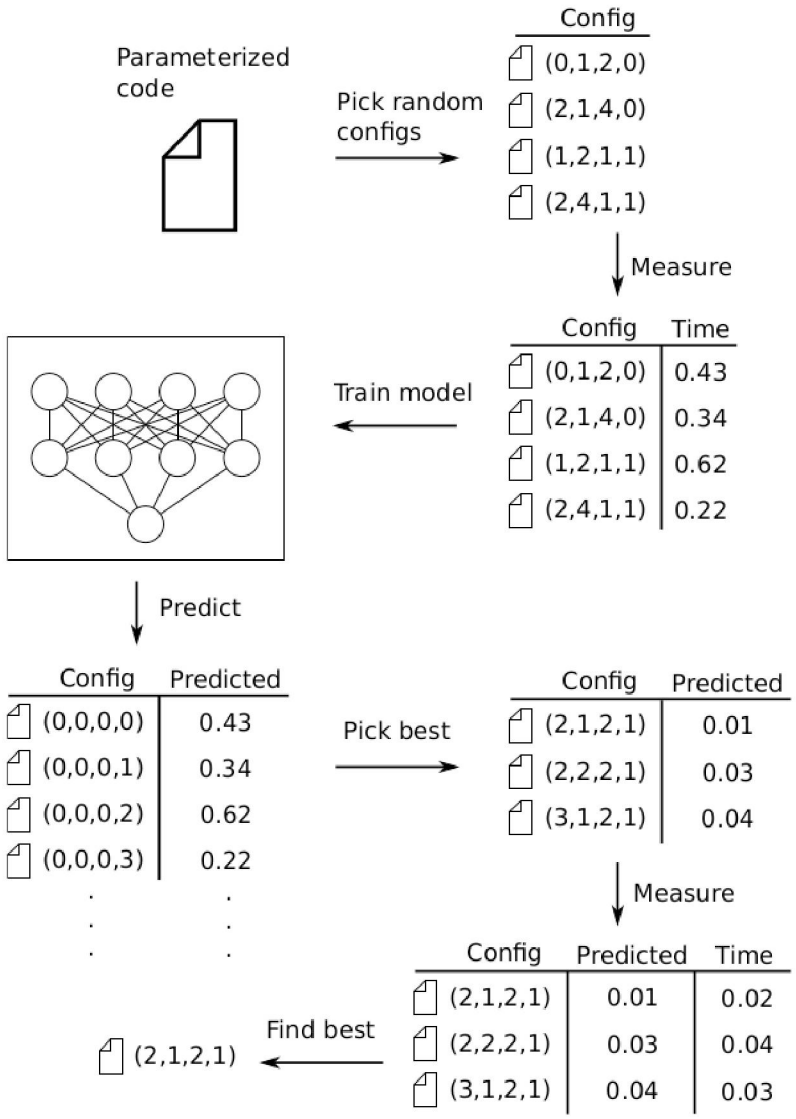



Figura 4.1: Modelo de otimização baseado em rede neural proposto por Falch e Elster (2015). O modelo utiliza um subconjunto do espaço de parâmetros de otimização, previamente avaliado, para construir e treinar uma rede neural. Depois a rede é utilizada para avaliar novas configurações e selecionar apenas as mais promissoras, as quais então serão efetivamente testadas para identificar aquela que apresenta melhor desempenho.

restando escolher quais serão usadas. Além disso, todas as configurações são avaliadas usando a rede neural previamente treinada. Ainda que passar uma configuração pela rede já treinada é um processo bem rápido, se o número de configurações disponíveis for muito grande, pode inviabilizar essa etapa. Também, nesse caso, seria necessário uma quantidade expressiva de execuções reais para treinar a rede adequadamente.

## 4.3 CONCLUSÃO

Neste capítulo foi apresentado uma síntese das abordagens de otimização do escalonamento Halide disponíveis na literatura, relatando a ideia principal e algumas de suas características e limitações. Uma das abordagens relatadas emprega algoritmos genéticos (Ragan-Kelley et al., 2013), cujo principal problema evidenciado foi a necessidade de utilização de heurísticas de mutação específicas para cada *pipeline*, tornando-se uma solução inadequada pela falta de

generalização. Outra abordagem (Ansel et al., 2014b), que emprega um conjunto de técnicas de busca, obteve melhor generalização, porém apresentou falhas para convergir em *pipelines* grandes, com diversos estágios, além de produzir, nesses cenários, escalonamentos com desempenho significativamente inferior aqueles elaborados por programadores Halide (Mullapudi et al., 2015).

A última abordagem identificada (Mullapudi et al., 2016), e ainda em evolução, utiliza regras determinísticas para analisar o *pipeline* e sugerir um escalonamento. Essa abordagem obteve bons resultados nos cenários avaliados, no entanto, justamente por ser baseada em regras de análise previamente definidas, ela pode não ser muito eficaz em cenários novos, que ainda não possuem regras específicas incorporadas à solução. Uma outra abordagem (Falch e Elster, 2015), usando aprendizado supervisionado através de rede neural, também foi apresentada, e apesar de não estar diretamente relacionada ao Halide, aborda o mesmo problema tentando criar um mecanismo de otimização de programas para diferentes plataformas, servindo assim como exemplo de como uma rede neural pode ser aplicada na otimização de programas.

Contudo, como nenhuma das técnicas de otimização do escalonamento Halide identificadas na literatura emprega aprendizado por reforço, experimentos com essa abordagem podem contribuir para determinar se essa técnica pode ser uma solução promissora para o problema em questão, abrindo caminho para o eventual desenvolvimento de uma solução futura. Nesse contexto, a partir do capítulo seguinte, é apresentado uma abordagem de aplicação do modelo de aprendizado por reforço para geração automática do escalonamento Halide.

# 5 OTIMIZAÇÃO DO ESCALONAMENTO HALIDE ATRAVÉS DE APRENDIZADO POR REFORÇO

A otimização do escalonamento Halide se refere a tarefa de encontrar um escalonamento de execução (*schedule* Halide) para um determinado *pipeline* de processamento de imagem e plataforma de hardware que resulte em um menor tempo de execução. A solução desenvolvida neste trabalho é apresentada de maneira geral no diagrama da figura 5.1.

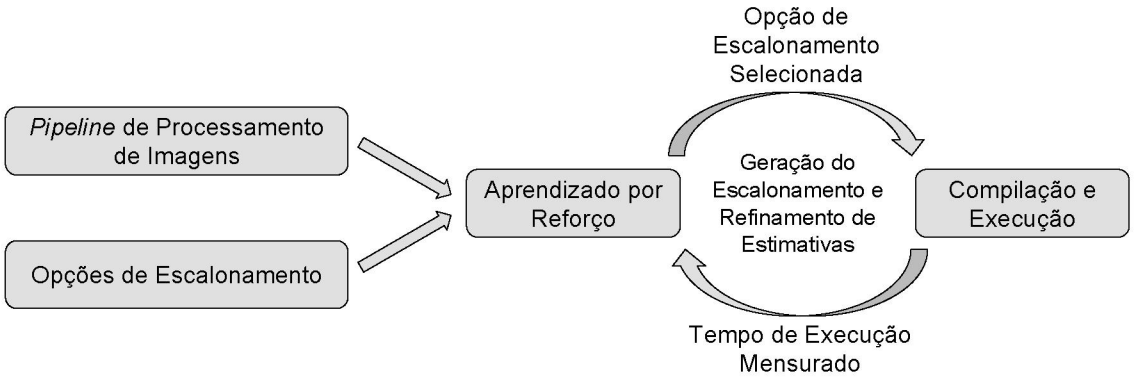



Figura 5.1: Visão geral da solução desenvolvida. O *pipeline* de processamento de imagens e as opções de escalonamento disponíveis para esse *pipeline* são inicialmente carregadas no ambiente de aprendizado por reforço, do qual um agente de aprendizado por reforço passa iterativamente a gerar escalonamentos de execução escolhendo e executando uma opção de escalonamento a cada vez, refinando estimativas para as próximas escolhas com base nas informações obtidas das escolhas anteriores. O processo inicia randomicamente, mas gradualmente, a partir da experiência acumulada, o agente aprende a escolher opções de escalonamento cada vez melhores.

A solução desenvolvida emprega o aprendizado por reforço como uma técnica para gerar escalonamentos de execução otimizados. A solução recebe como entrada o *pipeline* de processamento de imagem que se deseja otimizar e um conjunto de opções de escalonamento para esse *pipeline*, conjunto este derivado das diretivas de escalonamento disponibilizadas pela linguagem Halide. O aprendizado por reforço se encarrega então de aprender quais opções de escalonamento são melhores para otimizar o *pipeline*. A aprendizagem ocorre de maneira iterativa, escolhendo uma opção de escalonamento e testando o desempenho obtido, gerando e refinando estimativas com informações coletadas nos testes executados. Gradualmente, com a experiência acumulada, opções de escalonamento cada vez melhores são selecionadas, sendo que as estimativas produzidas pelo aprendizado por reforço ajudam a escolher opções de escalonamento com maiores probabilidades de gerar bons resultados, mesmo aquelas que ainda não foram efetivamente testadas.

Os programas de processamento de imagem implementados em Halide, através do paradigma funcional, representam um *pipeline* de processamento composto por estágios, no qual cada estágio pode conter diversas operações aritméticas, lógicas, e dependências de dados de estágios anteriores. A implementação de um programa em Halide é dividida em duas partes: primeiro a definição da lógica e regras do algoritmo, depois a definição do escalonamento de execução. Como essas duas partes são independentes na linguagem, é possível experimentar diferentes variações do escalonamento de execução sem afetar a lógica ou resultado do programa. Essa característica também permite criar mecanismos automatizados para explorar alternativas de escalonamento de execução, buscando aquela que apresenta melhor desempenho.

Na prática, mesmo na programação manual o programador realiza um processo similar. Partindo de um *pipeline* desenvolvido, começa a elaborar e testar diferentes escalonamentos, buscando aquele que obtém melhor desempenho. Para cada teste realizado, avalia o resultado tentando melhorar nos passos seguintes, gerando uma nova versão do escalonamento, possivelmente melhor, repetindo o processo até o ponto que considera não conseguir melhorar mais ou até consumir o prazo disponível. A cada novo experimento realizado, o programador acumula mais e mais experiência. Esse processo consome algum tempo, dependendo da experiência do programador.

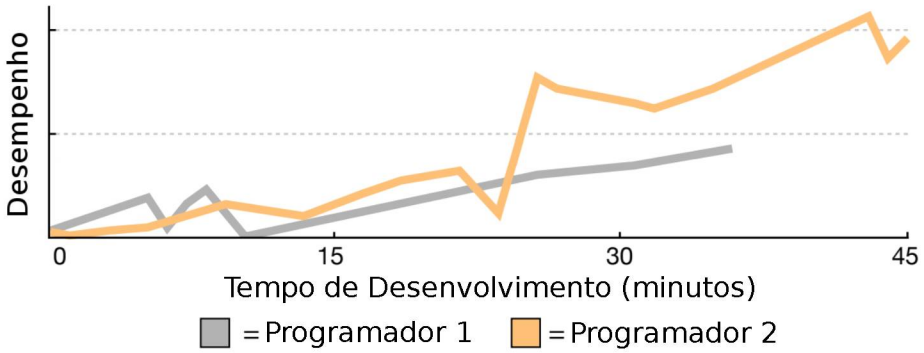



Figura 5.2: Desenvolvimento Manual do Escalonamento de Execução. Adaptado de Mullapudi et al. (2016).

Por exemplo, para ilustrar essa dinâmica, Mullapudi et al. (2016) apresentou o gráfico da figura 5.2 comparando o desempenho obtido e o tempo gasto por dois programadores experientes para desenvolver um escalonamento para um dado *pipeline* de teste que estes ainda não haviam implementado, até o ponto que consideravam ter atingido o desempenho máximo. Essa dinâmica apresentada pode ser vista como resultado de um processo iterativo de otimização. Mesmo sem conhecer previamente todo o problema ou quais seriam as condições para encontrar a melhor solução, os programadores conseguem gradualmente melhorar experimentando operações e parâmetros diferentes, aprendendo quais são mais eficazes para o problema específico e quais são menos eficazes ou não geram ganho representativo.

Essa dinâmica também é muito similar às técnicas empregadas nos agentes de aprendizado por reforço. Diferentemente de uma abordagem determinística que seria capaz de produzir a solução quase imediatamente, mas que depende de haver um conjunto de regras e passos para isso previamente conhecidos, a abordagem iterativa demanda sucessivas repetições, porém tem a vantagem de poder ser aplicada nos casos em que ainda não há uma solução determinística para resolver o problema.

Conforme ilustrado no código 5.1, um escalonamento de execução Halide também pode ser visto como uma sequência de diretivas (operações) de escalonamento aplicadas aos estágios do *pipeline* desejado, tendo como objetivo final chegar a uma sequência que maximize o desempenho, ou seja, que tenha um menor tempo de execução.

Código 5.1: Exemplo de Sequência de Diretivas de Escalonamento.

```
1 Func blur_x, blur_y;
2 Var x, y, xi, yi;
3
4 // Algoritmo / pipeline de processamento
5 blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
6 blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;
7
8 // Sequência de diretivas do escalonamento de execução
```

```
9  blur_y.tile(x, y, xi, yi, 256, 32);
10 blur_y.vectorize(xi, 8);
11 blur_y.parallel(y);
12 blur_x.compute_at(blur_y, x);
13 blur_x.vectorize(x, 8);
```

Com base no exposto acima, o presente trabalho aplica uma abordagem de exploração das alternativas de escalonamento de execução usando aprendizado por reforço, por meio de um agente PPO.

## 5.1 FORMALIZAÇÃO DO PROBLEMA

Seja $h$ um *pipeline* de processamento de imagem implementado na linguagem Halide, $S_h$ o conjunto de alternativas de escalonamento de execução possíveis para o *pipeline* $h$, e $f(h,s)$ uma função representando o tempo de execução do *pipeline* $h$ para um escalonamento $s \in S_h$ aplicado a um determinado hardware, o problema considerado neste trabalho é encontrar um $s \in S_h$ que minimize $f(h,s)$ conforme a equação 5.1:

$$\underset{s \in S_h}{argmin}\ f(h,s) \tag{5.1}$$

Essa formalização é similar a sugerida por Chen et al. (2018), contudo, como o conjunto $S$ é derivado do conjunto de diretivas de escalonamento $D$ suportados na linguagem Halide, um novo conjunto $S' \subseteq S$ pode ser definido limitando o número de opções disponíveis através de um mapeamento $S' = m(h,D)$ aplicado aos estágios do *pipeline* $h$. Dessa maneira a equação 5.1 pode atuar sobre o conjunto $S'_h$ de tamanho menor que $S_h$.

Porém, como um escalonamento de execução é formado por uma sequência de diretivas de escalonamento e algumas opções podem depender de outras para que sejam operações válidas, a ordem é relevante durante a escolha de cada escalonamento. Além da ordem, a quantidade de diretivas necessárias para produzir um escalonamento otimizado para um dado *pipeline* também precisa ser determinada durante o processo de otimização, pois não é previamente conhecida.

## 5.2 AMBIENTE DE APRENDIZADO POR REFORÇO

Para viabilizar a aplicação do aprendizado por reforço é necessário modelar o problema como um MDP e possuir um ambiente (*environment*) que represente o problema, definindo uma representação para estados, ações e recompensas. Conforme explicado no capítulo 3, é através desse ambiente que o agente de aprendizado por reforço interage com o problema modelado e executa o processo de otimização provido pelo agente.

A figura 5.3 ilustra os principais componentes do aprendizado por reforço e a relação entre eles, conforme exposto no capítulo 3. No caso do ambiente, foi necessário desenvolver uma implementação especificamente para este trabalho, com a capacidade de interagir com a linguagem Halide e os *pipelines* de processamento de imagens, enquanto que para o agente foi utilizado implementações já disponíveis em bibliotecas *open source*. Os estados, ações e recompensas, cujas representações são compartilhadas entre o agente e o ambiente e precisam ser compreendidas por ambos, são definidas dentro do ambiente e foram elaboradas conforme apresentado nas seções seguintes.

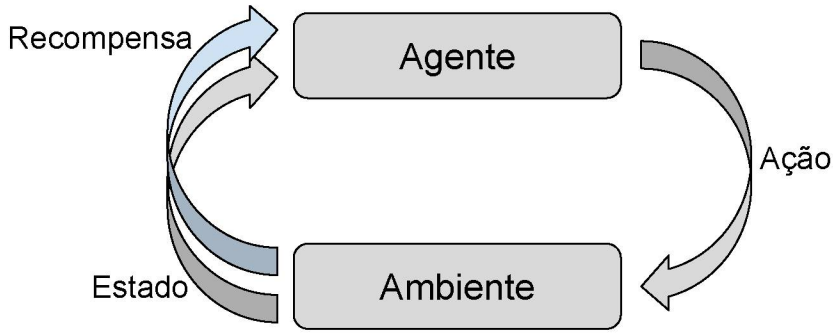



Figura 5.3: Relação entre os componentes do aprendizado por reforço. A interação entre o ambiente e o agente é baseada em ações que o agente envia para o ambiente e em estados e recompensas enviadas do ambiente para o agente.

### 5.2.1 Representação do Estado

Para representar o estado do ambiente de aprendizado por reforço foi utilizado a situação corrente do próprio escalonamento de execução sendo gerado para um dado *pipeline* de processamento de imagem, ou seja, o conjunto de diretivas já utilizadas na definição do escalonamento.

Conforme apresentado no capítulo 2, as diretivas de escalonamento são aplicadas aos estágios do *pipeline* e possuem parâmetros de diferentes tipos, de acordo com cada diretiva. O valor informado aos parâmetros das diretivas, bem como a qual estágio do *pipeline* a diretiva foi aplicada, naturalmente, afetam o escalonamento resultante, e portanto também são importantes para a representação do estado do ambiente.

Contudo, o escalonamento precisa ser representado sob a forma numérica para possibilitar a utilização do aprendizado por reforço. Assim, individualmente, cada estágio do *pipeline*, diretiva e parâmetro utilizado são mapeados para identificadores numéricos com um código único. Dessa maneira, um escalonamento completo, com várias diretivas, se torna um vetor de dados numéricos.

A seguir, na tabela 5.1, são apresentados exemplos ilustrando como ficam os identificadores numéricos atribuídos individualmente a cada tipo de elemento do Halide que pode compor um escalonamento de execução, sejam estes elementos estágios do *pipeline*, diretivas e ou mesmo valores possíveis para os parâmetros das diretivas. Os elementos utilizados no exemplo são referentes ao código 5.1 apresentado anteriormente.

Como a linguagem Halide é efetivamente uma biblioteca C++, conforme explicado no capítulo 2, cada elemento da linguagem é implementado como uma estrutura ou classe C++, assim, ao utilizar um elemento Halide na composição do escalonamento atual, na prática estão sendo utilizadas referências para objetos em memória. Então para definir identificadores numéricos a cada elemento utilizado no escalonamento, e assim criar a representação do estado atual, são armazenadas em um lista as referências de todos os objetos utilizados, sendo definido uma lista para cada tipo de elemento suportado no Halide. Dessa maneira, sempre que uma nova referência é incluída na lista correspondente ao tipo do elemento, o objeto referenciado recebe um índice de identificação numérico, o qual é utilizado para compor a representação do estado. Porém, sempre que um dado elemento Halide utilizado no escalonamento já existe na lista, então o mesmo identificador numérico será utilizado. No caso de parâmetros de diretivas que podem receber valores discretos, o próprio valor é inserido em uma lista e o índice usado como identificador.

Tabela 5.1: Exemplo de Identificadores atribuídos à Estágios, Diretivas e Parâmetros de Diretivas.

| Tipos de Elemento do Halide | | Elemento | Identificador |
|---|---|---|---|
| Estágios do *Pipeline* | | `blur_x` | 001 |
| | | `blur_y` | 002 |
| Diretivas de Escalonamento | | `tile` | 101 |
| | | `vectorize` | 102 |
| | | `parallel` | 103 |
| | | `compute_at` | 104 |
| Parâmetros de Diretivas | Variáveis de Domínio | `x` | 201 |
| | | `y` | 202 |
| | | `xi` | 203 |
| | | `yi` | 204 |
| | Valores Discretizados | `8` | 301 |
| | | `32` | 302 |
| | | `256` | 303 |

Usando os identificadores do exemplo apresentado na tabela 5.1 acima, o escalonamento do código 5.1 anterior pode ser representado conforme a tabela 5.2. Dessa maneira, unindo todas as representações individuais de cada elemento, tem-se um vetor de identificadores que corresponde ao estado do ambiente. Esse vetor também preserva a posição onde cada elemento Halide aparece dentro do escalonamento correspondente.

Tabela 5.2: Exemplo de representação do estado do ambiente de aprendizado por reforço. O estado corresponde ao escalonamento corrente mapeado para um vetor de dados numérico equivalente.

| Diretivas de Escalonamento Utilizadas | Vetor de Representação do Estado |
|---|---|
| `blur_y.tile(x, y, xi, yi, 256, 32);` | 002 101 201 202 203 204 303 302 |
| `blur_y.vectorize(xi, 8);` | 002 102 203 301 000 000 000 000 |
| `blur_y.parallel(y);` | 002 103 202 000 000 000 000 000 |
| `blur_x.compute_at(blur_y, x);` | 001 104 002 201 000 000 000 000 |
| `blur_x.vectorize(x, 8);` | 001 102 201 301 000 000 000 000 |

### 5.2.2 Representação das Ações

Para representar as ações do aprendizado por reforço foi considerado o conjunto de diretivas de escalonamento do Halide disponíveis para serem utilizadas na geração do escalonamento de execução de um dado *pipeline* de processamento de imagem. Dessa maneira, quando o agente de aprendizado por reforço enviar uma ação para o ambiente, significa aplicar uma diretiva de escalonamento ao *pipeline* de processamento de imagem.

Conforme exposto anteriormente, as diretivas de escalonamento do Halide são aplicadas aos estágios do *pipeline* e podem possuir diferentes parâmetros, de acordo com cada diretiva. Assim, para representar uma única ação do aprendizado por reforço é necessário considerar também a qual estágio do *pipeline* a diretiva será aplicada e quais serão os valores informados para cada parâmetro da diretiva. Além disso, as ações também precisam ser representadas de forma numérica para possibilitar a utilização do aprendizado por reforço.

Para implementar essa representação das ações, cada combinação entre estágios do *pipeline*, diretivas de escalonamento, e valores possíveis para os parâmetros das diretivas, são

mapeados para um intervalo de códigos inteiros, no qual cada valor nesse intervalo representa uma opção possível, ou seja, um estágio e uma diretiva com seus argumentos para cada parâmetro. O conjunto de todas as opções mapeadas representa o espaço de ações disponível ao agente de aprendizado por reforço.

Na tabela 5.3 são apresentados alguns exemplos ilustrando como fica o mapeamento das diretivas de escalonamento para ações do aprendizado por reforço, e vice-versa. Cada número de ação identifica unicamente uma combinação entre estágio, diretiva e argumentos para parâmetros da diretiva. Note que os valores apresentados são apenas exemplos, os valores reais associados são alocados conforme necessário durante a inicialização do ambiente de aprendizado por reforço, de acordo com o *pipeline* de entrada.

Tabela 5.3: Representação do mapeamento de diretivas de escalonamento para ações do aprendizado por reforço, e vice-versa. Cada número de ação identifica unicamente uma combinação entre estágio, diretiva e argumentos para parâmetros da diretiva.

| Diretivas de Escalonamento | Ações do Aprendizado por Reforço |
|---|---|
| `blur_x.compute_at(blur_y, x)` | 1 |
| `blur_x.compute_at(blur_y, y)` | 2 |
| `blur_x.store_at(blur_y, x)` | 3 |
| `blur_x.store_at(blur_y, y)` | 4 |
| `blur_y.parallel(x)` | 5 |
| `blur_y.parallel(y)` | 6 |

Contudo, o mapeamento das diretivas de escalonamento para ações do aprendizado por reforço depende da elicitação de quais opções de escalonamento serão consideradas, ou seja, quais estágios, quais diretivas e quais valores para parâmetros das diretivas serão mapeados. Essas opções são previamente definidas para cada *pipeline* de processamento de imagem através de um mecanismo de mapeamento desenvolvido neste trabalho, e então passadas como uma entrada para o aprendizado por reforço.

O mecanismo de mapeamento foi implementado em uma classe C++ chamada `HalideScheduleMapper`, a qual encapsula as diretivas de escalonamento existentes na linguagem Halide, permitindo ao programador elaborar o mapeamento das opções desejadas de maneira muito similar a sintaxe da linguagem Halide. Por exemplo, para o algoritmo *Blur*, o mapeamento apresentado na tabela 5.3 acima pode ser definido através do código 5.2 abaixo.

Código 5.2: Exemplo de mapeamento das opções de escalonamento que serão consideradas para representar as ações do aprendizado por reforço. O mapeamento considera o estágio do *pipeline*, a diretiva de escalonamento e possíveis argumentos para parâmetros da diretiva.

```
1  // Algoritmo / pipeline de processamento
2  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
3  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;
4
5  // Mapeamento das opções de escalonamento
6  HalideScheduleMapper m;
7  m.map(blur_x)
8          .compute_at({blur_y}, {x, y})
9          .store_at({blur_y}, {x, y});
10 m.map(blur_y)
11         .parallel({x, y});
```

De maneira geral, a classe `HalideScheduleMapper` permite que para cada parâmetro esperado, de cada tipo de diretiva, seja informado uma lista de argumentos possíveis, a qual

será armazenada internamente na classe para utilização posterior, contabilizando e acumulando a quantidade de opções mapeadas para cada estágio do *pipeline*. O total acumulado ao final do mapeamento representa o espaço de ações disponíveis ao agente de aprendizado por reforço. Quando o agente está em execução, a classe `HalideScheduleMapper` recebe um número de ação produzido pelo agente e decodifica esse valor, recuperando o estágio do *pipeline*, a diretiva e os argumentos correspondentes ao número da ação informado.

A implementação do encapsulamento das diretivas Halide realizado através da classe `HalideScheduleMapper` define um método para cada diretiva e utiliza um objeto tipo `vector` para receber a lista de argumentos possíveis para cada tipo de parâmetro da diretiva, respeitando os tipos de dados originais de cada diretiva, definindo assim uma assinatura de método conforme ilustrado nos exemplos do código 5.3 abaixo. Apenas para conveniência, cada método após armazenar internamente as informações de mapeamento, retorna a própria instância `this` da classe de encapsulamento.

Código 5.3: Assinatura de método usado na classe de encapsulamento das diretivas de escalonamento do Halide. Os mesmos nomes das diretivas e tipos de dados são utilizados, porém permite passar uma lista de argumentos para cada parâmetro ao invés de um único argumento.

```
1 // Diretiva: compute_at(Func f, Var v)
2 HalideScheduleMapper& compute_at(vector<Func> f, vector<Var> v)
3
4 // Diretiva: store_at(Func f, Var v)
5 HalideScheduleMapper& store_at(vector<Func> f, vector<Var> v)
6
7 // Diretiva: parallel(VarOrRVar v)
8 HalideScheduleMapper& parallel(vector<VarOrRVar> v)
```

Esse mecanismo de mapeamento permite ao programador Halide informar opções de escalonamento mais plausíveis, descartando opções ineficientes ou inválidas, de acordo com a experiência profissional e conhecimento do algoritmo de processamento de imagem em questão, reduzindo assim o tamanho do espaço de ações que o agente de aprendizado por reforço precisará explorar.

### 5.2.3 Representação da Recompensa

A representação da recompensa é baseada no tempo de execução obtido com a aplicação de uma dada diretiva (ação) no escalonamento atual (estado). A recompensa é representada por um valor escalar retornado pelo ambiente de aprendizado por reforço após executar uma ação, na qual valores positivos significam que a ação gerou uma redução no tempo de execução do *pipeline* de processamento de imagem, ou seja, que melhorou, o valor zero indica que não houve redução do tempo, e valores negativos indicam que causou algum erro na execução.

Conforme apresentado na equação 5.2, o cálculo do valor da recompensa considera a diferença do tempo de execução do escalonamento produzido pela ação atual (*curr_exectime*) e a pela ação anterior (*prev_exectime*), multiplicada por um fator de ajuste de escala (*reward_scale*), de maneira que quanto menor o tempo de execução obtido, maior será o valor da recompensa.

Porém, quando o tempo de execução aumentar em relação ao anterior, a recompensa é fixada em zero, evitando valor negativo.

$$Recompensa = \begin{cases} (prev\_exectime - curr\_exectime) * reward\_scale & \text{if } curr < prev \\ 0 & \text{if } curr >= prev \\ -1 & \text{if } error \end{cases} \tag{5.2}$$

O fator de ajuste de escala foi calculado conforme equação 5.3, durante a inicialização do ambiente de aprendizado por reforço, usando o valor inverso do tempo inicial de execução do *pipeline* de processamento de imagem (*init_exectime*) obtido antes do agente começar a interagir com o ambiente. Dessa maneira, mesmo para distintos *pipelines* que tenham tempo de execução bastante diferentes, a recompensa terá uma amplitude similar entre os diferentes *pipelines*, proporcional ao ganho gerado. O fator de escala da recompensa é citado por Henderson et al. (2017) como um aspecto relevante que pode influenciar a evolução do agente de aprendizado por reforço.

$$reward\_scale = \frac{100}{init\_exectime} \tag{5.3}$$

### 5.2.4 Operações do Ambiente

A interação entre o ambiente e o agente foi viabilizada por meio de três operações principais definidas e implementadas dentro do ambiente de aprendizado por reforço elaborado neste projeto. Essas operações são invocadas pelo agente para se comunicar com o ambiente. Os pontos mais relevantes de cada operação são detalhados a seguir:

- **`init`:** Esta operação é chamada na inicialização do aprendizado por reforço. Ela é responsável por definir o estado inicial do ambiente, bem como determinar o espaço de ações que estará disponível ao agente, por meio do carregamento das opções de escalonamento e do *pipeline* de processamento de imagem fornecidos como entradas para o aprendizado por reforço.

  Durante a inicialização também é coletado o tempo de execução inicial do *pipeline* informado (*init_exectime*), sem acrescentar diretivas de escalonamento.

- **`step`:** Esta operação é chamada pelo agente a cada passo de tempo, iterativamente, durante toda a execução do aprendizado por reforço. Na chamada o agente fornece como entrada um ação e recebe como saída um novo estado do ambiente, uma recompensa imediata da ação, e um sinal que indica se é um estado terminal, que sinaliza o fim de um episódio. As ações, estados e recompensas seguem as definições apresentadas acima.

  Nesta operação o ambiente decodifica a ação recebida como entrada, usando as informações de mapeamento da classe `HalideScheduleMapper`, e adiciona a diretiva correspondente ao escalonamento atual, modificando assim o estado do ambiente, então executa o *pipeline* com o novo escalonamento e obtém o tempo de execução usado no cálculo da recompensa.

  Caso ocorra algum erro ao processar uma ação, o ambiente descarta a última ação, sem alterar o estado do ambiente. Nesse caso a recompensa sempre terá valor `-1`. Um erro pode ocorrer em uma das seguintes situações:

  - Falha na compilação ou execução do escalonamento produzido pela última ação.
  - Ultrapassar um limite de diretivas permitidas para um mesmo estágio do *pipeline*. Similar ao controle adotado por Ragan-Kelley et al. (2013), esse limite é uma propriedade configurável do ambiente usada para prevenir que o agente produza escalonamentos exorbitantes ou muito complexos sem ganho de desempenho.

  O estado terminal é sinalizado quando ocorrer uma das seguintes situações:

  - Receber uma ação especial de controle, chamada *no-operation*, que apenas encerra o escalonamento atual sem qualquer modificação. Nesse caso a recompensa sempre terá valor `0`.
  - Atingir o menor tempo de execução, ou seja, melhor desempenho, desde a inicialização do ambiente. Note que, nessa situação, a recompensa total do episódio, somadas as ações do estado inicial até o estado terminal, também vai corresponder ao melhor valor obtido até o momento.

- **`reset`:** Esta operação é chamada pelo agente toda vez que o ambiente atingir um estado terminal, após o retorno da operação `step`. Ela é responsável por reiniciar o estado do ambiente e começar um novo episódio, porém, o agente mantém o aprendizado acumulado até o momento.

  O estado do ambiente referente a um novo episódio corresponde ao estado carregado na inicialização, através da operação `init`, somado a uma diretiva de escalonamento inicial escolhida randomicamente entre o espaço de ações disponível. Na prática, toda vez que começar um novo episódio, a situação do escalonamento inicial, bem como do estado do ambiente, será diferente.

## 5.3 FRAMEWORK DE OTIMIZAÇÃO

O framework desenvolvido para otimização do escalonamento Halide através da aplicação do aprendizado por reforço é apresentado na figura 5.4. O framework possui um módulo Python responsável pela heurística de otimização propriamente, provida pelo agente PPO, cuja implementação utilizada é disponibilizada pela biblioteca OpenAI Baselines (Dhariwal et al., 2017). O agente interage com o problema modelado por meio do ambiente de aprendizado por reforço que foi construído utilizando a biblioteca OpenAI Gym (Brockman et al., 2016), a qual dispõem de interfaces padronizadas de comunicação com o agente e da infraestrutura base necessária para essa finalidade. O ambiente recebe ações do agente e retorna os novos estados e recompensas correspondentes.

O framework também possui um módulo C++ responsável por executar os programas Halide de processamento de imagens, além de manter as opções de escalonamento mapeadas para cada *pipeline* de processamento de imagem. O ambiente de aprendizado por reforço em Python se comunica com a classe C++ de mapeamento das opções de escalonamento por meio de mensagens RPC (*Remote Procedure Call*) síncronas, implementadas através das bibliotecas gRPC/Protobuf[1]. A classe de mapeamento é responsável por decodificar as ações recebidas pelo ambiente e aplicar as diretivas correspondentes no programa Halide, retornando os identificadores numéricos de representação das diretivas, bem como os tempos de execução mensurados.

[1] *gRPC/Protocol Buffers* - `https://grpc.io`

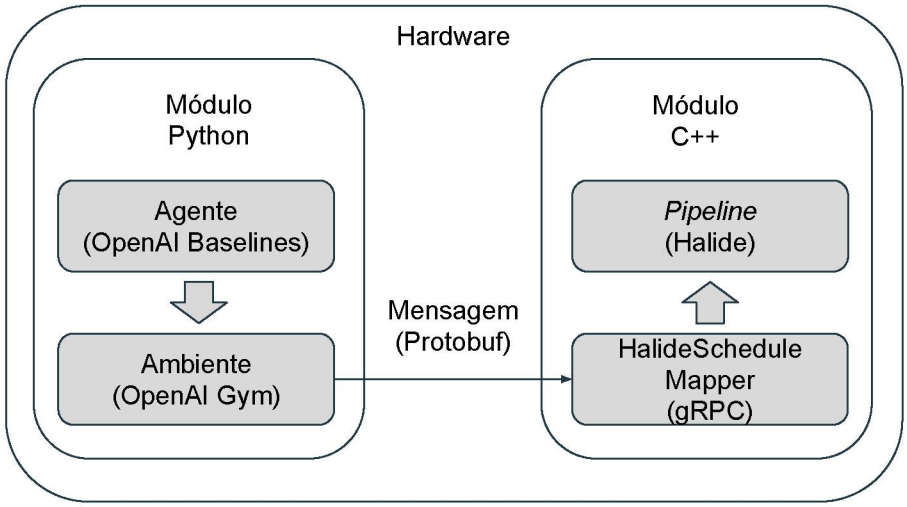



Figura 5.4: Framework desenvolvido para otimização do escalonamento Halide através de aprendizado por reforço. Um módulo Python responsável pela otimização e aprendizado propriamente se comunica com um módulo C++ que faz a interface com os programas Halide. A implementação de ambos os módulos foi baseada em bibliotecas de projetos *open source*.

Com a arquitetura do framework apresentado na figura 5.4 também é possível utilizar, caso desejado, máquinas físicas distintas para executar o agente de aprendizado por reforço e para executar os programas Halide de processamento de imagens.

## 5.4 CONCLUSÃO

Neste capítulo foi exposto como funciona a solução desenvolvida usando aprendizado por reforço como uma abordagem para explorar o problema da otimização do escalonamento Halide. A solução apresentada utiliza frameworks abertos de aprendizado por reforço que oferecem a infraestrutura necessária para o projeto. Também foi apresentado como o problema foi modelado para viabilizar a aplicação da técnica sugerida, indicando qual agente de aprendizado por reforço foi usado no trabalho. No capítulo seguinte será apresentado o protocolo experimental, configurações e parâmetros considerados nos experimentos realizados.

# 6 MATERIAL E MÉTODO

A seguir são apresentados os cenários considerados e quais testes foram realizados para avaliação da abordagem descrita no capítulo anterior, citando configurações e parâmetros adotados na realização dos experimentos.

## 6.1 *PIPELINES* DE PROCESSAMENTO DE IMAGENS

O método de otimização baseado em aprendizado por reforço foi aplicado, separadamente, a três *pipelines* de processamento de imagens implementados e disponíveis no repositório da linguagem Halide[1]. Os *pipelines* são utilizados como entrada para o aprendizado por reforço.

Cada *pipeline* recebe uma imagem de entrada e passa essa imagem por um processamento composto por múltiplos estágios, produzindo como resultado uma outra imagem de saída. Em todos os casos a implementação disponível foi escrita puramente na linguagem Halide, conforme apresentado no apêndice A.

Os *pipelines* mencionados são os seguintes:

1. ***Blur*:** Filtro de suavização de imagens implementado em Halide com um *pipeline* de apenas 2 estágios, o qual recebe como entrada imagens em níveis de cinza e produz saída no mesmo formato.

2. **Harris:** Detector de cantos Harris (Harris e Stephens, 1988), cuja implementação em Halide apresenta um *pipeline* de processamento com 13 estágios, recebendo como entrada imagens coloridas e produzindo saída em níveis de cinza.

3. **Interpolação:** Interpolação de pixels de imagem usando pirâmides, trabalhando com diferentes escalas de resolução e dependências de dados (Mullapudi et al., 2016), totalizando um *pipeline* de processamento com 52 estágios e 10 níveis de pirâmides, recebendo como entrada imagens coloridas com canal alfa e produzindo saída no mesmo formato, sem canal alfa.

Na avaliação de cada *pipeline* acima foram considerados três tamanhos de imagens de entrada para cada caso, conforme listado abaixo, em formato compatível com a entrada esperada.

- **Imagem 1:** Dimensão de 962x642, contendo 0.6M pixels.
- **Imagem 2:** Dimensão de 1924x1284, contendo 2.4M pixels.
- **Imagem 3:** Dimensão de 3848x2568, contendo 9.8M pixels.

Na execução do aprendizado por reforço para geração do escalonamento de cada *pipeline* foi utilizado a terceira imagem, de tamanho maior, depois, para testes individuais de avaliação de desempenho do escalonamento gerado e comparação com outras abordagens foram utilizadas as três imagens.

Uma vez que os *pipelines* de processamento de imagens citados não apresentam expressões condicionais dependentes de dados em sua parte lógica implementada em Halide, o conteúdo da imagem propriamente não causa impacto relevante no tempo de execução dos mesmos.

[1] `https://github.com/halide/Halide`

## 6.2 OPÇÕES DE ESCALONAMENTO

As opções de escalonamento disponibilizadas como entrada para o aprendizado por reforço foram mapeadas a partir de exemplos de escalonamentos disponíveis no repositório da linguagem Halide e a partir do escalonamento automático gerado pela linguagem para os *pipelines* avaliados, conforme seção 4.1.3. Um mapeamento foi elaborado para cada *pipeline* de processamento de imagem e arquitetura de hardware utilizada.

No mapeamento das opções de escalonamento foram consideradas as seguintes alternativas para parâmetros das diretivas de escalonamento:

- Tamanhos para cada dimensão das janelas de processamento (*tile* e *split*): 8, 16, 32, 64, 128, 256, 512.
- Tamanhos para operações com instruções de vetorização (*vectorize*): 4, 8, 16.
- Limites de decomposição de laços de processamento em operações subsequentes (*unroll*): 2, 3, 4.
- Posições alternativas para processamento de estágios intermediários e armazenamento de resultados parciais (*compute_root*, *compute_at*, *store_at*), configuradas de acordo com a dependência de dados entre os estágios de cada *pipeline*.
- Possibilidades de paralelizar trabalhos (*parallel*) em diferentes estágios do *pipeline*.
- Possibilidades de fixar os limites conhecidos das dimensões (*bound*) nas variáveis de domínio usadas em estágios do *pipeline*.

As mesmas alternativas acima foram utilizadas em ambas as arquiteturas CPU e GPU, de acordo com a necessidade das diretivas de cada arquitetura. O mapeamento elaborado pode ser consultado nos apêndices B e C.

## 6.3 CONFIGURAÇÕES DE HARDWARE E SOFTWARE

Os experimentos referente a cada um dos *pipelines* de processamento de imagens citados acima foram executados em duas arquiteturas de hardware distintas: CPU (x86) e GPU (NVIDIA).

Na execução dos experimentos foram utilizadas as seguintes especificações de hardware:

- CPU Intel Xeon E5-4627 v2 3.30GHz 16MB Cache, 32 Core, 4 Socket, 256GB NUMA
- GPU NVIDIA Tesla K40m, 2880 CUDA Cores 745MHz, 3.5 CUDA CC, 12GB GDDR5

Também foram utilizadas as seguintes configurações de software:

- Linux x86_64
- GCC 5.4.0
- Python 3.4.8
- gRPC 1.12.0
- Halide 2018/02/15 (linux-64-gcc53-trunk)

- OpenAI Baselines 0.1.5
- OpenAI Gym 0.10.5

Durante a execução de cada experimento, os processos gerados foram associados a um único *socket* do hardware utilizado, através do utilitário de sistema `numactl`, limitando assim o acesso a 8 *cores* e a memória local de cada *socket*. Essa configuração foi feita para garantir que não ocorram acessos não uniforme à memória de outros *sockets*. O Halide também foi configurado com a quantidade de *cores* disponíveis no *socket* para adequar o número de *threads* usadas ao paralelizar trabalhos.

## 6.4 GERAÇÃO DO ESCALONAMENTO HALIDE ATRAVÉS DO AGENTE PPO

Para geração do escalonamento de execução Halide através do agente de aprendizado por reforço PPO, os seguintes parâmetros e configurações do agente foram definidos, conforme tabela 6.1. Essas configurações são utilizadas internamente pelo agente PPO conforme descrito anteriormente na seção 3.7. Os valores utilizados foram baseados nos valores indicados nos trabalhos de Schulman et al. (2017) e Henderson et al. (2017), otimizados empiricamente através de testes realizados com o ambiente de aprendizado por reforço construído neste trabalho.

Tabela 6.1: Parâmetros usados no Agente de Aprendizado por Reforço.

| Parâmetro | Valor |
|---|---|
| Tamanho do Segmento de Treinamento (*horizon*) | 256 |
| Número de Épocas de Otimização (*epochs*) | 4 |
| Tamanho do Lote de Otimização (*batchsize*) | 64 |
| Taxa de Aprendizado do Otimizador Adam (*stepsize*) | $2.5 \times 10^{-3}$ |
| Fator de Desconto de Recompensas Futuras (*gamma*) | 0.99 |
| Parâmetro do Estimador de Vantagem GAE (*lambda*) | 0.95 |
| Limiar da Razão de Probabilidade da Política (*clip*) | 0.2 |
| Coeficiente de Entropia para Estimular Exploração | 0.03 |
| Tipo de *Annealing* da Taxa de Aprendizado e do Limiar | linear |
| Limite de Iterações do Agente (máximo de episódios) | 10000 |
| Número de Atuadores Paralelos do Agente (*actors*) | 1 |
| Tipo de Rede do Agente | MLP/tanh |
| Tamanho da Camada de Rede Oculta (*hid_size*) | 64 |
| Número de Camadas de Rede Ocultas (*hid_layers*) | 2 |

A execução de cada experimento segue o modelo apresentado na figura 5.1, inicializando o ambiente de aprendizado por reforço com o *pipeline* desejado e com as opções de escalonamento para a arquitetura de hardware pretendida. Cada experimento foi repetido 4 vezes, porém usando um *seed* randômico diferente a cada vez. Apesar do aprendizado por reforço usar o conceito de episódio, no qual o ambiente é frequentemente reiniciado para o estado inicial toda vez que atinge um estado terminal, as repetições dos experimentos aumentam a confiabilidade dos resultados. Essa quantidade de quatro repetições fica entre a média usada na literatura de aprendizado por reforço, conforme levantamento realizado por Henderson et al. (2017).

Nos experimentos também foi considerado que quando o ambiente de aprendizado por reforço recebe uma nova ação produzida pelo agente, o cálculo da recompensa retornada utiliza o menor tempo de execução obtido entre 3 execuções sucessivas do escalonamento resultante da última ação, o qual é testado em uma imagem com dimensão de 3848x2568.

## 6.5 COMPARAÇÃO ENTRE MÉTODOS DE GERAÇÃO DO ESCALONAMENTO HALIDE

Os três *pipelines* de processamento de imagens citados acima também foram avaliados quanto ao desempenho apresentado, isto é, tempo de execução, frente a diferentes escalonamentos de execução. Tais escalonamentos foram produzidos utilizando métodos distintos de geração, contemplando as duas arquiteturas de hardware consideradas, CPU e GPU.

Os métodos de geração considerados são os seguintes:

1. **Escalonamento Manual:** Trata-se do escalonamento desenvolvido e otimizado manualmente por programadores Halide experientes para cada *pipeline* de processamento de imagem e arquitetura de hardware, e disponível no repositório da linguagem (Mullapudi et al., 2016). Como os escalonamentos manuais disponíveis são compatíveis com a especificação do hardware utilizado, como por exemplo, o tamanho de instruções de vetorização, não houve necessidade de ajustes nos mesmos. Porém, no caso do *pipeline* de interpolação, o escalonamento disponível para GPU executa o último estágio do *pipeline* diretamente na CPU, visando limitar o uso de memória da GPU. Essa restrição foi retirada do escalonamento, passando então a executar integralmente na GPU, e assim melhorando seu desempenho nessa arquitetura. Esse ajuste foi realizado para possibilitar uma comparação justa com os demais métodos de escalonamento, os quais também não possuem essa restrição.

2. **Escalonamento Automático:** É o escalonamento produzido automaticamente pelo compilador da linguagem Halide através do mecanismo descrito na seção 4.1.3 da revisão bibliográfica. Os parâmetros de geração automática foram definidos conforme a especificação do hardware utilizado. Entretanto, na versão corrente da linguagem Halide, esse mecanismo apenas produz escalonamentos para arquitetura CPU, assim, não foi possível avaliar o escalonamento automático para GPU.

3. **Escalonamento PPO:** Refere-se ao melhor escalonamento gerado pelo agente de aprendizado por reforço PPO, entre as quatro repetições da execução do aprendizado por reforço citadas na seção anterior, para cada *pipeline* de processamento de imagem e arquitetura de hardware pretendida. Assim, foram quatro execuções do agente para a arquitetura CPU e outras quatro para GPU, em cada um dos *pipelines*, das quais o escalonamento que obteve melhor desempenho foi utilizado para comparação com os demais métodos. Contudo, a evolução do agente durante cada execução individualmente também é apresentada nos resultados.

Os escalonamentos referentes a cada um dos métodos de geração foram aplicados aos *pipelines* e executados em um equipamento com as configurações descritas na seção 6.3, sendo mensurado os tempos de execução de acordo com o protocolo representado pela equação 6.1.

$$Tempo\ de\ Execucao = \min_{1..10}(\sum_{1}^{10} exectime(pipeline) \times 0.1) \tag{6.1}$$

Na equação 6.1 são realizadas 10 execuções sucessivas do mesmo *pipeline* e calculado a média simples do tempo de execução transcorrido (*exectime*), depois esse processo é repetido por 10 vezes e então é retornada a menor média obtida. No tempo mensurado está incluído apenas a duração da execução propriamente, ou seja, não inclui o tempo gasto na compilação do *pipeline* após aplicar um escalonamento.

Esse mesmo protocolo de mensuração do tempo de execução foi aplicado a todos os cenários avaliados, sendo posteriormente comparado o desempenho entre os escalonamentos de cada um dos métodos de geração. A comparação foi realizada para cada um dos *pipelines* de processamento de imagem, arquitetura de hardware e imagem de entrada, permitindo identificar em cada caso qual método apresentou melhor desempenho, ou seja, menor tempo de execução.

## 6.6 CONCLUSÃO

Neste capítulo foram apresentados os cenários de teste considerados para avaliação da abordagem desenvolvida, citando os *pipelines* de processamento de imagens utilizados na avaliação, bem como as características das imagens de entrada consideradas. Também foram apresentadas as configurações de hardware e software onde os experimentos foram executados, os valores adotados para cada parâmetro do agente de aprendizado por reforço, e o protocolo utilizado para registro dos resultados usados na comparação entre os diferentes métodos de geração do escalonamento Halide. No próximo capítulo são apresentados os resultados obtidos.

# 7 RESULTADOS

Neste capítulo são apresentados e analisados os resultados obtidos na execução dos cenários mencionados no capítulo anterior, que incluem testes com imagens de tamanhos distintos e com diferentes *pipelines* de processamento de imagens e arquiteturas de hardware, sumarizados a seguir:

- 3 *pipelines* de processamento de imagens: *Blur*, Harris e Interpolação.
- 3 imagens de entrada com diferentes dimensões: Imagem 1 (962x642), Imagem 2 (1924x1284) e Imagem 3 (3848x2568).
- 2 arquiteturas de hardware distintas: CPU (x86) e GPU (NVIDIA).
- 3 métodos de geração do escalonamento para CPU: Manual, Automático e PPO.
- 2 métodos de geração do escalonamento para GPU: Manual e PPO.

Primeiro são apresentados os gráficos da evolução do agente PPO durante a geração do escalonamento Halide usando a técnica de aprendizado por reforço, depois são comparados os resultados obtidos entre os diferentes métodos de geração: manual, automático e PPO.

## 7.1 EVOLUÇÃO DO AGENTE PPO DURANTE GERAÇÃO DO ESCALONAMENTO HALIDE

Nesta seção é apresentado como transcorreu a evolução do agente de aprendizado por reforço PPO referente à geração e otimização do escalonamento de execução de cada *pipeline* avaliado. A evolução aqui se refere ao período em que o agende ficou executando cada experimento, do início ao fim. As informações da evolução são apresentadas usando dois tipos de dados: primeiro o tempo de execução do *pipeline* que o agente está otimizando e depois a recompensa obtida nos episódios completados pelo agente.

A recompensa de um episódio é a soma das recompensas de cada passo executado pelo agente no mesmo episódio, ou seja, os passos a partir do estado inicial até atingir um estado final. Já o tempo de execução do *pipeline* corresponde ao menor valor absoluto, em milissegundos, obtido entre todos os passos do mesmo episódio. Do ponto de vista prático, o tempo de execução do *pipeline* é o objeto principal de interesse, o que de fato se pretende otimizar, porém, a recompensa é o insumo principal que guia a tomada de decisão e o aprendizado do agente, sendo necessário para que este siga um caminho que o conduza à otimização do primeiro, logo ambas as informações são apresentadas juntas para facilitar a compreensão da evolução do agente.

Na figura 7.1 estão os gráficos da evolução do agente referente ao primeiro *pipeline* avaliado, o *Blur*, tanto para arquitetura CPU quanto para GPU. Os gráficos foram gerados usando dados de análise gravados pelo framework OpenAI utilizado neste trabalho.

Tanto a informação do tempo de execução quanto da recompensa, apresentadas de maneira gráfica, representam a média dos últimos 100 episódios concluídos. No caso do tempo de execução, os resultados melhores são aqueles que apresentam valores menores, já no caso da recompensa, os melhores são aqueles que apresentam valores maiores. Além disso, cada gráfico contém quatro linhas distintas, de cores diferentes, que representam repetições dos mesmos experimentos, porém usando *seeds* randômicos diferentes, sendo que em cada execução de um

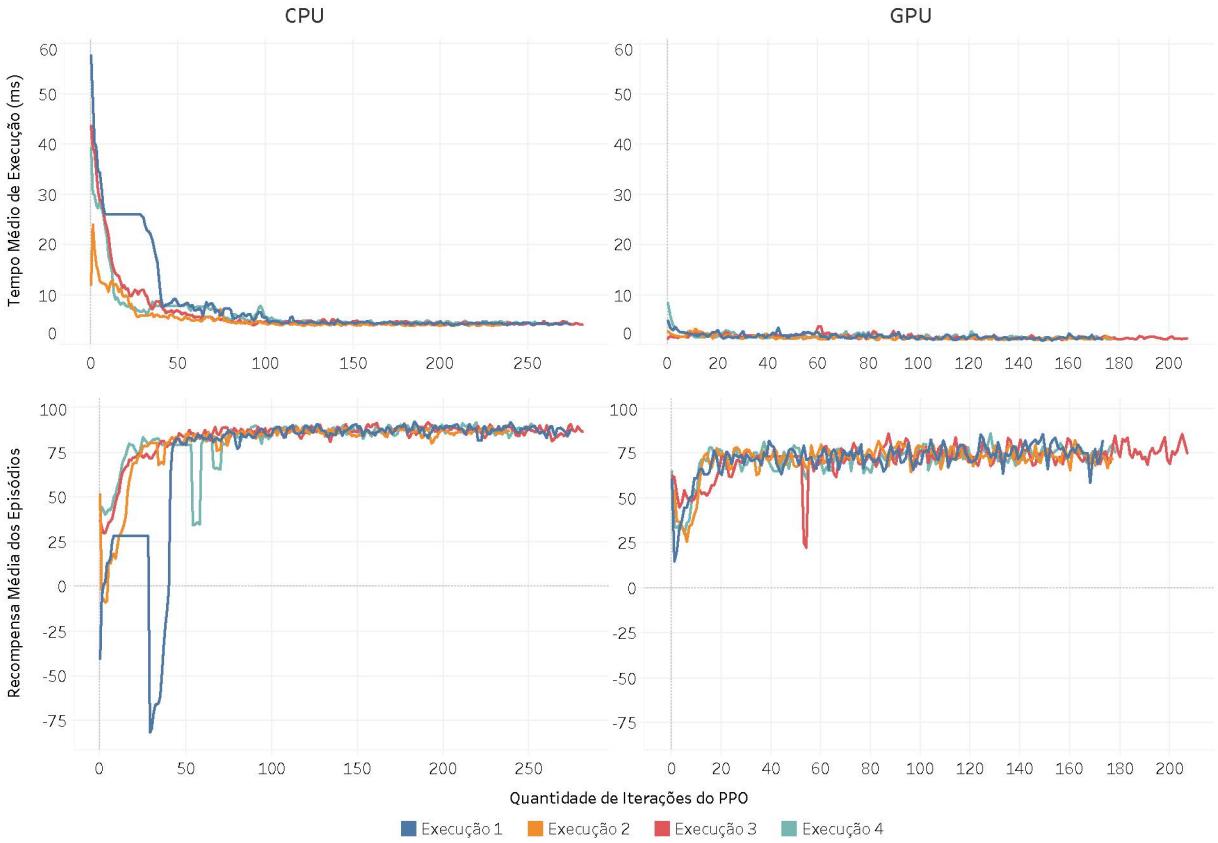



Figura 7.1: Evolução do agente de aprendizado por reforço durante quatro repetições do experimento de otimização do *pipeline Blur* nas arquiteturas CPU (esquerda) e GPU (direita). Cada execução inicia o aprendizado por reforço e termina quando o agente completa o limite de episódios configurados para o experimento, e pode gerar escalonamentos diferentes toda vez. Tempo de execução e recompensa média dos últimos 100 episódios concluídos.

experimento o agente pode produzir escalonamentos diferentes. Apesar do aprendizado por reforço usar o conceito de episódio, no qual o ambiente é frequentemente reiniciado para o estado inicial toda vez que atinge um estado terminal, as repetições dos experimentos aumentam a confiabilidade dos resultados. Essa quantidade de quatro repetições fica entre a média usada na literatura de aprendizado por reforço, conforme levantamento realizado por Henderson et al. (2017).

O eixo horizontal dos gráficos corresponde a quantidade de iterações realizadas pelo agente PPO ao longo da duração do experimento, ou seja, a quantidade de vezes que o agente coletou um segmento de dados de otimização e atualizou as redes neurais internas. O tamanho de cada segmento de dados e a condição de parada do agente, baseada no número de episódios, foi a mesma para todos os experimentos realizados, conforme tabela 6.1, no entanto, pelos gráficos é possível observar que a quantidade de iterações realizadas pelo agente ficou diferente em cada caso. Isso ocorre porque a quantidade de passos realizados pelo agente para completar um episódio pode variar a cada experimento executado, e portanto afeta a quantidade de segmentos de dados que serão gerados.

Na figura 7.2 estão os gráficos de resultados da evolução referente ao *pipeline* Harris. Apesar de ser outro *pipeline*, com tamanho e finalidade diferente do anterior, os resultados indicam evolução similar do agente. No caso do *Blur*, figura 7.1, analisando os gráficos nota-se que a maior redução no tempo de execução do *pipeline* ocorreu nas 100 primeiras iterações do agente, com o respectivo aumento da recompensa, depois continuam ocorrendo oscilações, porém sem grande impacto. Esse comportamento manteve-se em todas as repetições do experimento,

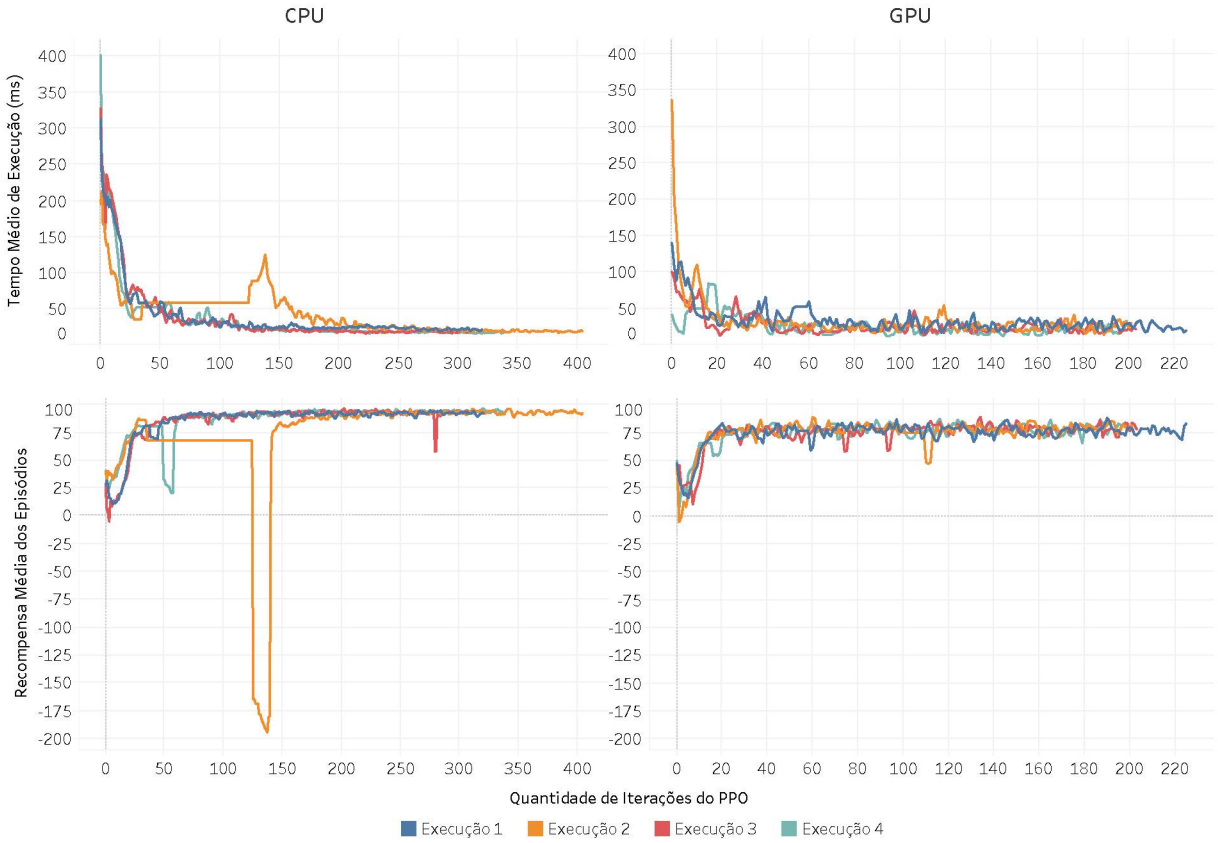



Figura 7.2: Evolução do agente de aprendizado por reforço durante quatro repetições do experimento de otimização do *pipeline* Harris nas arquiteturas CPU (esquerda) e GPU (direita). Cada execução inicia o aprendizado por reforço e termina quando o agente completa o limite de episódios configurados para o experimento, e pode gerar escalonamentos diferentes toda vez. Tempo de execução e recompensa média dos últimos 100 episódios concluídos.

mesmo naquelas com maior oscilação no início. Para o Harris o comportamento foi similar, mas exigiu mais iterações do agente.

Em ambos os resultados anteriores também é possível observar que em certos momentos ao longo de alguns experimentos houve uma queda acentuada na recompensa dos episódios. Isso ocorre quando o agente precisa de várias iterações para concluir um determinado episódio, e dentro deste episódio muitas ações geraram recompensa negativa, conforme definido anteriormente na equação 5.2.

Na sequência, na figura 7.3, estão os resultados referente ao *pipeline* de Interpolação.

Em todos os experimentos, o tempo de execução é significantemente diferente entre os *pipelines*, mas nota-se que a recompensa tende a apresentar valores similares à medida em que avançam as iterações do agente. Isso ocorre devido ao fator de ajuste de escala usado pelo ambiente no cálculo da recompensa, conforme equação 5.3.

Outro ponto a observar nos gráficos de recompensa, é que no início do processo de otimização, nas primeiras iterações, a recompensa tende a piorar, podendo atingir valores negativos. Isso ocorre porque o agente parte de um estado inicial e começa a explorar as opções de diretivas disponíveis, através das interações com o ambiente, gerando normalmente muitos escalonamentos iniciais inválidos, que causam erro na compilação ou execução, e assim produzem recompensas negativas, as quais serão acumuladas dentro do mesmo episódio. À medida que a quantidade de iterações aumenta, a experiência acumulada do agente faz com que este passe a escolher opções melhores, que produzem resultados positivos, convergindo gradualmente para uma solução válida e otimizada. Contudo, ao longo de todo o processo o agente continua

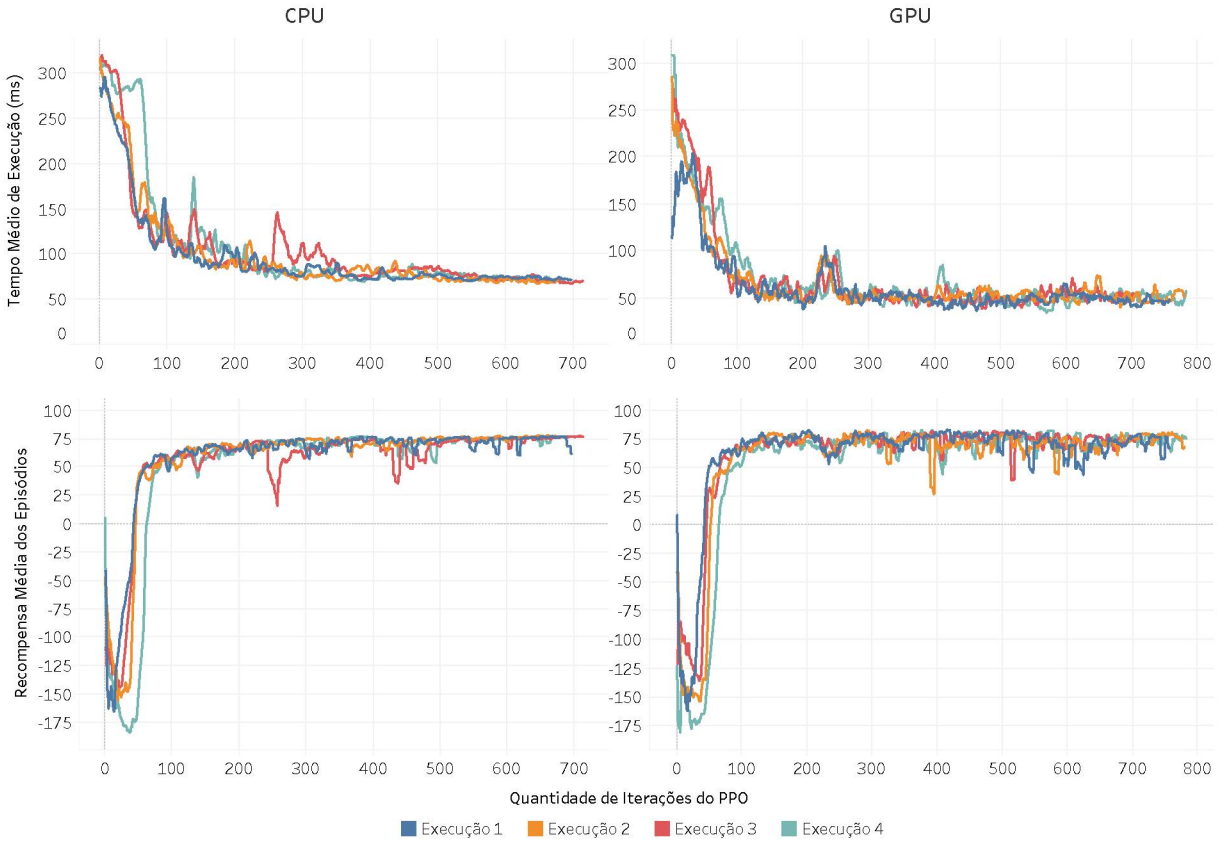



Figura 7.3: Evolução do agente de aprendizado por reforço durante quatro repetições do experimento de otimização do *pipeline* de Interpolação nas arquiteturas CPU (esquerda) e GPU (direita). Cada execução inicia o aprendizado por reforço e termina quando o agente completa o limite de episódios configurados para o experimento, e pode gerar escalonamentos diferentes toda vez. Tempo de execução e recompensa média dos últimos 100 episódios concluídos.

explorando opções distintas de diretivas, que ocasionalmente geram escalonamentos inválidos, porém, na média o retorno obtido nos episódios tende a ser positivo.

Com relação ao tempo de duração dos experimento, ou seja, o período em que o agente ficou rodando cada caso, até concluir o número de episódios configurados como condição de parada, foi de aproximadamente um dia para os cenários do *pipeline Blur*, um dia e meio para o Harris, e sete dias para o caso da Interpolação. O melhor resultado de cada experimento foi obtido em algum momento durante esse período de duração, mas como pode ser observado nos gráficos acima, alguns casos já estavam estabilizando o resultado antes do final do experimento.

O principal fator que afeta a duração do experimento, contudo, não é o tempo de execução do *pipeline* propriamente, nem o tempo gasto nas atualizações das redes neurais do agente, mas sim o tempo consumido pelo Halide para compilar cada novo escalonamento gerado, o que ocorre a cada passo do agente ao acrescentar e testar uma nova diretiva de escalonamento. O tempo de compilação é especialmente afetado em *pipelines* maiores, como é o caso do *pipeline* de Interpolação. Dessa maneira é plausível considerar que quanto maior for o tamanho do *pipeline*, também será maior o tempo necessário para rodar o agente de aprendizado por reforço para a mesma quantidade de episódios.

A utilização de um agente já treinado com um *pipeline* para otimizar o escalonamento de outro *pipeline* não é viável diretamente, pois a quantidade e os códigos de mapeamento das ações e estados do ambiente são específicos para cada *pipeline* e afetam o tamanho das camadas de entrada e saída das redes neurais do agente, porém, as camadas ocultas são as mesmas. Dessa maneira, uma possibilidade que pode ser explorada futuramente é a utilização de

técnicas de *Transfer Learning* (Pan et al., 2010), as quais abrem caminho para o aproveitamento do treinamento de partes específicas das redes neurais, o que poderia eventualmente reduzir a duração dos experimentos e aumentar a generalização da solução desenvolvida.

## 7.2 COMPARAÇÃO ENTRE OS MÉTODOS DE GERAÇÃO DO ESCALONAMENTO HALIDE

A seguir são apresentados os resultados comparativos obtidos na execução dos escalonamentos produzidos entre os diferentes métodos de geração do escalonamento, que inclui a geração manual por programadores Halide, o método automático disponível na linguagem Halide, e o método desenvolvido neste trabalho que utiliza aprendizado por reforço através de um agente PPO. Os resultados permitem avaliar a eficácia do agente PPO comparativamente aos demais métodos, referente a otimização dos cenários de teste considerados. No caso do método que usa o agente PPO, conforme relatado acima, houve repetições dos mesmos experimentos usando esse mesmo método. Nesta seção os resultados reportados utilizam o melhor escalonamento PPO encontrado entre as repetições da execução do agente.

No gráfico da figura 7.4 constam os valores de desempenho relativo de cada método de geração em relação ao melhor resultado obtido em cada cenário, adotado como referência, separadamente para cada imagem de entrada e arquitetura de hardware. O desempenho relativo é baseado na razão entre os tempos de execução dos escalonamentos comparados. O melhor apresenta valor 1.0 e os demais apresentam valores inferiores.

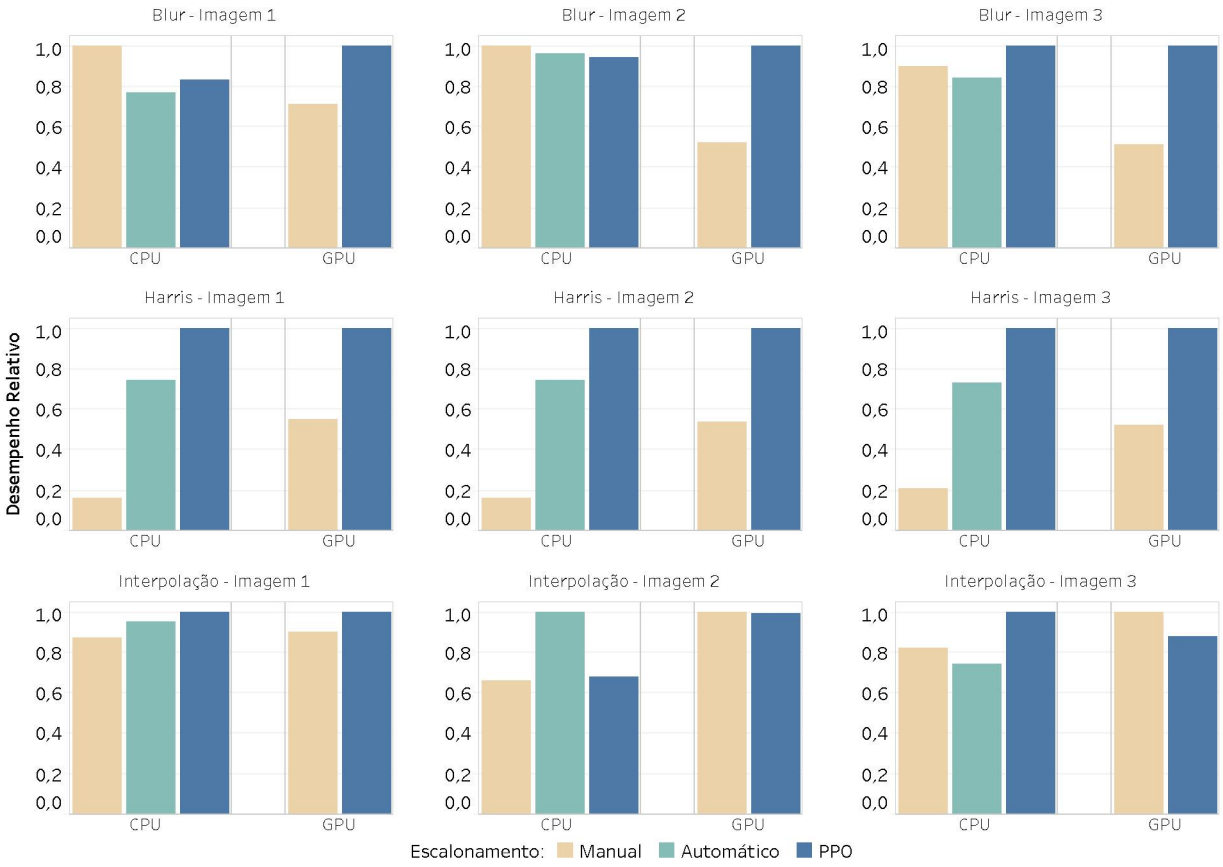



Figura 7.4: Desempenho relativo ao melhor resultado por arquitetura e método de escalonamento. Quanto maior, melhor. O desempenho relativo é baseado na razão entre os tempos de execução dos escalonamentos comparados. O melhor apresenta valor 1.0 e os demais apresentam valores inferiores.

Pelo gráfico é possível observar que há casos em que o método PPO foi melhor e outros em que não atingiu o melhor resultado, em relação aos outros dois métodos. Na comparação com o método automático, o qual atualmente está disponível apenas para arquitetura CPU, o PPO se destacou mais significativamente no *pipeline* Harris, enquanto que no Blur ficou similar e na Interpolação ficou inferior no cenário com a imagem 2. No caso da arquitetura GPU, comparado apenas com o método manual, o PPO se destacou em ambos os *pipelines* Blur e Harris, mas oscilou na Interpolação, sendo melhor para imagem 1 e inferior na imagem 3.

Na tabela 7.1 constam os valores absolutos do tempo de execução em cada cenário avaliado, e também o *slowdown*[1] de cada caso relativo ao melhor resultado dentro da mesma arquitetura de hardware, *pipeline* e imagem de entrada. Na tabela, os cenários que obtiveram melhor resultado, isto é, menor tempo de execução, estão destacados em negrito, já aqueles com resultado inferior apresentam ao lado o *slowdown* em relação ao melhor caso do cenário correspondente. De maneira geral, a tabela apresenta as mesmas informações do gráfico da figura 7.4, porém mais detalhadamente, listando os valores efetivos obtidos durante os experimentos.

Tabela 7.1: Tempo de execução de cada *pipeline* por arquitetura e método de escalonamento. Valor absoluto (ms) e *slowdown*[1] (×) relativo ao melhor resultado obtido na mesma arquitetura, *pipeline* e imagem de entrada.

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

Além dos testes de desempenho, um teste adicional de verificação foi realizado, comparando se os resultados dos *pipelines* de processamento de imagem eram os mesmos entre os diferentes escalonamentos, manual, automático, e PPO. No teste não foi identificado diferenças na imagem de saída, resultado este já esperado devido ao desacoplamento entre a lógica do algoritmo e o escalonamento de execução, garantido pela própria linguagem Halide.

## 7.3 CONCLUSÃO

Neste capítulo foi apresentado informações sobre a evolução do agente de aprendizado por reforço ao longo do processo de geração e otimização do escalonamento Halide para três *pipelines* de processamento de imagens e duas arquiteturas de hardware distintas. Depois os escalonamentos produzidos pelo agente foram comparados com versões geradas a partir de outros métodos, comparando o desempenho de cada versão ao processar imagens de diferentes tamanhos. Os resultados apresentados indicam que na maioria dos cenários testados o agente foi capaz de convergir para escalonamentos com desempenho competitivo aos demais métodos avaliados.

[1] O *slowdown* indica quantas vezes mais lento foi um resultado em relação a outro.

# 8 CONCLUSÃO

Neste trabalho foi apresentado uma abordagem de otimização do escalonamento Halide através da aplicação de uma técnica de aprendizado por reforço. A implementação da abordagem consiste em dois módulos, um módulo Python para a heurística de aprendizagem e otimização, e outro módulo C++ para interação com a linguagem Halide e os *pipelines* de processamento de imagens. No módulo Python foi utilizado um agente PPO de aprendizado por reforço, através de uma implementação disponível na biblioteca OpenAI Baselines, e construído um ambiente de aprendizado por reforço que respeita as interfaces padronizadas disponibilizadas pela biblioteca OpenAI Gym, e portando compatível com outros frameworks e agentes de aprendizado por reforço que suportam as mesmas interfaces. Um dos pontos importantes dentro da definição do ambiente é o cálculo da recompensa, a qual guia o aprendizado do agente. No módulo C++ foi desenvolvido um mecanismo de interação entre a linguagem Halide e o ambiente de aprendizado por reforço, por meio do mapeamento das diretivas de escalonamento para identificadores numéricos, possibilitando a aplicação do aprendizado por reforço.

O mapeamento das diretivas de escalonamento é uma atividade realizada manualmente pelo programador Halide para cada *pipeline* que se deseja otimizar, necessária para elencar e construir o espaço de ações que serão exploradas pelo agente de aprendizado por reforço para gerar escalonamentos para o *pipeline* correspondente. Um ponto forte nesse aspecto é que permite ao programador Halide utilizar sua experiência profissional e conhecimento do *pipeline* em questão para mapear apenas opções mais plausíveis, reduzindo assim o espaço que o agente de aprendizado por reforço precisará explorar. Por outro lado, como requer uma intervenção do programador, o mecanismo desenvolvido não é totalmente automatizado, se posicionando como um modelo intermediário entre o desenvolvimento manual do escalonamento e a opção do escalonamento automático disponível na linguagem. Entretanto, a abordagem apresentada é independente da arquitetura de hardware utilizada. No presente trabalho a abordagem foi avaliada em duas arquiteturas, CPU e GPU, mas também permite sua utilização em outras arquiteturas suportadas pela linguagem Halide.

Os resultados obtidos mostram que o agente de aprendizado por reforço foi capaz de partir de um estado inicial e convergir para bons escalonamentos de execução Halide nos *pipelines* avaliados, em ambas as arquiteturas, ainda que não atingiu o melhor resultado em alguns dos cenários considerados, comparativamente aos demais métodos, manual e automático. Esses resultados também indicam que o ambiente construído, bem como o método de cálculo da recompensa, além da representação dos estados e ações, foram efetivos em representar o problema de forma compatível com a técnica de aprendizado por reforço. Contudo, um ponto que requer atenção, é com relação ao considerável tempo de duração dos experimentos, que pode eventualmente inviabilizar a aplicação para grandes *pipelines*, porém, justamente neste aspecto, a abordagem de aprendizado de máquina empregada neste trabalho pode abrir caminho para o desenvolvimento de reaproveitamento de informações de treinamentos anteriores, conforme sugerido a seguir como trabalhos futuros.

De maneira geral, dentro dos cenários avaliados, a abordagem do aprendizado por reforço por meio do agente PPO se mostrou promissora, contudo, é apenas um trabalho preliminar, mais cenários de teste e um conjunto mais amplo de *pipelines* precisam ser avaliados para se ter um indicador mais abrangente e fidedigno.

## 8.1 TRABALHOS FUTUROS

A partir do desenvolvimento e dos testes realizados, alguns aprimoramentos para a solução elaborada e novos cenários de testes foram identificados e são sugeridos como trabalhos futuros, listados a seguir:

- Gerar automaticamente o mapeamento das opções de escalonamento mais plausíveis com base nas informações do *pipeline*. O escalonador automático da linguagem Halide poderia ser adaptado para usar as informações extraídas através do compilador da linguagem e gerar o mapeamento das opções que seriam então exploradas pelo agente de aprendizado por reforço.

- Permitir o mapeamento de grupos de diretivas que, devido a dependência entre elas, precisam estar juntas para serem uma operação válida. No atual mecanismo de mapeamento desenvolvido cada opção corresponde a uma única diretiva e o agente precisa encontrar combinações válidas quando existem dependências entre diretivas. O conceito de grupo viabilizaria que uma única opção mapeada represente uma sequência de duas ou mais diretivas.

- Utilizar técnicas de *Transfer Learning* (Pan et al., 2010) para tentar reaproveitar o conhecimento adquirido pelo agente durante a otimização do escalonamento de diferentes *pipelines*, e assim acelerar o processo de otimização de novos *pipelines*.

- Incluir dados adicionais na representação do estado do ambiente de aprendizado por reforço, com informações coletadas sobre a utilização do hardware durante a execução dos *pipelines*, como por exemplo, percentual de utilização dos processadores e da memória, quantidade de leituras e escritas na memória, *cache misses*, etc.

- Avaliar outros agentes de aprendizado por reforço, ou até mesmo outras técnicas de otimização, usando a mesma representação do problema e ambiente desenvolvido neste projeto. A partir dos mesmos cenários de teste, os resultados poderiam serem comparados para identificar quais agentes produzem melhores resultados em cada situação. Além disso, novos cenários de teste e *pipelines* também podem ser abordados.

- Por fim, como uma proposta de generalização da abordagem utilizada, seria implementar um mecanismo similar de mapeamento e representação do problema em outras linguagens DSL, como por exemplo, linguagem GraphIt (Zhang et al., 2018), e então avaliar a aplicação do aprendizado por reforço na otimização de programas em outros domínios.

# REFERÊNCIAS


Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M. e Amarasinghe, S. (2014a). Opentuner: An extensible framework for program autotuning. Em *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, páginas 303–316, Edmonton, Canada. ACM.

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M. e Amarasinghe, S. (2014b). Opentuner: An extensible framework for program autotuning. `http://opentuner.org`. Acessado em 07/01/2018.

Bellman, R. E. (1957). *Dynamic programming*. Princeton University Press, Princeton, NJ, USA.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. e Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C. e Krishnamurthy, A. (2018). Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166*.

Costa, M. L. D. (2017). *Uma abordagem utilizando aprendizagem por reforço hierárquica e computação paralela para o problema dos k-servos*. Tese de doutorado, Pós-graduação em Ciência e Engenharia de Petróleo - Universidade Federal do Rio Grande do Norte, Natal - RN.

Denniston, T. (2016). Terapixel image processing and simulation with distributed halide. Dissertação de Mestrado, Massachusetts Institute of Technology, Boston - Massachusetts, USA.

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S. e Wu, Y. (2017). Openai baselines. `https://github.com/openai/baselines`. Acessado em 05/09/2018.

Durand, F. (2015a). Halide scheduling. `http://stellar.mit.edu/S/course/6/sp15/6.815/courseMaterial/topics/topic2/lectureNotes/15_Halide2/15_Halide2.pdf`. MIT Course 6.815/6.865 - Digital & Computational Photography. Acessado em 18/04/2017.

Durand, F. (2015b). High-performance image processing. `http://stellar.mit.edu/S/course/6/sp15/6.815/courseMaterial/topics/topic2/lectureNotes/14_Halide_print/14_Halide_print.pdf`. MIT Course 6.815/6.865 - Digital & Computational Photography. Acessado em 18/04/2017.

Durand, F. (2015c). High-performance image processing last lecture on halide. `http://stellar.mit.edu/S/course/6/sp15/6.815/courseMaterial/topics/topic2/lectureNotes/17_Halide4/17_Halide4.pdf`. MIT Course 6.815/6.865 - Digital & Computational Photography. Acessado em 18/04/2017.

Falch, T. L. e Elster, A. C. (2015). Machine learning based auto-tuning for enhanced opencl performance portability. Em *Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, páginas 1231–1240, Hyderabad, Índia. IEEE.

Fowler, M. e Parsons, R. (2010). *Domain-Specific Languages*. Addison-Wesley Professional.

Harris, C. e Stephens, M. (1988). A combined corner and edge detector. Em *Alvey Vision Conference*, volume 15, páginas 10–5244. Manchester, UK.

Hegarty, J., Brunhaver, J., DeVito, Z., Ragan-Kelley, J., Cohen, N., Bell, S., Vasilyev, A., Horowitz, M. e Hanrahan, P. (2014). Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144–1.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D. e Meger, D. (2017). Deep reinforcement learning that matters. *arXiv preprint arXiv:1709.06560*.

Huang, S. (2018). Introduction to various reinforcement learning algorithms, part i: Q-learning, SARSA, DQN, DDPG. `https://towardsdatascience.com/72a5e0cb6287`. Acessado em 17/09/2018.

Hudak, P. (1997). Domain-specific languages. *Handbook of programming languages*, 3(39-60):21.

Juliani, A. (2016a). Simple reinforcement learning with tensorflow part 0: Q-learning with tables and neural networks. `https://medium.com/emergent-future/d195264329d0`. Acessado em 08/11/2017.

Juliani, A. (2016b). Simple reinforcement learning with tensorflow part 7: Action-selection strategies for exploration. `https://medium.com/emergent-future/d3a97b7cceaf`. Acessado em 07/11/2017.

Juliani, A. (2016c). Simple reinforcement learning with tensorflow part 8: Asynchronous actor-critic agents (A3C). `https://medium.com/emergent-future/c88f72a5e9f2`. Acessado em 19/09/2018.

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

Lattner, C. (2018). LLVM design & overview: Intro to LLVM. `https://llvm.org/docs`. Acessado em 23/10/2018.

Lattner, C. e Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. Em *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, página 75. IEEE Computer Society.

Li, T.-M., Gharbi, M., Adams, A., Durand, F. e Ragan-Kelley, J. (2018). Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics (TOG)*, 37(4):139.

Li, Y. (2017). Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*.

Li, Y., Chang, K., Bel, O., Miller, E. L. e Long, D. D. (2017). CAPES: unsupervised storage performance tuning using neural network-based deep reinforcement learning. Em *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, página 42. ACM.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. e Wierstra, D. (2016). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lopes, R. A. S. e Braga, V. G. D. M. (2017). Um sistema para o aprendizado automático de jogos eletônicos baseado em redes neurais e q-learning usando interface natural. Bacharelado em Ciência da Computação - Universidade de Brasília, Brasília - DF.

Matiisen, T. (2015). Demystifying deep reinforcement learning. `http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning`. Acessado em 08/11/2017.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. e Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G. et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

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

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

Nelder, J. A. e Mead, R. (1965). A simplex method for function minimization. *The computer journal*, 7(4):308–313.

Ottoni, A. L. C., Oliveira, M. S., Nepomuceno, E. G. e 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 Transactions on knowledge and data engineering*, 22(10):1345–1359.

Ragan-Kelley, J. e Adams, A. (2012). Halide: A language for image processing. `http://halide-lang.org`. Acessado em 08/01/2018.

Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S. e 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., Adams, A. e Sharlet, D. (2015). An introduction to halide. `http://halide-lang.org/assets/lectures/Halide_CVPR_intro.pdf`. IEEE Conference on Computer Vision and Pattern Recognition. Acessado em 20/04/2017.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F. e 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*. Tese de doutorado, Massachusetts Institute of Technology.

Ravishankar, M., Holewinski, J. e Grover, V. (2015). Forma: A DSL for image processing applications to target GPUs and multi-core CPUs. Em *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, páginas 109–120. ACM.

Sarkar, S. e Alavani, G. (2018). How easy it is to write software for heterogeneous systems? *ACM SIGSOFT Software Engineering Notes*, 42(4):1–7.

Schulman, J., Levine, S., Abbeel, P., Jordan, M. e Moritz, P. (2015a). Trust region policy optimization. Em *Proceedings of the 31st International Conference on Machine Learning (ICML-15)*, páginas 1889–1897, Lille, France.

Schulman, J., Moritz, P., Levine, S., Jordan, M. e Abbeel, P. (2015b). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

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

Seno, T. (2017). Welcome to deep reinforcement learning part 1: DQN. `https://towardsdatascience.com/c3cab4d41b6b`. Acessado em 08/11/2017.

Shacham, O. e Reynders, M. (2017). Pixel visual core: image processing and machine learning on pixel 2. `https://www.blog.google/products/pixel/pixel-visual-core-image-processing-and-machine-learning-pixel-2`. Acessado em 29/03/2019.

Sharlet, D. (2015). Recursive filtering in halide. `http://halide-lang.org/assets/lectures/CVPR_Halide_RecursiveFiltering.pdf`. IEEE Conference on Computer Vision and Pattern Recognition. Acessado em 20/04/2017.

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

Silva, R. E. G. F. d. (2018). Artificial intelligence techniques applied for the predictive control of stationary storage. Dissertação de Mestrado, Faculdade de Engenharia - Universidade do Porto, Porto - Portugal.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D. e Riedmiller, M. (2014). Deterministic policy gradient algorithms. Em *International Conference on Machine Learning*.

Stringhini, D., Gonçalves, R. A. e Goldman, A. (2012). Introdução à computação heterogênea. Em *XXXI Jornadas de Atualização em Informática (JAI)*, páginas 262–309, Curitiba - PR.

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

Torczon, V. J. (1989). *Multidirectional search: a direct search algorithm for parallel machines*. Tese de doutorado, Rice University.

Van Deursen, A., Klint, P. e Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36.

Watkins, C. J. e Dayan, P. (1992). Technical note q-learning. *Machine Learning*, 8(3-4):279–292.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. Tese de doutorado, King's College, Cambridge.

Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J. e Amarasinghe, S. (2018). Graphit: a high-performance graph DSL. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):121.

# APÊNDICE A – CÓDIGO HALIDE DOS *PIPELINES* DE PROCESSAMENTO DE IMAGENS

Implementação dos *pipelines* de processamento de imagens utilizados neste trabalho como entrada para o aprendizado por reforço. Código fonte conforme disponível no repositório da linguagem Halide (`https://github.com/halide/Halide`).

## A.1 *BLUR*

```
Func blur(Buffer<float> in) {
    Func in_b = BoundaryConditions::repeat_edge(in);

    Var x("x"), y("y");

    Func blur_x("blur_x");
    blur_x(x, y) = (in_b(x - 1, y) + in_b(x, y) + in_b(x + 1, y)) / 3;

    Func blur_y("blur_y");
    blur_y(x, y) = (blur_x(x, y - 1) + blur_x(x, y) + blur_x(x, y + 1)) / 3;

    return blur_y;
}
```

## A.2 HARRIS

```
Expr sum3x3(Func f, Var x, Var y) {
    return f(x - 1, y - 1) + f(x - 1, y) + f(x - 1, y + 1) +
           f(x, y - 1) + f(x, y) + f(x, y + 1) +
           f(x + 1, y - 1) + f(x + 1, y) + f(x + 1, y + 1);
}

Func harris(Buffer<float> in) {
    Func in_b = BoundaryConditions::repeat_edge(in);

    Var x("x"), y("y");

    Func gray("gray");
    gray(x, y) = 0.299f * in_b(x, y, 0) + 0.587f * in_b(x, y, 1) +
           0.114f * in_b(x, y, 2);

    Func Iy("Iy");
    Iy(x, y) = gray(x - 1, y - 1)*(-1.0f / 12) +
           gray(x - 1, y + 1)*(1.0f / 12) +
           gray(x, y - 1)*(-2.0f / 12) + gray(x, y + 1)*(2.0f / 12) +
           gray(x + 1, y - 1)*(-1.0f / 12) + gray(x + 1, y + 1)*(1.0f / 12);

    Func Ix("Ix");
    Ix(x, y) = gray(x - 1, y - 1)*(-1.0f / 12) +
           gray(x + 1, y - 1)*(1.0f / 12) +
           gray(x - 1, y)*(-2.0f / 12) + gray(x + 1, y)*(2.0f / 12) +
           gray(x - 1, y + 1)*(-1.0f / 12) + gray(x + 1, y + 1)*(1.0f / 12);

    Func Ixx("Ixx");
```

```
    Ixx(x, y) = Ix(x, y) * Ix(x, y);

    Func Iyy("Iyy");
    Iyy(x, y) = Iy(x, y) * Iy(x, y);

    Func Ixy("Ixy");
    Ixy(x, y) = Ix(x, y) * Iy(x, y);

    Func Sxx("Sxx");
    Sxx(x, y) = sum3x3(Ixx, x, y);

    Func Syy("Syy");
    Syy(x, y) = sum3x3(Iyy, x, y);

    Func Sxy("Sxy");
    Sxy(x, y) = sum3x3(Ixy, x, y);

    Func det("det");
    det(x, y) = Sxx(x, y) * Syy(x, y) - Sxy(x, y) * Sxy(x, y);

    Func trace("trace");
    trace(x, y) = Sxx(x, y) + Syy(x, y);

    Func harris("harris");
    harris(x, y) = det(x, y) - 0.04f * trace(x, y) * trace(x, y);

    Func shifted("shifted");
    shifted(x, y) = harris(x + 2, y + 2);

    return shifted;
}
```

## A.3 INTERPOLAÇÃO

```
Func interp(Buffer<float> in) {
    Func in_b = BoundaryConditions::repeat_edge(in);

    const int levels = 10;

    Func downsampled[levels];
    Func downx[levels];
    Func interpolated[levels];
    Func upsampled[levels];
    Func upsampledx[levels];

    Var x("x"), y("y"), c("c");

    downsampled[0](x, y, c) = in_b(x, y, c) * in_b(x, y, 3);

    for (int l = 1; l < levels; ++l) {
        Func prev = downsampled[l - 1];

        if (l == 4) {
            Expr w = in.width() / (1 << l);
            Expr h = in.height() / (1 << l);
            prev = lambda(x, y, c, prev(clamp(x, 0, w), clamp(y, 0, h), c));
        }
```

```
        downx[l](x, y, c) = (prev(x * 2 - 1, y, c) +
                2.0f * prev(x * 2, y, c) +
                prev(x * 2 + 1, y, c)) * 0.25f;
        downsampled[l](x, y, c) = (downx[l](x, y * 2 - 1, c) +
                2.0f * downx[l](x, y * 2, c) +
                downx[l](x, y * 2 + 1, c)) * 0.25f;
    }
    interpolated[levels - 1](x, y, c) = downsampled[levels - 1](x, y, c);
    for (int l = levels - 2; l >= 0; --l) {
        upsampledx[l](x, y, c) = (interpolated[l + 1](x / 2, y, c) +
                interpolated[l + 1]((x + 1) / 2, y, c)) / 2.0f;
        upsampled[l](x, y, c) = (upsampledx[l](x, y / 2, c) +
                upsampledx[l](x, (y + 1) / 2, c)) / 2.0f;
        interpolated[l](x, y, c) = downsampled[l](x, y, c) +
                (1.0f - downsampled[l](x, y, 3)) * upsampled[l](x, y, c);
    }

    Func normalize("normalize");
    normalize(x, y, c) = interpolated[0](x, y, c) / interpolated[0](x, y, 3);

    // Default needed to compile
    for (int l = 1; l < levels-1; ++l) {
        downsampled[l].compute_root();
        interpolated[l].compute_root();
    }

    return normalize;
}
```

# APÊNDICE B – MAPEAMENTO DAS OPÇÕES DE ESCALONAMENTO PARA ARQUITETURA CPU

Código fonte de mapeamento das opções de escalonamento usadas como entrada para o aprendizado por reforço, conforme cada *pipeline* de processamento de imagem avaliado, referente à arquitetura CPU.

## B.1 *BLUR*

```
void schedule_mapping(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);
}
```

## B.2 HARRIS

```
void schedule_mapping(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(shifted)
        .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(gray)
        .store_at({shifted}, {y})
        .compute_at({shifted}, {x, yi})
        .unroll({x}, unroll_factor)
        .vectorize({x}, vecto_factor);
    sm.map(Ix)
        .store_at({shifted}, {y})
```

```
        .compute_at({shifted}, {x, yi})
        .unroll({x}, unroll_factor)
        .vectorize({x}, vecto_factor);
    sm.map(Iy)
        .store_at({shifted}, {y})
        .compute_at({shifted}, {x, yi})
        .unroll({x}, unroll_factor)
        .vectorize({x}, vecto_factor);
}
```

## B.3 INTERPOLAÇÃO

```
void schedule_mapping(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};

    for (int l = 1; l < levels-1; ++l) {
        sm.map(downsampled[l])
            .tile({x}, {y}, {xi}, {yi}, split_factor, split_factor)
            .split({y}, {yi}, split_factor)
            .parallel({y, c})
            .unroll({xi, x, c}, unroll_factor)
            .vectorize({xi, x}, vecto_factor);
        sm.map(downx[l])
            .store_at({downsampled[l]}, {y})
            .compute_at({downsampled[l]}, {x, yi})
            .unroll({x}, unroll_factor)
            .vectorize({x}, vecto_factor);
        sm.map(interpolated[l])
            .tile({x}, {y}, {xi}, {yi}, split_factor, split_factor)
            .split({y}, {yi}, split_factor)
            .parallel({y, c})
            .unroll({xi, x, c}, unroll_factor)
            .vectorize({xi, x}, vecto_factor);
        sm.map(interpolated[l])
            .store_at({normalize}, {y})
            .compute_at({normalize}, {x, yi});
        sm.map(upsampledx[l])
            .store_at({normalize}, {y})
            .compute_at({normalize}, {x, yi})
            .unroll({x}, unroll_factor)
            .vectorize({x}, vecto_factor);
    }
    sm.map(normalize)
        .bound({c}, {0}, {3})
        .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, c})
        .unroll({xi, x, c}, unroll_factor)
        .vectorize({xi, x}, vecto_factor);
}
```

# APÊNDICE C – MAPEAMENTO DAS OPÇÕES DE ESCALONAMENTO PARA ARQUITETURA GPU

Código fonte de mapeamento das opções de escalonamento usadas como entrada para o aprendizado por reforço, conforme cada *pipeline* de processamento de imagem avaliado, referente à arquitetura GPU.

## C.1 *BLUR*

```
void schedule_mapping(HalideScheduleMapper &sm) {
    vector<Expr> split_factor = {8, 16, 32, 64, 128, 256, 512};
    vector<Expr> unroll_factor = {2, 3, 4};

    sm.map(blur_y)
        .bound({y}, {0}, {input.height()})
        .bound({x}, {0}, {input.width()})
        .compute_root()
        .gpu_tile({x}, {y}, {xi}, {yi}, split_factor, split_factor)
        .unroll({xi, yi}, unroll_factor);
    sm.map(blur_x)
        .store_at({blur_y}, {x, y})
        .compute_at({blur_y}, {x, y})
        .gpu_threads({x}, {y})
        .unroll({x, y}, unroll_factor);
}
```

## C.2 HARRIS

```
void schedule_mapping(HalideScheduleMapper &sm) {
    vector<Expr> split_factor = {8, 16, 32, 64, 128, 256, 512};
    vector<Expr> unroll_factor = {2, 3, 4};

    sm.map(shifted)
        .bound({y}, {0}, {input.height()})
        .bound({x}, {0}, {input.width()})
        .compute_root()
        .gpu_tile({x}, {y}, {xi}, {yi}, split_factor, split_factor)
        .unroll({xi, yi}, unroll_factor);
    sm.map(gray)
        .store_at({shifted}, {x, y})
        .compute_at({shifted}, {x, y})
        .gpu_threads({x}, {y})
        .unroll({x, y}, unroll_factor);
    sm.map(Ix)
        .store_at({shifted}, {x, y})
        .compute_at({shifted}, {x, y})
        .gpu_threads({x}, {y})
        .unroll({x, y}, unroll_factor);
    sm.map(Iy)
        .store_at({shifted}, {x, y})
        .compute_at({shifted}, {x, y})
        .gpu_threads({x}, {y})
        .unroll({x, y}, unroll_factor);
```

```
}
```

## C.3 INTERPOLAÇÃO

```
void schedule_mapping(HalideScheduleMapper &sm) {
    vector<Expr> split_factor = {8, 16, 32, 64, 128, 256, 512};
    vector<Expr> unroll_factor = {2, 3, 4};

    for (int l = 1; l < levels-1; ++l) {
        sm.map(downsampled[l])
            .gpu_tile({x}, {y}, {c}, {xi}, {yi}, {ci},
                split_factor, split_factor, {4})
            .unroll({xi, yi, ci}, unroll_factor);
        sm.map(downx[l])
            .store_at({downsampled[l]}, {x, y, c})
            .compute_at({downsampled[l]}, {x, y, c})
            .gpu_threads({x}, {y}, {c})
            .unroll({x, y, c}, unroll_factor);
        sm.map(interpolated[l])
            .gpu_tile({x}, {y}, {c}, {xi}, {yi}, {ci},
                split_factor, split_factor, {4})
            .unroll({xi, yi, ci}, unroll_factor);
        sm.map(interpolated[l])
            .store_at({normalize}, {x, y, c})
            .compute_at({normalize}, {x, y, c})
            .gpu_threads({x}, {y}, {c})
            .unroll({x, y, c}, unroll_factor);
        sm.map(upsampledx[l])
            .store_at({normalize}, {x, y, c})
            .compute_at({normalize}, {x, y, c})
            .gpu_threads({x}, {y}, {c})
            .unroll({x, y, c}, unroll_factor);
    }
    sm.map(normalize)
        .bound({c}, {0}, {3})
        .bound({y}, {0}, {input.height()})
        .bound({x}, {0}, {input.width()})
        .compute_root()
        .gpu_tile({x}, {y}, {c}, {xi}, {yi}, {ci},
            split_factor, split_factor, {3})
        .unroll({xi, yi, ci}, unroll_factor);
}
```