Universidade Federal de Pernambuco
Centro de Informática

Graduação em Engenharia da Computação

Hardware Acelerador da Técnica de Reconhecimento de Caracteres em Imagens de Cenas Naturais

Luiz Antonio de Oliveira Junior

Trabalho de Graduação

Recife
07 de julho de 2017

Universidade Federal de Pernambuco
Centro de Informática

Luiz Antonio de Oliveira Junior

Hardware Acelerador da Técnica de Reconhecimento de Caracteres em Imagens de Cenas Naturais

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientadora: *Edna Natividade da Silva Barros*

Recife
07 de julho de 2017

Resumo

O reconhecimento de caracteres em imagens de cenas naturais é uma tarefa desafiadora na área de Visão Computacional, uma vez que o *background* das imagens geralmente é não-uniforme, existe uma variedade de possíveis fontes, tamanhos irregulares dos caracteres na imagem, diferentes condições de iluminação, diferentes disposições do caractere na imagem e diferentes texturas. Além disso, aplicações que envolvem reconhecimento de caracteres em cenas naturais geralmente precisam ser embarcadas em circuitos integrados, que têm requisitos críticos, como baixo tempo de resposta do sistema, baixa potência e suporte à mobilidade, ao mesmo tempo em que requerem boa taxa de reconhecimento. Nesse trabalho, propomos um sistema heterogêneo que combina hardware e software para acelerar uma técnica de reconhecimento de caracteres em cenas naturais, baseada em *Histogram of Oriented Objects* e em uma rede neural *Extreme Learning Machine*, para aplicações embarcadas com restrições de tempo, área e consumo de energia. Os resultados experimentais mostraram que o sistema tem acurácia de 65.5%, resultado superior aos trabalhos comparados nesse documento, é capaz de processar até 11 *frames* por segundo e ocupa apenas 11% dos elementos lógicos da FPGA Cyclone IV, sendo viável sua utilização em sistemas embarcados com restrições de tempo e área.

Palavras-chave: FPGA, processamento de imagens, scene text character recognition, acelerador em hardware, sistemas embarcados, visão computacional

Sumário

1 Introdução **1**
- 1.1 Objetivos 2
- 1.2 Estrutura do Trabalho 2

2 Fundamentos **3**
- 2.1 Scene Text Character Recognition 3
  - 2.1.1 Pré-processamento da Imagem 4
  - 2.1.2 Extração de Características 5
  - 2.1.3 Reconhecimento do Caractere 6
  - 2.1.4 *Benchmark* Chars74K 7
  - 2.1.5 *Scene Text Character Recognition* em aplicações de tempo real 7
- 2.2 Tecnologias de Projeto 7
  - 2.2.1 Central Processing Units e Graphic Processing Units 7
  - 2.2.2 *Field Programmable Gate Array* 8

3 Trabalhos Relacionados **9**
- 3.1 Character Recognition in Natural Images [1] 9
- 3.2 Character Recognition In Natural Scene Images Using Rank1 Tensor Decomposition [2] 9
- 3.3 Scene Character Recognition Using PCANet [3] 10
- 3.4 Scene Text Recognition in Mobile Applications by Character Descriptor and Structure Configuration [4] 10
- 3.5 Comparação das Abordagens 10

4 Algoritmo Proposto para STCR **11**
- 4.1 Visão Geral 11
- 4.2 Conversão do Espaço de Cores RGB para Grayscale 12
- 4.3 Resize 12
- 4.4 Limiarização de Otsu 15
- 4.5 Extrator de Características HOG 17
- 4.6 Classificador Extreme Learning Machine 20
- 4.7 Resultados da Estratégia Proposta 22

5 Arquitetura em Hardware e Software Proposta para STCR **25**
5.1 Visão Geral 25
5.2 Comunicação CPU-FPGA 28
5.2.1 Comunicação - Software 28
5.2.2 Comunicação - Hardware 31
FPGA_RX 32
FPGA_TX 33
5.3 O Módulo BIA 33
5.3.1 Buffer Control Unit 36
5.3.2 Kernel Function Unit 37
5.3.3 Filter Control Unit 38
5.4 Implementação do módulo BIA 40

6 Resultados e Experimentos **43**
6.1 Configuração dos Experimentos 43
6.1.1 Parâmetros da *Extreme Learning Machine* 44
6.2 Performance 45
6.2.1 Profiling 45
6.2.2 Comparação com Outros Métodos 45
6.2.3 Ponto Fixo 46

7 Conclusão **49**

Lista de Figuras

2.1 Exemplo de imagens de caracteres em cenas naturais e suas respectivas classes. A similaridade entre imagens de classes diferentes é um dos desafios em *Scene Text Character Recognition*. 3
2.2 *Flowchart* de um sistema de STCR. 4
2.3 Segmentação do caractere, através de uma operação de limiarização. 4
2.4 2.4(a) ilustra a reconstrução de detalhes do caractere, através de uma operação de dilatação. 2.4(b) ilustra remoção de ruídos, através de uma operação de erosão. 4
2.5 Remoção de objetos indesejados na imagem, através de Análise de Componentes Conectados. 5
2.6 Exemplo de imagens de caracteres de diferentes tamanhos em cenas naturais. Redimensionar as imagens para um tamanho comum é fundamental para o funcionamento correto dos extratores de características. 5
2.7 As figuras (a), (b) e (c) foram retiradas da classe correspondente à letra I no *dataset* Chars74K-15. Elas ilustram algumas das possíveis variações nas imagens de uma mesma classe (letra I). Na figura 2.7(a), o caractere "I"se localiza no centro da imagem e ocupa aproximadamente 50% dela, enquanto que na figura 2.7(b), o caractere "I"ocupa quase toda a imagem e não está centralizado. Na figura 2.7(c) , o mesmo caractere está rotacionado e transladado. 6

4.1 Fluxo de execução do sistema de STCR proposto 11
4.2 Método de Interpolação Bicúbica. A região sombreada sobre a imagem de entrada representa a vizinhança de 4×4 pixels (janela) utilizada para produzir um pixel na imagem de saída. É possível observar nas figuras (a) e (b) que pontos próximos na imagem de saída ((i, j) e $(i, j+1)$) são gerados por pontos distantes na imagem de entrada. Adicionalmente, pode-se ver que para dois pixels de saída diferentes, os valores do kernel não são os mesmos. 15
4.3 Definição de células e blocos no algoritmo de HOG. É possível observar os dois blocos adjacentes 1 e 2 (linha azul pontilhada e linha vermelha contínua, respectivamente) se sobrepõem. Nesse trabalho, utilizamos células de tamanho 18×18 pixels e blocos de tamanho 36×36 pixels (2×2 células) 18
4.4 Exemplo do histograma de uma célula. Os ângulos mostrados no eixo x são o ponto médio de cada intervalo. 19
4.5 Rede neural com uma camada escondida 21
4.6 Número de neurônios da camada escondida em função da taxa de acerto da rede neural ELM com função de ativação tangente hiperbólica. 23

4.7 Número de neurônios da camada escondida em função da taxa de acerto da rede neural ELM com função de ativação linear. 23

5.1 Duração média, em milissegundos, de cada etapa do algoritmo de STCR. 26
5.2 Arquitetura proposta. 27
5.3 Arquitetura da comunicação, referente ao envio de dados da CPU para a FPGA. 29
5.4 Arquitetura da comunicação, referente ao envio de dados da FPGA para a CPU. 31
5.5 Máquina de estado da unidade de controle do módulo FPGA_RX. 32
5.6 Máquina de estado da unidade de controle do módulo FPGA_TX. 33
5.7 Arquitetura do Bicubic Interpolation Accelerator e sua integração com os módulos FPGA_RX e FPGA_TX. 34
5.8 Arquitetura do Buffer Control Unit. 36
5.9 Exemplo de um mapeamento dos pixels da imagem de entrada nas BRAMs. Os pixels são armazenados na BRAM com a mesma cor. Os pixels consecutivos nunca estarão armazenados na mesma BRAM. 37
5.10 Arquitetura do KFU. 38
5.11 Paralelismo do algoritmo de interpolação bicúbica explorado pelo FCU: a cada ciclo de clock, 4 pixels da janela são processados em paralelo. 39
5.12 Máquina de estados da FCU. 39
5.13 Fluxo geral do projeto do módulo BIA. 40
5.14 Modelo de *testbench* utilizado para validar o módulo em hardware implementado. 41

6.1 Número de neurônios da camada escondida em função do tempo médio que a rede neural ELM leva para classificar uma imagem do conjunto de teste. 44
6.2 Duração média, em milissegundos, de cada etapa do algoritmo de STCR na arquitetura híbrida proposta em comparação com a abordagem apenas em software 45
6.3 Número de neurônios da camada escondida em função da taxa de acerto da rede neural ELM com função de ativação tangente linear. A curva do sistema que utiliza ponto flutuante é indicada por quadrados, enquanto que o sistema que utiliza ponto fixo é indicado por círculos. 47

Lista de Tabelas

3.1 Performance dos métodos de STCR no dataset Chars74K-15 10

4.1 Parâmetros do algoritmo HOG utilizados nesse trabalho 20
4.2 Linguagem utilizada e número de linhas de código de cada etapa da estratégia proposta 22

5.1 Linguagem utilizada e número de linhas de código de cada módulo em hardware e para as tarefas *Send Data CPU - FPGA e Receive Data FPGA - CPU* 41
5.2 Percentual da utilização de recursos da FPGA Cyclone IV pelos módulos em hardware 42

6.1 Comparação dos métodos de STCR no dataset Chars74K-15. 45

Lista de Acrônimos

STCR Scene Text Character Recognition
OCR Optical Character Recognition
PPU Processador de Propósito Único
FPGA Field Programmable Gate Array
DSP Digital Signal Processor
CPU Cental Processing Unit
IP Intellectual Property
NSCR Natural Scene Character Recognition
HOG Histogram of Oriented Gradients
CNN Convolutional Neural Networks
SVM Support Vector Machine
RBF Radial Basis Function
FERNS Ferns Classifier
NRE Non-recurring Engineering
GPU Graphic Processing Unit
VLSI Very Large Scale Integration
SC Shape Context
GB Geometric Blur
SIFT Scale Invariant Feature Transform
MR8 Maximum Response of filters
PCH Patch descriptor
MKL Multiple Kernel Learning
I2CDML Image-to-class Distance Metric Learning
PCANet Principal Component Analysis Network
PCA Principal Component Analysis
BOW Bag-of-Words
GMM Gaussian Mixture Model
ELM Extreme Learning Machine
RNA Rede Neural Artificial
SO Sistema Operacional
BIA Bicubic Interpolation Accelerator
BCU Buffer Control Unit
KFU Kernel Function Unit
FCU Filter Control Unit

RTL Register Transfer Level
TDP Thermal Design Power

Capítulo 1

Introdução

Scene Text Character Recognition (STCR) e *Optical Character Recognition* (OCR) são tarefas desafiadoras nas áreas de Processamento de Imagens e Aprendizagem de Máquina, que objetivam desenvolver sistemas computacionais capazes de reconhecer automaticamente caracteres em imagens digitais.

Esses sistemas têm sido amplamente utilizados em aplicações industriais e comerciais. Um exemplo de aplicações clássicas de OCR é a digitalização automática de passaportes, extratos bancários e outros documentos impressos por um computador. Nesse contexto, os caracteres geralmente estão impressos em um mesmo tipo de fonte e estão dispostos em um background aproximadamente uniforme.

Por outro lado, aplicações de STCR, como assistentes de navegação mobile e scene understanding [4], reconhecimento automático de placas de [5] e busca de imagens pelo seu conteúdo textual envolvem o reconhecimento de caracteres em cenas naturais (natural scenes). Nessa perspectiva, o reconhecimento de caracteres pode ser mais desafiador e computacionalmente custoso, uma vez que o background geralmente é não-uniforme, existe uma variedade de possíveis fontes, tamanhos irregulares dos caracteres, diferentes condições de iluminação, diferentes disposições do caractere na imagem e diferentes texturas [6].

Além disso, algumas aplicações que utilizam STCR precisam ser embarcadas em circuitos integrados ou dispositivos móveis, que têm requisitos críticos, como baixo tempo de resposta do sistema, baixo custo, baixa potência e suporte a mobilidade, ao mesmo tempo em que requerem boa taxa de reconhecimento. Alguns exemplos desse tipo de aplicação são os assistentes para deficientes visuais [7] e veículos autônomos [5].

Uma solução para esses problemas é implementar o algoritmo de STCR em uma arquitetura embarcada heterogênea, composta por um processador embarcado de propósito geral e aceleradores em hardware (PPUs, como FPGAs ou DSPs). Nesse tipo de solução, as tarefas mais computacionalmente custosas do algoritmo são implementadas diretamente em hardware, o que permite que a solução atinja alto desempenho, enquanto mantém baixa potência, área reduzida e custo reduzido.

Vários métodos de STCR têm sido propostos na literatura. Por exemplo, Campos et al. [1] introduziram o *benchmark* de caracteres em cenas naturais Chars74K-15 e provaram que ferramentas comerciais de reconhecimento de caracteres em documentos não têm bom desempenho nesse tipo de imagem. Ali et al. [2], Chen et al. [3] e Yi et al. [4] propuseram novas técnicas

para melhorar o desempenho do reconhecimento de caracteres no *benchmark* Chars74k-15. Apesar de esses trabalhos possuírem boa precisão, o tempo de reconhecimento de cada caractere é relativamente alto, o que pode inviabilizar seu uso em sistemas de tempo real.

Por outro lado, alguns trabalhos recentes utilizaram FPGA para acelerar o reconhecimento de caracteres [8], [9]. Embora o tempo de execução seja menor do que as abordagens em software, as técnicas não foram avaliadas em um *benchmark* de caracteres em cenas naturais, como o Chars74k-15. Portanto, não é possível fazer uma comparação justa entre esses métodos e as abordagens em software.

Nessa perspectiva, desenvolvemos um sistema heterogêneo de hardware e software para reconhecimento automático de caracteres em cenas naturais, que foi prototipado na placa Terasic DE2i-150 [10], uma plataforma embarcada híbrida composta por um processador Intel ATOM N2600 conectado a uma FPGA da família Cyclone IV da Altera, via PCI Express. Os resultados mostraram que o sistema atinge um *speedup* de até 11.7 em relação a outros sistemas de STCR e acurácia de 65.5% no *benchmark* Chars74k-15 [1].

1.1 Objetivos

O objetivo deste trabalho foi propor, implementar e avaliar o desempenho de uma arquitetura heterogênea de hardware e software (CPU-FPGA) para acelerar o tempo de execução de uma técnica de reconhecimento automático de caracteres em cenas naturais para aplicações que precisam ser embarcadas em dispositivos com restrições de tempo, área e potência.

A implementação do algoritmo de STCR foi dividida entre hardware e software. Os módulos de hardware foram implementados como IP Cores, de modo a acelerar alguma das tarefas mais computacionalmente custosas do algoritmo e foram prototipados em FPGA. Por outro lado, os módulos de software executam na CPU embarcada.

1.2 Estrutura do Trabalho

O restante deste trabalho está organizado da seguinte maneira: o capítulo 2 apresenta conceitos fundamentais da área de reconhecimento de caracteres em cenas naturais e quais tecnologias de projetos têm sido utilizadas. O capítulo 3 sumariza alguns dos principais trabalhos da área, que foram avaliados no *benchmark* Chars74K-15. O capítulo 4 aborda o algoritmo proposto para STCR e os detalhes da implementação em software. O capítulo 5 apresenta a implementação em hardware e software desenvolvida para acelerar o algoritmo proposto no capítulo 4. No capítulo 6, descrevemos os resultados do sistema proposto e os experimentos realizados. No capítulo 7, concluímos o trabalho e abordamos possíveis trabalhos futuros que podem extendê-lo.

Capítulo 2

Fundamentos

Neste capítulo, introduzimos alguns conceitos fundamentais para o entendimento do trabalho proposto e abordamos as tecnologias de implementação utilizadas.

2.1 Scene Text Character Recognition

Scene Text Character Recognition (STCR) ou *Natural Scene Character Recognition* (NSCR) é uma aplicação de visão computacional que objetiva classificar caracteres individuais a partir de imagens de cenas naturais. Porém, reconhecer caracteres nesse tipo de imagem é uma tarefa desafiadora e computacionalmente custosa, uma vez que o *background* geralmente é não-uniforme, os caracteres são de fontes e tamanhos diversos, existem diferentes condições de iluminação, diferentes disposições do caractere na imagem, diferentes texturas e presença de ruído. Por exemplo, a figura 2.1 ilustra como diferentes fontes, texturas e o nível de ruído podem tornar caracteres de classes diferentes muito similares.

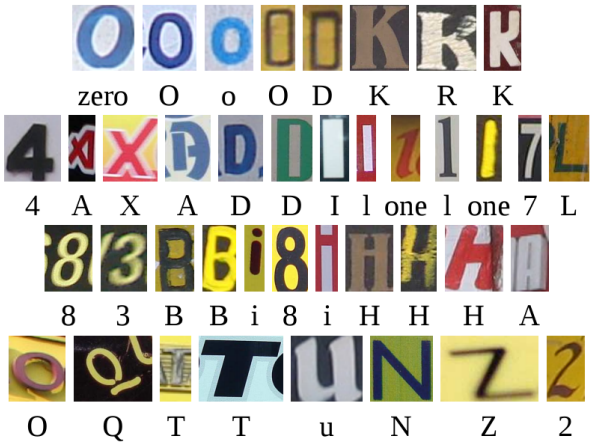


Figura 2.1: Exemplo de imagens de caracteres em cenas naturais e suas respectivas classes. A similaridade entre imagens de classes diferentes é um dos desafios em *Scene Text Character Recognition*.

Várias técnicas têm sido propostas na literatura a fim de endereçar esses problemas. De uma maneira geral, a maioria dos algoritmos propostos contêm 3 etapas principais: (1) Pré-processamento da imagem, (2) Extração de Características e (3) Reconhecimento do Caractere. A figura 2.2 ilustra o *flowchart* de um sistema de *STCR*. Para cada uma das três etapas, diferentes abordagens podem ser utilizadas, as quais serão discutidas a seguir.

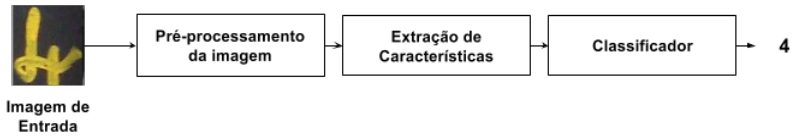


Figura 2.2: *Flowchart* de um sistema de STCR.

2.1.1 Pré-processamento da Imagem

Nessa etapa, algoritmos de processamento de imagens são utilizados para melhorar a qualidade da informação visual, a fim de que o vetor de características que é gerado na etapa 2 contenha apenas informações relacionadas ao caractere em si e não informações relativas a ruídos [11], ou à mudança de coloração do *background*, por exemplo. As técnicas de pré-processamento mais utilizadas são:

- Conversão de Espaço de Cores e Limiarização: geralmente são empregadas em conjunto com o objetivo de segmentar o caractere do *background* (figura 2.3).

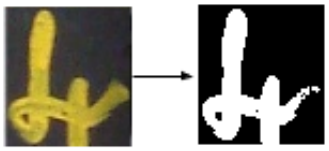


Figura 2.3: Segmentação do caractere, através de uma operação de limiarização.

- Operações Morfológicas: Operações como Erosão e Dilatação podem ser utilizadas para remoção de ruídos e reconstrução de objetos na imagem (figura 2.4).

(a) (b)

Figura 2.4: 2.4(a) ilustra a reconstrução de detalhes do caractere, através de uma operação de dilatação. 2.4(b) ilustra remoção de ruídos, através de uma operação de erosão.

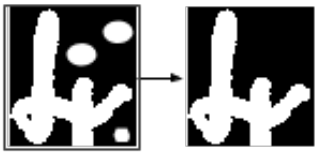


Figura 2.5: Remoção de objetos indesejados na imagem, através de Análise de Componentes Conectados.

- Análise de Componentes Conectados: pode ser utilizada para remoção de ruídos ou para isolar o caractere de outros objetos na imagem (figura 2.5).

Adicionalmente, a maioria dos algoritmos de Extração de Características, como o *Histogram of Oriented Gradients* (HOG) [12] e *Convolutional Neural Networks* (CNN) [13], exigem que todas as imagens de entrada sejam de um tamanho específico. No entanto, imagens de caracteres em cenas reais podem variar de dezenas a milhares de pixels (figura 2.6), dependendo da distância entre o caractere e a câmera e do tipo da fonte, por exemplo. Portanto, é necessário aplicar algum método de redimensionamento de imagens na etapa Pré-processamento da Imagem, como os algoritmos de interpolação cúbica *spline* [14] e interpolação bicúbica [15], a qual será detalhada na seção 4.3.

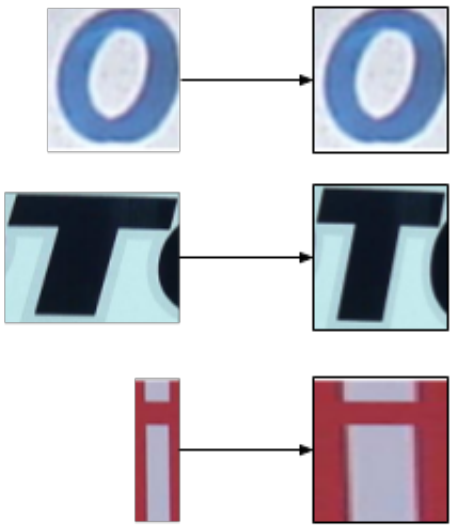
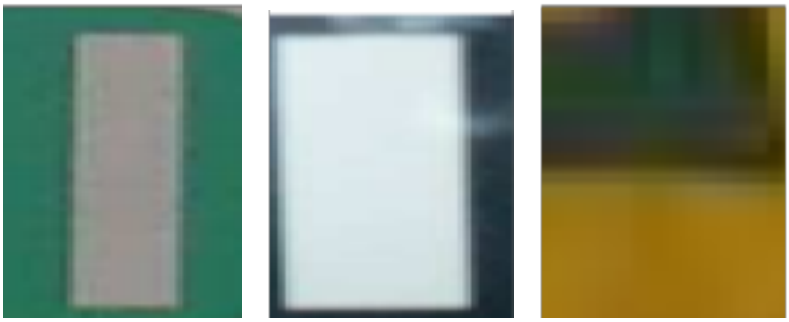


Figura 2.6: Exemplo de imagens de caracteres de diferentes tamanhos em cenas naturais. Redimensionar as imagens para um tamanho comum é fundamental para o funcionamento correto dos extratores de características.

2.1.2 Extração de Características

Dada uma imagem entrada que foi pré-processada, essa etapa é responsável por extrair características da imagem que são intrínsecas à classe de caracteres à qual ela pertence. Nesse contexto,

um bom extrator deve produzir características similares para todos os caracteres pertencentes a uma mesma classe, mesmo que haja: (1) variações no tamanho do caractere na imagem, (2) rotação e/ou (3) translação. A figura 2.7 ilustra esses três cenários.



(a) (b) (c)

Figura 2.7: As figuras (a), (b) e (c) foram retiradas da classe correspondente à letra I no *dataset* Chars74K-15. Elas ilustram algumas das possíveis variações nas imagens de uma mesma classe (letra I). Na figura 2.7(a), o caractere "I"se localiza no centro da imagem e ocupa aproximadamente 50% dela, enquanto que na figura 2.7(b), o caractere "I"ocupa quase toda a imagem e não está centralizado. Na figura 2.7(c) , o mesmo caractere está rotacionado e transladado.

Nesse contexto, três principais tipos de extratores de características têm sido utilizados nos trabalhos propostos na literatura [16]:

- *Histogram of Oriented Gradients* (HOG) e seus variantes [17], [18]
- *Convolutional Neural Networks* (CNN) e outros métodos baseados em *deep learning* [19], [20]
- Características estruturais dos caracteres [21], [22]

O melhor algoritmo para extrair as características das imagens varia de acordo com a aplicação e suas restrições. Portanto, a escolha do extrator de características deve levar em consideração a acurácia, a complexidade do algoritmo e a utilização de recursos computacionais.

2.1.3 Reconhecimento do Caractere

Nessa etapa, um classificador recebe como entrada o vetor de características da imagem de entrada, gerado na etapa anterior, e classifica a imagem correspondente em sua respectiva classe. Uma das técnicas de classificação mais utilizadas pelos trabalhos propostos na literatura [16] é o *Support Vector Machine* (SVM), com *kernel* linear, Radial Basis Function (RBF) ou chi-quadrado. Porém, outros classificadores como Redes Neurais Convolucionais (CNN), *Random Ferns Classifier* (FERNS), *Random Forests* e Redes Neurais *feed-forward* também têm sido utilizados.

De maneira similar à etapa de Extração de Características, a escolha do classificador tem impacto direto na acurácia do sistema e deve ser escolhido levando em consideração também as restrições de tempo e recursos computacionais.

2.1.4 *Benchmark* Chars74K

O *dataset* Chars74K [1] é um dos *benchmarks* mais usados para avaliação do desempenho e precisão de aplicações de STCR. Ele é composto por imagens de caracteres cortadas de imagens do *Google Street View*. Essas imagens consistem em caracteres da língua inglesa cortados de cenas naturais, como placas de carro, números de casas, propagandas em outdoors, cartazes nas ruas, etc.

O *dataset*, que contém 7705 imagens, diferencia números, letras maiúsculas e letras minúsculas, portanto, é composto por 62 classes ([0-9], [A-Z], [a-z]). Porém, para tornar a avaliação dos sistemas de reconhecimento de caracteres mais robusta, os autores do *dataset* recomendam que as aplicações de STCR sejam avaliadas e comparadas utilizando o dataset Chars74K-15.

O Chars74K-15 é um subconjunto do Chars74K, no qual só existem 15 imagens por classe de treinamento e 15 imagens por classe de teste, totalizando 930 imagens de treinamento e 930 imagens de teste. As imagens têm variações significativas na resolução, tamanho, alinhamento, rotação, translação e ruído. Os caracteres da figura 2.1 foram obtidos do Chars74K-15

2.1.5 *Scene Text Character Recognition* em aplicações de tempo real

Sistemas de *Scene Text Character Recognition* são fundamentais para várias aplicações de tempo real, como reconhecimento de placas de trânsito [5], navegação *mobile* e *scene understanding* [4] e assistentes de navegação para deficientes visuais [7]. No entanto, para atingir uma boa taxa de reconhecimento, os sistemas têm se tornado cada vez mais caros computacionalmente, demandando mais tempo para processar cada imagem.

Além disso, geralmente as aplicações de *Scene Text Character Recognition* em tempo real precisam ser embarcadas em circuitos integrados ou dispositivos móveis, que têm restrições de memória e área, ao mesmo tempo em que precisam manter uma boa acurácia e serem rápidos.

2.2 Tecnologias de Projeto

A escolha da Tecnologia de Projeto utilizada é fundamental para que um sistema embarcado de tempo real para *Scene Text Character Recognition* funcione de acordo com os requisitos de tempo, memória e acurácia.

2.2.1 Central Processing Units e Graphic Processing Units

Tradicionalmente, algoritmos de visão computacional são completamente implementados em CPUs (Central Processing Units). Nesse tipo de arquitetura, a aplicação é representada como um conjunto de instruções sequenciais e o papel principal da CPU consiste em: ler cada instrução da memória, decodificá-las para determinar qual operação será executada, para finalmente executar

a instrução. Esse processo geralmente torna lenta a execução de cada operação individual, o que poderia ser solucionado aumentando a frequência do clock ou adicionando múltiplos cores. Porém, essas duas soluções também geram um aumento na área e energia consumida pelo processador [23].

Nesse contexto, uma grande vantagem de utilizar CPUs em aplicações de visão computacional é que o desenvolvimento da aplicação se torna mais simples, uma vez que a codificação geralmente é realizada em linguagens de alto nível, reduzindo o custo NRE (*non-recurring-engineering*) do projeto. No entanto, utilizar apenas essa arquitetura pode inviabilizar o desenvolvimento de uma aplicação de tempo real embarcada, devido à limitação da velocidade de processamento.

Por outro lado, as GPUs (Graphic Processing Units) são outra tecnologia bastante utilizada em projetos de visão computacional. Esse tipo de processador é de baixo custo, tem várias ferramentas e linguagens de programação consolidadas e, diferentemente das CPUs, consegue atingir um alto nível de paralelismo. Porém, restrições de consumo de energia inviabiliza sua utilização em muitas aplicações em sistemas embarcados [23].

2.2.2 *Field Programmable Gate Array*

Diferentemente das CPUs e GPUs, as *Field Programmable Gate Array* (FPGAs) têm características únicas que são fundamentais para desenvolver sistemas de visão computacional. As FPGAs combinam o processamento inerentemente paralelo do hardware com a flexibilidade de um software, no sentido de que sua funcionalidade pode ser reconfigurada e reprogramada à medida que as tecnologias mudam ou evoluem, diferentemente do VLSI (*Very Large Scale Integration*) [23]. Além disso, as FPGAs conseguem atingir alto desempenho, enquanto mantém baixo consumo de energia, em comparação com CPUs e GPUs [23]. O problema é que alguns algoritmos de visão computacional são muito complexos para serem completamente implementados em FPGA em um curto espaço de tempo.

Nesse contexto, o desenvolvimento sistemas híbridos (ou sistemas heterogêneos) têm sido uma tendência no projeto de aplicações complexas e que requerem alto desempenho. Nesse tipo de sistema as FPGAs funcionam como coprocessadores ou aceleradores integrados a outro tipo de processador, com o objetivo de manter uma boa relação entre desempenho, preço e consumo de energia. Portanto, sistemas heterogêneos são fortes candidatos no projeto de um sistema embarcado de visão computacional em tempo real.

Capítulo 3

Trabalhos Relacionados

Nesse capítulo são apresentadas as técnicas e resultados de trabalhos relevantes na área de reconhecimento de caracteres, avaliados na base de dados Chars74K-15.

3.1 Character Recognition in Natural Images [1]

Esse trabalho introduziu o dataset Chars74K e mostrou que ferramentas comerciais de OCR (Optical Character Recognition) não têm boa acurácia quando são aplicadas a STCR. Os autores usaram vários extratores de características como Shape Context (SC), Geometric Blur (GB), Scale Invariant Feature Transform (SIFT), Spin image, Maximum Response of filters (MR8) e Patch descriptor (PCH) e classificadores baseados em SVM, Nearest Neighbours (NN), Multiple Kernel Learning (MKL) .

O melhor resultado que os autores alcançaram no dataset Chars74K-15 foi 55.26% de acurácia, usando a técnica MKL e os extratores de características SC, GB, SIFT, MR8 e PCH em conjunto. Todas as técnicas foram implementadas em software e os tempos de execução de cada método não foram abordados no artigo.

3.2 Character Recognition In Natural Scene Images Using Rank1 Tensor Decomposition [2]

Os autores propuseram um novo sistema de STCR que utiliza decomposição de tensores para extrair as características da imagem e *image-to-class distance metric learning* (I2CDML) como classificador. Os autores afirmam que seus resultados são melhores do que os métodos baseados em extratores de características locais, como HOG.

A técnica proposta foi implementada em software e atinge uma acurácia de 59% no dataset Chars74K-15. Mais uma vez, os autores focaram na técnica de reconhecimento e não adicionaram informação relativas ao tempo de processamento da imagem e reconhecimento do caractere.

3.3 Scene Character Recognition Using PCANet [3]

Nesse projeto, os autores propõem o uso da técnica Principal Component Analysis Network (PCANet) como extrator de características e classificador para STCR. PCANet é uma técnica de deep learning, na qual vários classificadores baseados em PCA são associados em cascata, de modo parecido com as Redes Neurais Convolucionais (CNN).

O método proposto alcançou acurácia de 64% no dataset Chars74K-15 e leva aproximadamente 1 segundo para processar cada imagem de teste. Da mesma forma que os trabalhos anteriores, o sistema foi projetado para executar em uma CPU.

3.4 Scene Text Recognition in Mobile Applications by Character Descriptor and Structure Configuration [4]

Esse trabaho propõe um método de reconhecimento de texto em cenas naturais para aplicações mobile. Os autores propuseram um extrator de características que consiste em três estágios. Dada uma imagem de entrada, o algoritmo de HOG é aplicado aos keypoints da imagem, que são detectados através dos algoritmos Harris Detector, MSER Detector, Dense Detector e Random Detector. Então, os modelos Bag-of-Words (BOW) e Gaussian Mixture Model (GMM) são empregados para agregar as características extraídas. Para classificação, os autores utilizaram o algoritmo de Adaboost em cascada.

O método proposto atingiu taxa de acerto de 60% no dataset Chars74K-15 e o tempo para processar cada frame é de aproximadamente 1 segundo, resultado similar ao trabalho anterior. O sistema foi avaliado em um Samsung Galaxy II, com Android 2.3. Esse dispositivo contém um 1.2 GHz dual-core ARM Cortex-A9 e uma GPU GeForce ULP de oito núcleos.

3.5 Comparação das Abordagens

A tabela 3.1 sintetiza os resultados dos trabalhos descritos nas seções anteriores.

Tabela 3.1: Performance dos métodos de STCR no dataset Chars74K-15

Método	Tempo de Execução	Arquitetura	Acurácia
MKL [1]	-	-	55.76%
GB+SVM [1]	-	-	52.58%
SIFT+SVM [1]	-	-	21.40%
Rank1 Tensor + I2CDML [2]	-	-	59%
PCANet [3]	1s	CPU	64%
Keypoints + HOG + Adaboost [4]	1s	CPU ARM Cortex-A9 1.2 GHz dual-core + GPU GeForce ULP	60%

Capítulo 4

Algoritmo Proposto para STCR

Nesse capítulo, abordamos o algoritmo proposto para *Scene Text Character Recognition* e descrevemos a implementação em software da técnica proposta.

4.1 Visão Geral

Conforme visto no capítulo 2, um sistema de reconhecimento de caracteres geralmente é formado por 3 etapas: (1) pré-processamento da imagem, (2) extração de características e (3) classificação. O maior desafio nessa área é escolher quais algoritmos devem ser utilizados em cada uma das etapas, com o objetivo de manter um compromisso entre acurácia e tempo de processamento.

Nesse contexto, propomos um novo sistema de reconhecimento de caracteres em cenas naturais, baseado em *Histogram of Oriented Gradients* (HOG) e em uma rede neural *Extreme Learning Machine*. A figura 4.1 ilustra o fluxo de execução da técnica e a imagem de saída gerada por cada módulo.

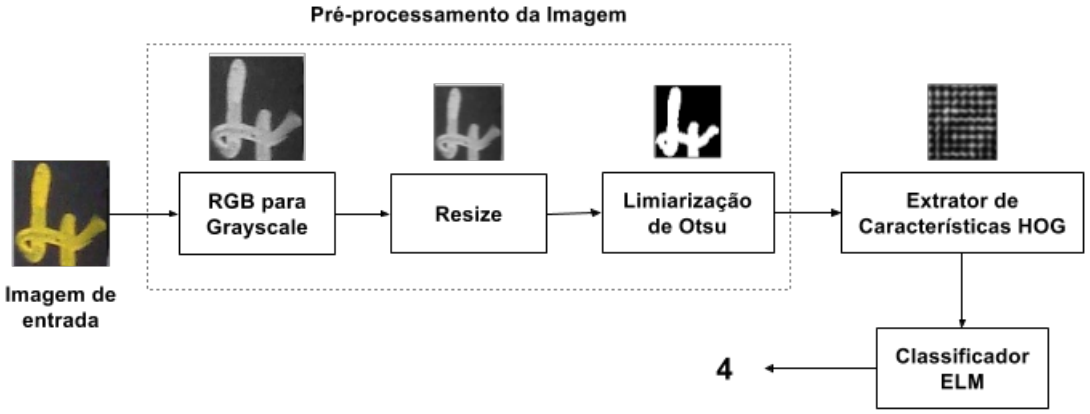


Figura 4.1: Fluxo de execução do sistema de STCR proposto

Como ilustrado na figura 4.1, o sistema proposto recebe como entrada uma imagem no espaço de cores RGB. Na etapa de pré-processamento, a imagem de entrada é primeiro convertida para o espaço de cores em escala de cinza, com o objetivo de reduzir a quantidade de informação da

imagem relativa a cor. Depois, a imagem em tons de cinza é redimensionada para um tamanho padrão de 128×128 pixels, com o objetivo de padronizar o tamanho das imagens de entrada. Finalmente, a imagem de tamanho 128×128 pixels é limiarizada, através da técnica de Otsu [24], com o objetivo de segmentar o caractere do *background*.

Na próxima etapa, um extrator de características baseado em *Histogram of Oriented Objects* (HOG) [12] é aplicado à imagem limiarizada, com o objetivo de gerar um vetor de dimensionalidade 1296 contendo as características relativas ao caractere ao qual a imagem pertence.

Por fim, esse vetor de características é a entrada de uma rede neural Extreme Learning Machine (ELM) [25], que é responsável por classificar o caractere.

Nas próximas seções, detalhamos a implementação de cada módulo da técnica proposta.

4.2 Conversão do Espaço de Cores RGB para Grayscale

Um espaço de cores (ou sistema de cores) objetiva definir um padrão de representação de cores. Ou seja, ele define um sistema de coordenadas e um sub-espaço dentro desse sistema, de modo que cada cor pode ser representada por um ponto (x, y) [26].

O espaço de cores mais comum na representação de imagens é o RGB. Nesse sistema, cada pixel é representado pela combinação das três cores primárias (vermelho, verde e azul), portanto cada pixel corresponde a um vetor de três componentes: (1) R - vermelho, (2) G - verde e (3) B - azul. Nesse trabalho, cada componente é representado por 8 bits, consequentemente, cada pixel é capaz de representar $(2^8)^3 = 16.777.216$ cores. O problema é que uma quantidade tão grande de informação de cor torna o STCR ainda mais difícil, uma vez que a coloração dos caracteres na imagem varia muito nesse tipo de problema.

Para resolver esse problema, na primeira etapa do sistema de STCR proposto convertemos cada pixel da imagem de entrada do espaço de cores RGB para um espaço em escala de cinza, através de uma soma ponderada dos componentes RGB (equação 4.1):

$$Gray = 0.21 * R + 0.72 * G + 0.07 * B \tag{4.1}$$

No sistema *Grayscale* cada pixel corresponde a um número natural de 8 bits. Portanto, um pixel consegue representar apenas $2^8 = 256$ tons.

Após a conversão entre os espaços de cores RGB e *Grayscale*, o sistema passa para a etapa Resize, a qual será descrita a seguir.

4.3 Resize

Como visto no capítulo 2, as imagens de caracteres em cenas naturais podem variar de centenas a milhares de pixels, portanto utilizamos o algoritmo de interpolação bicúbica para redimensionar as imagens de entrada de um tamanho qualquer para um tamanho padrão de 128×128 pixels. Como o dataset Chars74K-15, utilizado para avaliar a acurácia desse trabalho, contém muitas imagens pequenas (menores que 20×20 pixels), o redimensionamento permite que mais características sejam extraídas da imagem, aumentando a variabilidade dos dados e a acurácia. Decidimos redimensionar todas as imagens para o tamanho 128×128 pixels e utilizar

o algoritmo de interpolação bicúbica, porque estes foram os parâmetros que forneceram a melhor acurácia do sistema.

Uma função de interpolação é um tipo de função de aproximação que objetiva estimar valores intermediários de uma função contínua a partir de amostras discretas, com a condição de que o valor da função contínua coincida com o valor da amostra discreta nos pontos de interpolação. Ou seja, dada uma função de interpolação f e uma função discreta g, $g(x) = f(x)$ para todo x no domínio de g [27].
Nesse contexto, uma função de interpolação em uma dimensão pode ser definida como uma convolução entre uma função discreta g e um *kernel* de interpolação r (equação 4.2):

$$f(x') = [r * g](x') = \sum_{m=-\infty}^{+\infty} r(dx - m).g(x + m) \tag{4.2}$$

na qual x' é o ponto onde se deseja estimar o valor de f, sendo $x = \lfloor x' \rfloor$ e $dx = x' - x$.
A principal diferença entre os diversos métodos de interpolação é o *kernel* de convolução. O *kernel* da interpolação bicúbica, método utilizado nesse trabalho, é composto por polinômios de terceiro grau definidos nos intervalos: $(-2,-1)$, $(-1,0)$, $(0,1)$ e $(1,2)$, fora desse intervalo, ele assume o valor zero (equação 4.3).

$$r_{cub}(x,a) = \begin{cases} (-a+2)|x|^3 + (a-3)|x|^2 - 1, & \text{se } 0 \leq |x| < 1, \\ -a|x|^3 + 5a|x|^2 - 8a|x| + 4a, & \text{se } 1 \leq |x| < 2, \\ 0, & \text{se } |x| \geq 2 \end{cases} \tag{4.3}$$

como $r_{cub} = 0$ quando $|x| \geq 2$, apenas quatro valores discretos de $g(u)$ são necessários para a operação de convolução. Portanto, a equação 4.2 pode ser reescrita como descrito abaixo.

$$f(x') = [r * g](x') = \sum_{m=-1}^{+2} r_{cub}(dx - m).g(x + m) \tag{4.4}$$

A equação 4.4 define o caso unidimensional da função de interpolação bicúbica. Como imagens digitais são bidimensionais, precisamos redefinir a função de interpolação para o caso 2D. Portanto, seja I uma imagem de entrada de $a \times b$ pixels e $R_{cub2D}(x,y) = r_{cub}(x)r_{cub}(y)$ o *kernel* da interpolação bicúbica 2-D, podemos definir a imagem redimensionada I_{red} (de $p \times q$ pixels) através da seguinte equação:

$$I_{red}(x',y') = \sum_{m=-1}^{+2} \sum_{n=-1}^{+2} I(x+m, y+n) r_{cub}(dx - m) r_{cub}(dy - n) \tag{4.5}$$

na qual (x',y') são as coordenadas de um pixel qualquer na imagem de saída, sendo $(x,y) = (\lfloor x' * \frac{a}{p} \rfloor, \lfloor y' * \frac{b}{q} \rfloor)$, $dx = x' * \frac{a}{p} - x$ e $dy = y' * \frac{b}{q} - y$.
Esse método de redimensionamento é descrito no algoritmo 1. Nesse trabalho, a quantidade de linhas e colunas da imagem de saída, O_x e O_y são iguais a 128. O algoritmo descreve uma convolução entre a imagem de entrada e o *kernel* (similar à um processo de filtragem de sinais em duas dimensões), utilizando uma vizinhança de 4×4 pixels (janela) para produzir cada pixel da imagem de saída.

As figuras 4.2(a) e 4.2(b) ilustram o processo descrito pelo algoritmo 1. Para cada pixel da imagem de saída, uma região de 4×4 pixels adjacentes da imagem de entrada (janela) é selecionada e cada pixel contido nessa região é multiplicado por um peso correspondente, definido pelo *kernel* de interpolação.

Apesar de esse processo ser bastante similar a uma filtragem, existem duas diferenças principais entre esses dois processos:

1. Na interpolação bicúbica, para imagens grandes de entrada, pontos distantes na imagem de entrada podem resultar em pontos próximos na imagem de saída (ver figuras 4.2(a) e 4.2(b)). No caso da filtragem, pontos próximos na imagem de entrada sempre geram pontos próximos na imagem de saída.

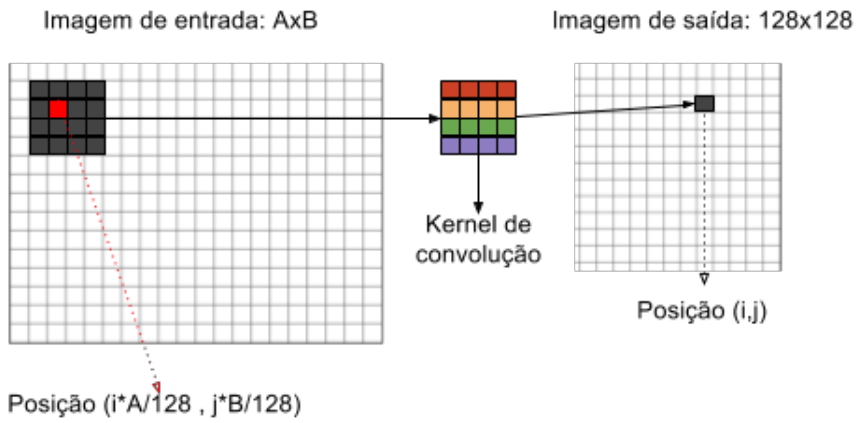
2. Na interpolação bicúbica, os valores do kernel de convolução, r_{cub}, variam a cada iteração, dependendo dos valores de dx e dy, o que pode ser observado na linha 15 do algoritmo 1. No caso da filtragem, os valores do filtro são estáticos.

Algoritmo 1: Redimensionamento de imagens, através do método de interpolação bicúbica

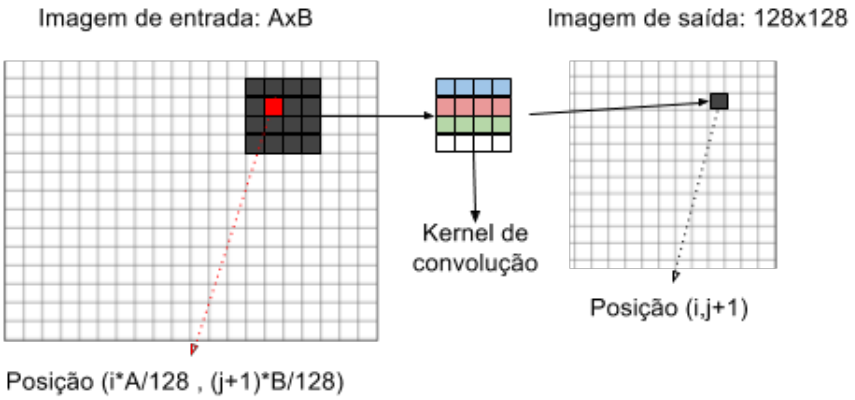
1 **Entrada:** I, a imagem original;
2 I_x e I_y, a quantidade de linhas e colunas da imagem original, respectivamente;
3 O_x e O_y, a quantidade de linhas e colunas da imagem de saída
4 **Saída:** O, a imagem redimensionada;
5 **for** $i \leftarrow 0 : O_x$ **do**
6 **for** $j \leftarrow 0 : O_y$ **do**
7 $x \leftarrow \lfloor i * \frac{I_x}{O_x} \rfloor$;
8 $y \leftarrow \lfloor j * \frac{I_y}{O_y} \rfloor$;
9 $O(i,j) \leftarrow 0$;
10 **for** $m \leftarrow -1 : 2$ **do**
11 **for** $n \leftarrow -1 : 2$ **do**
12 $dx \leftarrow i * \frac{I_x}{O_x} - y$;
13 $dy \leftarrow j * \frac{I_y}{O_y} - x$;
14 **if** $x + m < I_x$ ***and*** $y + n < I_y$ **then**
15 $O(i,j) \leftarrow$ $O(i,j) + I(x+m, y+n) * r_{cub}(dx-m)|_{a=0.5} * r_{cub}(dy-n)|_{a=0.5}$
16 **end**
17 **end**
18 **end**
19 **end**
20 **end**

A descrição detalhada do algoritmo 1 é feita na seção 5.3, na qual abordamos o Bicubic Interpolation Accelerator, um módulo em hardware para redimensionamento de imagens através do algoritmo de interpolação bicúbica.

Após o redimensionamento da imagem de entrada, o sistema passa para a etapa Limiarização de Otsu, a qual será descrita na próxima seção.



(a)



(b)

Figura 4.2: Método de Interpolação Bicúbica. A região sombreada sobre a imagem de entrada representa a vizinhança de 4×4 pixels (janela) utilizada para produzir um pixel na imagem de saída. É possível observar nas figuras (a) e (b) que pontos próximos na imagem de saída ((i, j) e $(i, j+1)$) são gerados por pontos distantes na imagem de entrada. Adicionalmente, pode-se ver que para dois pixels de saída diferentes, os valores do kernel não são os mesmos.

4.4 Limiarização de Otsu

Dada uma imagem I_{gray} de entrada, essa etapa é responsável por segmentar o caractere do *background*, através da equação 4.6:

$$I_{bin}(i,j) = \begin{cases} 255, & \text{se } I_{gray}(i,j) \geq \tau \\ 0, & \text{se } I_{gray}(i,j) < \tau \end{cases} \tag{4.6}$$

na qual (i, j) é a coordenada do pixel correspondente, I_{bin} é a imagem binária de saída e τ é o valor do limiar. O ideal é que o conjunto de pixels que correspondem ao caractere assumam o valor 255 (correspondente à cor branca) na imagem de saída, enquanto que os pixels que correspondem ao *background* assumam o valor 0, que corresponde à cor preta. Como as imagens

de *scene text* contêm diferentes condições de iluminação e *background* não-uniforme, definir um valor fixo de τ para toda as imagens tende a diminuir a acurácia da segmentação do caractere [24].

Nessa perspectiva, utilizamos o método de Otsu [24] para selecionar o limiar τ automaticamente, dependendo da imagem de entrada. Esse método itera sobre o histograma da imagem, com o objetivo de encontrar o valor de τ que maximize a separabilidade entre as classes (*foreground e background*) e, consequentemente, minimize a soma do espalhamento das duas classes, conforme o algoritmo 2. Ou seja, dado P, o histograma da imagem de entrada, o objetivo do método de Otsu é encontrar o valor τ (limiar ótimo) que minimize o valor da *weighted within-class variance* $\sigma_w^2(t)$, definida como:

$$\sigma_w^2(t) = q_b(t)\sigma_b^2(t) + q_f(t)\sigma_f^2(t) \tag{4.7}$$

onde as probabilidades das classes q_b e q_f são dadas por:

$$q_b(t) = \sum_{i=0}^{t} P(i) \tag{4.8}$$

$$q_f(t) = \sum_{i=t+1}^{255} P(i) \tag{4.9}$$

as médias das classes μ_b e μ_f:

$$\mu_b(t) = \sum_{i=0}^{t} \frac{iP(i)}{q_b(t)} \tag{4.10}$$

$$\mu_f(t) = \sum_{i=t+1}^{255} \frac{iP(i)}{q_f(t)} \tag{4.11}$$

e finalmente, a variância individual das classes σ_b^2 e σ_f^2:

$$\sigma_b^2(t) = \sum_{i=0}^{t} [i - \mu_b(t)]^2 \frac{P(i)}{q_b(t)} \tag{4.12}$$

$$\sigma_f^2(t) = \sum_{i=t+1}^{255} [i - \mu_f(t)]^2 \frac{P(i)}{q_f(t)} \tag{4.13}$$

O algoritmo abaixo descreve a implementação da limiarização usando o método de Otsu.

Algoritmo 2: Cálculo do valor do limiar ótimo, através do método de Otsu

Entrada: imagem em *grayscale* I_{gray}
Saída: limiar de Otsu τ
$\tau = 0$;
$min\_variance = \infty$;
P = histograma(I_{gray});
for $t \leftarrow 0,255$ **do**
    Calcule as probabilidades das classes, $q_b(t)$ e $q_f(t)$;
    Calcule as médias das classes, $\mu_b(t)$ e $\mu_f(t)$;
    Calcule a variância individual das classes, $\sigma_b^2(t)$ e $\sigma_f^2(t)$;
    Calcule $\sigma_w^2(t)$;
    if $\sigma_w^2(t) < min\_variance$ **then**
        $min\_variance = \sigma_w^2(t)$;
        $\tau = t$;
    end
end

Uma vez que a imagem de entrada foi limiarizada, a próxima etapa do sistema, descrita a seguir, é a Extração de Características usando o método HOG.

4.5 Extrator de Características HOG

Assim como no problema de reconhecimento de caracteres em cenas naturais, detectar pedestres em imagens de cenas naturais é uma tarefa desafiadora, uma vez o *background* é não-uniforme, existem diferentes condições de iluminação e a disposição das pessoas na imagem varia muito entre imagens diferentes. Para solucionar esses problemas, Dalal et al. [12] propuseram a utilização do *Histogram of Oriented Gradients* (HOG), um descritor de características para reconhecimento de pedestres, que alcança uma boa performance para esse tipo de problema.

Vários trabalhos recentes têm provado que o método HOG também é um extrator de características eficiente para outras aplicações, como reconhecimento de face [28], detecção de fungos [29], reconhecimento de fachadas de prédios em imagens naturais, detecção de veículos em imagens naturais [30] e reconhecimento de caracteres.

Nesse trabalho, utilizamos o método de HOG para extração de características dos caracteres em imagens de cenas naturais. A ideia básica do algoritmo é que a aparência e a forma de um objeto na imagem podem ser caracterizadas pela distribuição local da intensidade dos gradientes ou da orientação das bordas. Esse método é descrito pelo algoritmo 3. Cada passo do algoritmo

é descritos a seguir.

Algoritmo 3: Extração das características da imagem, através do algoritmo de HOG

Entrada: I, uma imagem em *grayscale* de 128x128 pixels;
Saída: H, vetor com as HOG *features* de I
Divida I em células de tamanho $cell\_size \times cell\_size$;
for *each cell* **do**
    Calcule os gradientes dos pixels de cada célula nas direções x e y;
    Calcule a orientação θ e magnitude m de cada gradiente.;
    Calcule o histograma dos gradientes, onde θ determina em quais intervalos do histograma aquele gradiente deve contribuir e m indica o quanto o gradiente contribui para aquele intervalo;
end
Normalize os gradientes, através da sobreposição de blocos;
O vetor de saída, H, é a concatenação de todos os vetores normalizados no passo anterior.

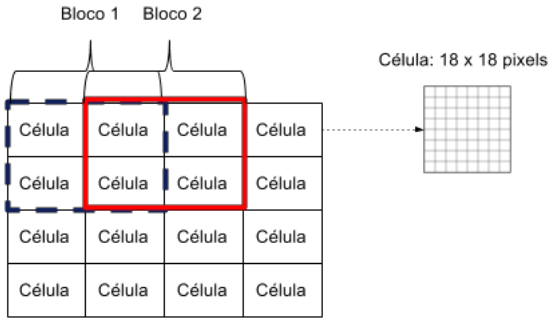


Figura 4.3: Definição de células e blocos no algoritmo de HOG. É possível observar os dois blocos adjacentes 1 e 2 (linha azul pontilhada e linha vermelha contínua, respectivamente) se sobrepõem. Nesse trabalho, utilizamos células de tamanho 18×18 pixels e blocos de tamanho 36×36 pixels (2×2 células)

No algoritmo 3, primeiro a imagem é dividida em células de tamanho $cell\_size \times cell\_size$, como na imagem 4.3. Depois, os gradientes nas direções x e y (g_x e (g_y, respectivamente) são calculados para cada pixel em uma célula, de acordo com as equações:

$$g_x(x,y) = I(x+1,y) - I(x-1,y) \tag{4.14}$$

$$g_y(x,y) = I(x,y+1) - I(x,y+1) \tag{4.15}$$

nas quais, $I(x,y)$ é a intensidade de um pixel na posição (x,y) na imagem I. A partir de g_x e g_y, podemos calcular os valores da magnitude $m(x,y)$ e direção $\theta(x,y)$ do vetor gradiente $G = [g_x, g_y]$, dados por:

$$m(x,y) = \sqrt[2]{g_x^2(x,y) + g_y^2(x,y)} \tag{4.16}$$

$$\theta(x,y) = arctan\frac{g_y}{g_x} \tag{4.17}$$

Com os valores de $\theta(x,y)$ e $m(x,y)$ calculados para todos os pixels dentro de uma célula, devemos gerar um histograma das orientações dos gradientes para a célula em questão. Nessa pesquisa, consideramos histogramas com 9 intervalos, divididos no intervalo [0° 180°), uma vez que Dalal et al. [12] mostraram que a acurácia do método HOG é superior com esses parâmetros. A figura 4.4 ilustra o exemplo de um possível histograma gerado nessa etapa do algoritmo para uma célula.

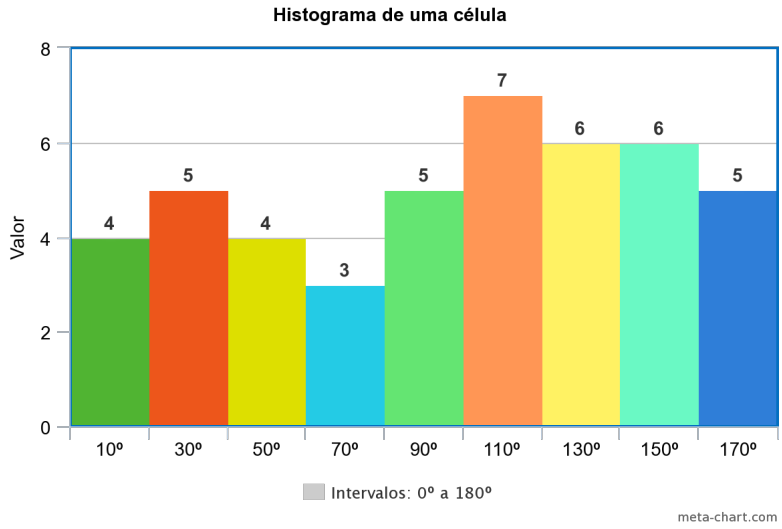


Figura 4.4: Exemplo do histograma de uma célula. Os ângulos mostrados no eixo x são o ponto médio de cada intervalo.

Assim, para cada pixel dentro de uma célula, o valor de $\theta(x,y)$ determina em quais intervalos do histograma aquele pixel deverá contribuir. Para atenuar o efeito de ruídos, os autores que introduziram o HOG recomendam que um pixel contribua não somente para o intervalo b no qual $\theta(x,y)$ está contido , mas também para os dois intervalos adjacentes a ele, $b \pm 1$, com pesos definidos através da interpolação bilinear entre os intervalos vizinhos [12].

Depois que os histogramas de todas as células foram calculados, células adjacentes são agrupadas em blocos que se sobrepõem (ver figura 4.3). Para cada bloco b_i, os histogramas das células que estão contidas em b_i são concatenados, formando um vetor v_i de 36 componentes (9 intervalos por histograma $\times$ 4 células em um bloco), que é normalizado através da normalização L2-Norm, dada por:

$$\text{L2-Norm: } v_{i-norm} = \frac{v_i}{\sqrt[2]{||v_i||^2 + \varepsilon^2}} \tag{4.18}$$

na qual, $||v_i|| = \sqrt[2]{\sum_{j=1}^{36} v_{ij}}$ e $\varepsilon = 0.001$

Finalmente, o vetor de características extraído pelo algoritmo HOG, dado por V_{norm}, é a concatenação dos i vetores v_{i-norm}. Nesse contexto, o tamanho do vetor de características V_{norm}

é definido pela seguinte equação:

$$length(V_{norm}) = \left\lfloor \frac{I_{rows}}{C_{rows}} - 1 \right\rfloor \times \left\lfloor \frac{I_{cols}}{C_{cols}} - 1 \right\rfloor \times B_{cells} \times n_{bins} \tag{4.19}$$

na qual I_{rows} e I_{cols} são a quantidade de linhas e colunas da imagem de entrada, respectivamente; C_{rows} e C_{cols} são a quantidade de linhas e colunas de cada célula; B_{cells} é a quantidade de células dentro de um bloco e n_{bins} é o número de intervalos nos quais o histograma é dividido.

Os parâmetros utilizados nesse trabalho para o algoritmo de HOG estão listados na tabela 4.1. Baseado nesses valores, o tamanho do vetor de características é 1296.

Tabela 4.1: Parâmetros do algoritmo HOG utilizados nesse trabalho

Parâmetro	Símbolo	Valor
Tamanho da célula	$C_{rows} \times C_{cols}$	18×18 pixels
Tamanho do bloco	$B_{rows} \times B_{cols}$ (2×2 células)	36×36 pixels
Tamanho da imagem	$I_{rows} \times I_{cols}$	128×128 pixels
Intervalos do histograma	n_{bins}	9
Quantidade de células dentro de um bloco	B_{cells}	4

Uma vez que o vetor de características da imagem, V_{norm}, foi calculado, a classificação da imagem de entrada deve ser realizada através de alguma técnica robusta de Aprendizagem de Máquina.

Nesse sentido, dado um vetor de características V_{norm}, a próxima etapa do método proposto é determinar à qual das 62 classes de caracteres a imagem de entrada pertence. Para isso, utilizamos a técnica de Aprendizagem de Máquina rede neural Extreme Learning Machine (ELM), que será descrita a seguir.

4.6 Classificador Extreme Learning Machine

Rede Neural Artificial (RNA) é uma técnica robusta de Aprendizagem de Máquina que utiliza minimização da superfície de erro para aprender regras (ajustar os pesos e *biases*) de maneira iterativa. O problema é que esse processo é computacionalmente intensivo, portanto não é a melhor abordagem para aplicações que requerem alto desempenho.

A *Extreme Learning Machine* (ELM) [25] é uma abordagem que surge para resolver esses problemas. A ELM usa um algoritmo de ajuste de parâmetros para uma rede neural com uma camada escondida, que calcula os valores dos pesos e *biases* através de uma simples operação de inversão de matrizes.

A primeira etapa desse algoritmo é definir os pesos e *biases* da camada escondida de maneira aleatória. Então, levando em consideração o conjunto de treinamento, os pesos da camada de saída são definidos através do cálculo da pseudo-inversa de uma matriz. Dessa forma, a ELM é capaz de gerar um ótimo classificador enquanto aprende e classifica milhares de vezes mais rápido que algoritmos de aprendizado tradicionais de redes neurais *feedforward* [25].

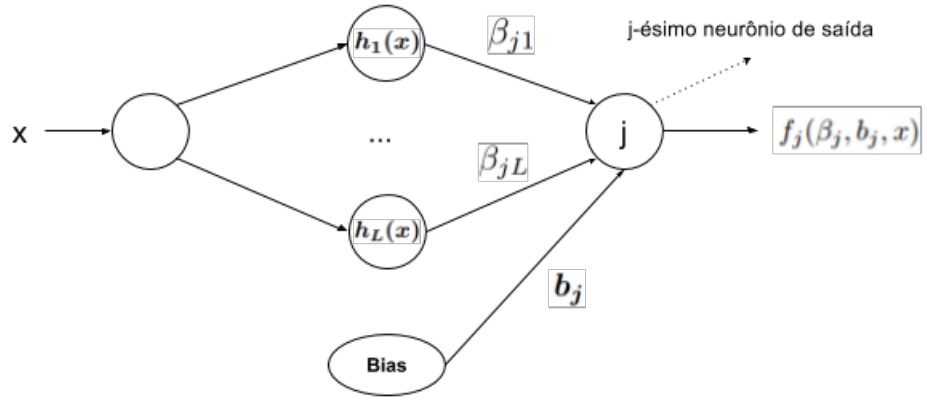


Figura 4.5: Rede neural com uma camada escondida

Portanto, sejam x o vetor de entrada da rede neural, $\beta_j = [\beta_{j1}, ..., \beta_{jL}]$ o vetor de pesos entre a camada escondida e a camada de saída, $h(x) = [h_1(x), ..., h_L(x)]$ o vetor que contém os resultados de todos os neurônios da camada escondida para a entrada $\mathbf{x}$ e b_j o bias do j-ésimo neurônio de saída (ver imagem 4.5), podemos definir o resultado do j-ésimo neurônio de saída como:

$$f_j(\beta_j, b_j, x) = \sum_{i=1}^{L} \beta_{ji} h_i(x) = \beta_\mathbf{j} \mathbf{h}(\mathbf{x}) + \mathbf{b_j} \tag{4.20}$$

Para cada entrada $\mathbf{x}$, o objetivo do algoritmo de treinamento é minimizar o erro entre a saída da rede neural e a saída esperada para aquela entrada $\mathbf{x}$, que está definida no conjunto de treinamento da rede.

Ou seja, o objetivo do algoritmo é encontrar o valor dos pesos da camada de saída, β, que minimizem $||\mathbf{H}\beta - \mathbf{T}||$, na qual $\mathbf{T}$ é uma matriz coluna que contém as saídas esperadas definidas no cojunto de treinamento, portanto seu conteúdo é conhecido a priori e $\mathbf{H}$ é a matriz que contém as saídas dos neurônios da camada escondida. Uma vez que os valores da matriz $\mathbf{H}$ são conhecidos, o algoritmo encontra os pesos da camada escondida através da seguinte equação:

$$\beta = \mathbf{H}^{+}\mathbf{T} \tag{4.21}$$

Os valores de $\mathbf{H}$ são calculados em função do conjunto de treinamento e dos pesos da camada escondida. Nesse contexto, sejam um conjunto de treinamento $\mathcal{N} = \{(x_i, t_i) | x_i \in R^n, t_i \in R^m, i = 1, ..., N\}$ e L neurônios na camada escondida, $\mathbf{b}_{n \times L}$, o vetor de *biases* da camada escondida e $\mathbf{A}_{n \times L}$ a matriz de pesos da camada escondida, cujos valores são definidos aleatoriamente, o cálculo de $\mathbf{H}$ é dado pela seguinte equação:

$$\mathbf{H}(\mathbf{A}, \mathbf{x}) = \begin{bmatrix} f(a_1, b_1, x_1) & \dots & f(a_L, b_L, x_1) \\ \dots & \dots & \dots \\ f(a_1, b_1, x_N) & \dots & f(a_L, b_L, x_N) \end{bmatrix} \tag{4.22}$$

Cada linha da matriz $\mathbf{H}$ representa a saída dos neurônios da camada escondida para um dado vetor de entrada x_i.

Finalmente, o algoritmo da *Extreme Learning Machine* pode ser resumido da seguinte maneira:

Algoritmo 4: Algoritmo de treinamento da ELM

1 **Entrada:** $\mathcal{N} = \{(x_i, t_i) | x_i \in R^n, t_i \in R^m, i = 1, ..., N\}$, um conjunto de treinamento; L, quantidade de neurônios na camada escondida; $g(x)$ uma função de ativação;
2 **Saída:** β, matriz dos pesos da camada de saída
3 Passo 1: Preencha o vetor de *bias* e a matriz de pesos da camada escondida, $\mathbf{b}_{n\times L}$ e $\mathbf{A}_{n\times L}$, respectivamente, com valores aleatórios;
4 Passo 2: Calcule a matriz de saída dos neurônios da camada escondida, $\mathbf{H}$;
5 Passo 3: Calcule a matriz dos pesos da camada de saída, β, através da equação 4.21, na qual $\mathbf{T} = [t_1, ..., t_N]$

Desse modo, os únicos parâmetros que precisam ser ajustados são a quantidade de neurônios na camada escondida, L, e a função de ativação, $g(x)$.

4.7 Resultados da Estratégia Proposta

Nessa seção, avaliamos a taxa de acerto do sistema proposto para o *benchmark* Chars74K-15, com o objetivo de verificar se essa estratégia tem um bom desempenho para reconhecer caracteres em imagens de cenas naturais.

Nesse sentido, implementamos o sistema proposto em C++, utilizando ponto fixo no formato Q1.8.8 para representar os números reais, sob um SO Ubuntu 16.04 LTS, executando em um processador Intel i7-4500U dual-core 3GHz, com 8GB de memória RAM e 1TB de disco rígido. A tabela 4.2 sumariza os detalhes da codificação de cada etapa da estratégia proposta.

Tabela 4.2: Linguagem utilizada e número de linhas de código de cada etapa da estratégia proposta

Tarefa	Linguagem	Linhas
RGB para Grayscale	C++	15
Resize (versão em software)	C++	103
Limiarização de Otsu	C++	78
HOG	C++	335
ELM	C++	487

Como visto na seção anterior, os parâmetros que precisam ser ajustados na ELM são a quantidade de neurônios na camada escondida, L e a função de ativação g. Portanto, realizamos dois experimentos com o objetivo de encontrar o número de neurônios na camada escondida e a função de ativação que resultaram o melhor desempenho da técnica proposta: (1) ELM com função de ativação linear (figura 4.7), ou seja $g(x) = x$, e (2) ELM com função de ativação tangente hiperbólica (figura 4.6), ou seja $g(x) = tanh(x)$, na qual $tanh(x)$ é dado pela equação abaixo:

$$g(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4.23}$$

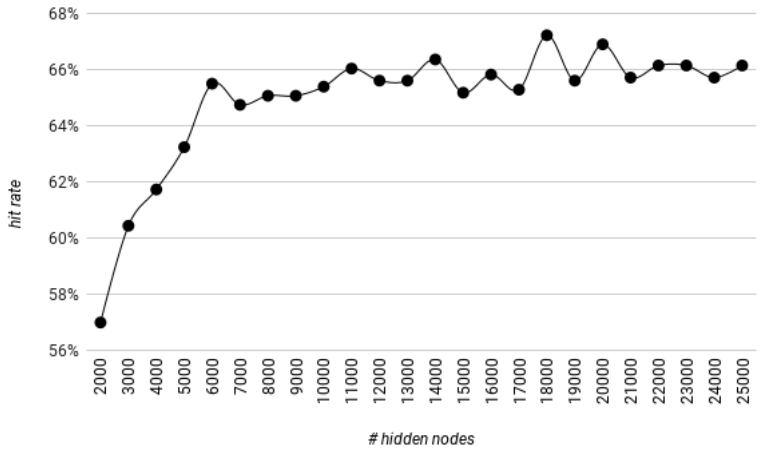


Figura 4.6: Número de neurônios da camada escondida em função da taxa de acerto da rede neural ELM com função de ativação tangente hiperbólica.

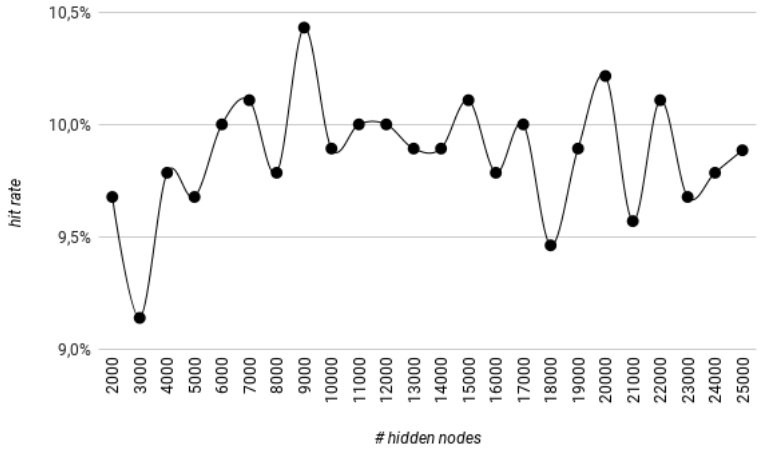


Figura 4.7: Número de neurônios da camada escondida em função da taxa de acerto da rede neural ELM com função de ativação linear.

Os resultados indicam que o melhor desempenho do sistema ocorre quando usamos uma rede neural ELM com 18000 neurônios na camada escondida, $L = 18000$, e função de ativação tangente hiperbólica, $g(x) = tanh(x)$. Nessa configuração, o sistema foi capaz de classificar corretamente até 67.2% das imagens de entrada. Como a taxa de acerto da estratégia proposta é superior às acurácias dos trabalhos recentes na área de STCR descritos no capítulo 3, consideramos que o sistema proposto é um bom classificador para imagens de caracteres em cenas naturais.

Capítulo 5

Arquitetura em Hardware e Software Proposta para STCR

Neste capítulo detalhamos a arquitetura heterogênea de hardware e software proposta para acelerar o algoritmo de STCR descrito no capítulo anterior. Combinamos hardware e software com o objetivo de reduzir o tempo de execução. Adicionalmente, apresentaremos o BIA (*Bicubic Interpolation Accelerator*), um IP-core para FPGA, acelerador do algoritmo de interpolação bicúbica aplicado ao redimensionamento de imagens digitais.

5.1 Visão Geral

O objetivo desse trabalho é desenvolver e avaliar o desempenho de uma arquitetura híbrida de hardware e software (CPU e FPGA) para acelerar o tempo de execução da técnica de reconhecimento de caracteres em cenas naturais (STCR) proposta no capítulo 4. Adicionalmente, esse sistema deve ser prototipado na placa Terasic DE2i-150, uma plataforma embarcada heterogênea composta por um processador Intel ATOM N2600 conectado a um FPGA da família Cyclone IV da Altera via PCI Express.

A primeira etapa para atingir esse propósito é definir quais tarefas devem executar em software (na CPU) e quais devem executar em hardware (na FPGA). Por esse motivo, implementamos o algoritmo de STCR descrito no capítulo anterior em C++ e medimos o tempo médio que cada etapa da aplicação levou para executar no processador Intel Atom N2600, com o objetivo de identificar os *hotspots* (tarefas mais demoradas). Depois, selecionamos a tarefa mais computacionalmente custosa do algoritmo para ser acelerada em hardware.

O procedimento para estimativa do tempo médio de execução foi feito de acordo com o algoritmo 20. Os tempos individuais de cada tarefa foram medidos através da ferramenta de *profiling* Gprof [31] e das funções da biblioteca *time.h*.

O resultado da análise descrita pelo algoritmo 20 (ver figura 5.1) mostra que a tarefa *Resize* é o gargalo da aplicação, consumindo 71.75% do tempo de execução total. Desse modo, desenvolvemos um módulo em hardware, o *Bicubic Interpolation Accelerator* (BIA), para acelerar a tarefa *Resize* e o integramos em uma arquitetura heterogênea CPU-FPGA, com o objetivo de acelerar o algoritmo de STCR proposto no capítulo 4.

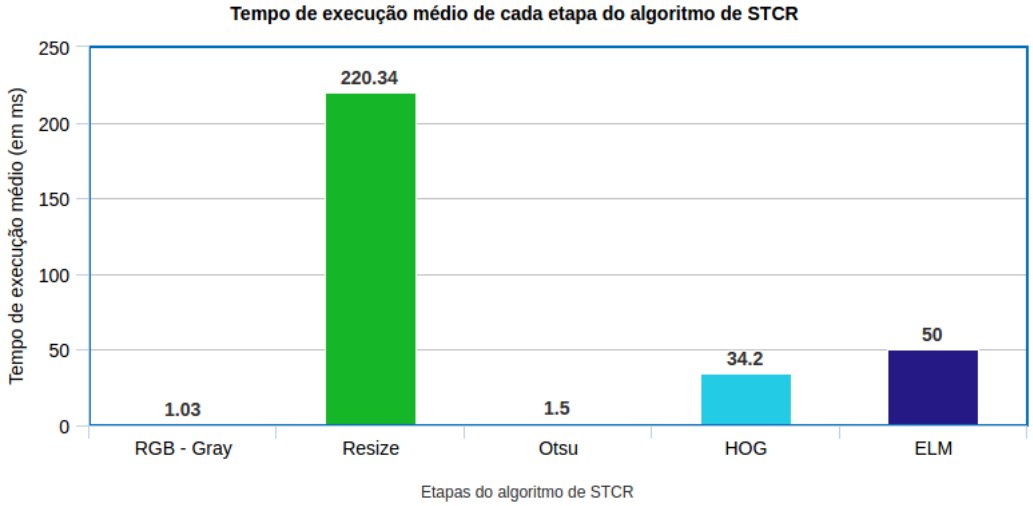


Figura 5.1: Duração média, em milissegundos, de cada etapa do algoritmo de STCR.

Algoritmo 5: Procedimento para medir o tempo de execução das etapas do algoritmo de STCR.

```
1  Entrada: test_set, o conjunto de imagens de teste do dataset Chars74k-15
2  Saída: média do tempo que cada etapa levou para processar todas as 930 imagens de teste
     do dataset Chars74k-15
3  tempo_rgb_to_gray = 0;
4  tempo_resize = 0;
5  tempo_otsu = 0;
6  tempo_hog = 0;
7  tempo_classificador_elm = 0;
8  for each image in test_set do
9      tempo_rgb_to_gray = tempo_rgb_to_gray + tempo(rgb_to_gray());
10     tempo_resize = tempo_resize + tempo(resize());
11     tempo_otsu = tempo_otsu + tempo(otsu());
12     tempo_hog = tempo_hog + tempo(hog());
13     tempo_classificador_elm = tempo_classificador_elm + tempo(classificador_elm());
14 end
15 total_images = length(test_set);
16 tempo_medio_rgb_to_gray = tempo_rgb_to_gray/total_images;
17 tempo_medio_resize = tempo_resize/total_images;
18 tempo_medio_otsu = tempo_otsu/total_images;
19 tempo_medio_hog = tempo_hog/total_images;
20 tempo_medio_classificador_elm = tempo_classificador_elm / total_images;
```

A arquitetura proposta é apresentada na figura 5.2. Ela é composta por um processador embarcado e uma FPGA, que se comunicam através do barramento PCI Express. A figura 5.2 também ilustra o fluxo de execução do algoritmo de STCR nessa arquitetura e a divisão das tarefas entre *hardware* e *software*. As tarefas que executam na CPU foram implementadas como

uma aplicação em software escrita na linguagem C++, enquanto que as tarefas que executam na FPGA foram implementados como módulos em hardware, descritos em System Verilog HDL.

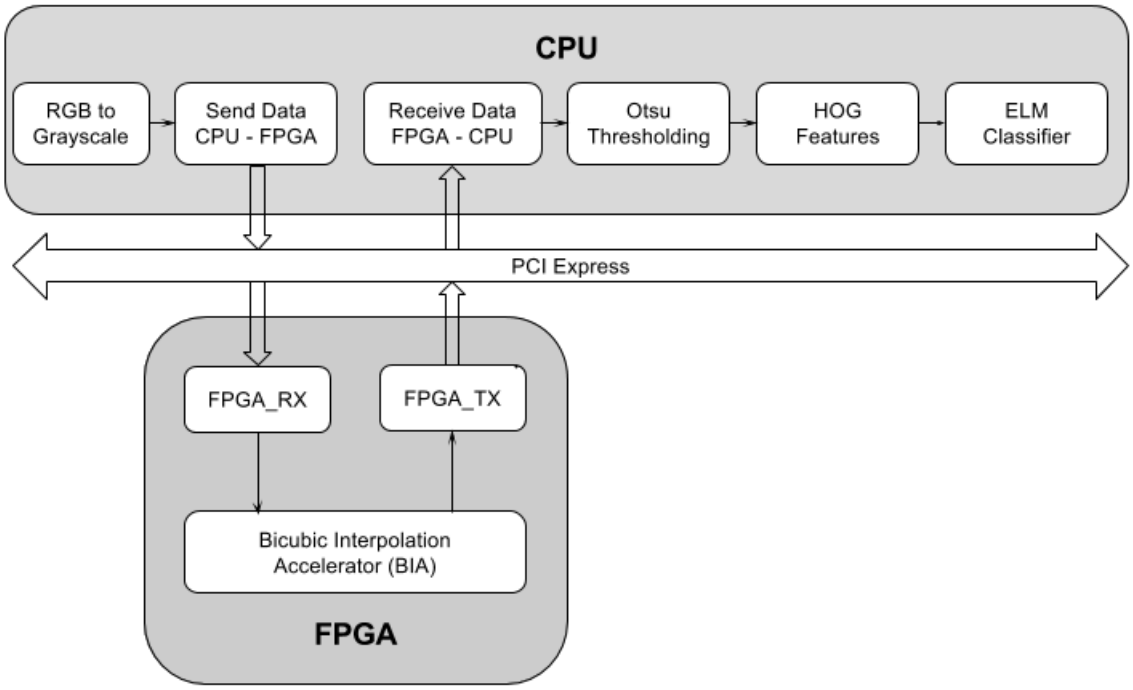


Figura 5.2: Arquitetura proposta.

De acordo com a figura 5.2, o sistema proposto recebe de entrada uma imagem no espaço de cores RGB. Na primeira etapa, RGB to Grayscale, a aplicação em software é responsável por converter a imagem de entrada do espaço de cores RGB para Escala de Cinza. Na segunda etapa, *Send Data CPU - FPGA*, a imagem em *grayscale* tipada é convertida em um stream de bytes ordenados e sem tipo, que é enviado para a FPGA, através do barramento PCI Express.

A próxima fase, FPGA_RX é responsável por controlar as transações de leitura no barramento PCI Express, com o objetivo de receber o stream de bytes enviados pela CPU na etapa *Send Data CPU - FPGA*. Depois, o módulo BIA redimensiona a imagem de entrada para uma imagem de tamanho 128×128 pixels, através do algoritmo de interpolação bicúbica. A imagem redimensionada é enviada de volta para a aplicação em software na etapa FPGA_TX.

Na etapa *Receive Data FPGA - CPU*, o stream de bytes enviados pela FPGA é convertido de volta para uma imagem tipada. Por fim, nas duas últimas fases , o extrator de características baseado em HOG é aplicado à imagem que foi enviada pela FPGA e o classificador baseado em uma rede neural *Extreme Learning Machine* classifica a imagem de entrada em uma dos 62 possíveis caracteres de saída.

Nas próximas seções, detalharemos a comunicação entre software e hardware (especificamente as tarefas *Send data CPU-FPGA*, *Receive data FPGA-CPU* e os módulos FPGA_TX e FPGA_RX) e a arquitetura do módulo BIA. O comportamento das tarefas relacionadas ao processamento da imagem e aprendizagem de máquina executadas pela aplicação em software é o mesmo descrito nas seções do capítulo 4.

5.2 Comunicação CPU-FPGA

A comunicação entre CPU e FPGA foi feita através do framework RIFFA [32]. Do lado do software, o RIFFA oferece uma camada de abstração acima do barramento PCI Express e fornece duas funções principais: $fpga\_send$ e $fpga\_receive$, que recebem como parâmetro um vetor do tipo *unsigned int*, contendo os dados que serão enviados ou recebidos. Do lado do hardware, os dados são fornecidos aos módulos através de FIFOs e a interface com o PCI Express é simplificada através de sinais de controle próprios do protocolo RIFFA.

Nessa seção, são apresentados os detalhes da implementação das tarefas Send Data CPU-FPGA, Receive Data FPGA-CPU, que executam no processador embarcado, e dos módulos $FPGA\_RX$ e $FPGA\_TX$, que executam na FPGA. As figuras 5.3 e 5.4 detalham a arquitetura dos módulos de comunicação.

5.2.1 Comunicação - Software

O protocolo do RIFFA exige que os dados que serão transmitidos à FPGA sejam empacotados em palavras de 32 bits (tamanho, em bits, de cada posição de um vetor do tipo *unsigned int* em C) [32]. Portanto, desenvolvemos a tarefa *Send Data CPU-FPGA* na aplicação em software. Essa tarefa é responsável por formatar a imagem de entrada em um conjunto de palavras de 32 bits de maneira ótima, para enviar os dados para a FPGA, conforme ilustrado na figura 5.3. O funcionamento da tarefa é descrito no algoritmo 6.

De acordo com o algoritmo 6 e a figura 5.3, a tarefa *Send Data CPU - FPGA* recebe como entrada uma imagem tipada em Escala de Cinza de tamanho MxN pixels, que é representada por uma matriz do tipo *integer*. Ou seja, cada pixel é representado por um inteiro de 32 bits.

Nesse sentido, a tarefa *Send Data CPU-FPGA* percorre cada linha dessa matriz, extrai os 8 bits menos significativos de quatro inteiros consecutivos da matriz (que representam 4 pixels consecutivos na imagem de entrada) e empacota esses quatro valores em uma palavra de 32 bits. Como o valor de cada pixel varia entre 0 e 255, um pixel da imagem de entrada pode ser totalmente representado por apenas 8 bits sem que haja perda de informação. Ao final dessa operação, o conjunto de todas as palavras é enviado à FPGA, através da função do driver do RIFFA *fpga_send*.

De modo geral, o procedimento realizado por essa tarefa é uma serialização dos pixels da imagem, ou seja, uma imagem tipada é convertida em um stream de bytes ordenados e sem tipos, que é a representação utilizada para a transmissão de dados entre duas arquiteturas heterogêneas

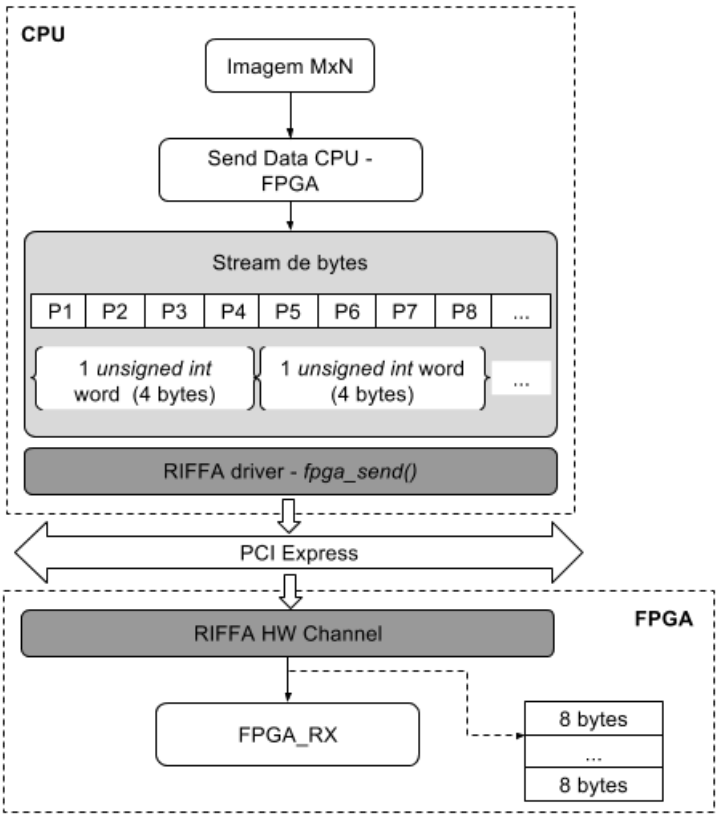


Figura 5.3: Arquitetura da comunicação, referente ao envio de dados da CPU para a FPGA.

através de um canal de comunicação (nesse caso, o barramento PCI Express).

Algoritmo 6: Pseudo-código da tarefa Send Data CPU-FPGA.

```
1  Entrada: I, uma imagem em grayscale de MxN pixels, representada por uma matriz do
     tipo integer de tamanho M × N;
2  buffer ← zeros(0, M*N/4 + 2);
3  shift ← 0;
4  buffer[0] ← M;
5  buffer[1] ← N;
6  buffer_index ← 2;
7  for i ← 0, M − 1 do
8      for j ← 0, N − 1 do
9          buffer[buffer_index] ← buffer[buffer_index] + (I(i, j) << shift);
10         shift ← (shift + 8) & 0x1F;
11         if shift is 0 then
12             buffer_index ← buffer_index + 1;
13         end
14     end
15 end
16 fpga_send(buffer, M*N/4 + 2)
```

De maneira similar à operação de transmissão de dados do CPU para a FPGA, o driver do RIFFA encapsula o stream de bits recebidos da FPGA pelo PCI Express como palavras de 32 bits ordenadas e sem tipo (figura 5.4). Portanto, a tarefa *Receive Data FPGA-CPU* objetiva extrair os valores dos pixels da imagem de saída da FPGA, a partir do stream de bytes recebido. Esse procedimento é detalhado no algoritmo 7.

Algoritmo 7: Pseudo-código da tarefa Receive Data FPGA - CPU.

```
1  Entrada: buffer, um vetor de 4096 palavras de 32 bits;
2  Saída: I, uma imagem em grayscale de 128 × 128 pixels
3  fpga_receive(buffer)
4  shift ← 0;
5  buffer_index ← 0;
6  for i ← 0, 127 do
7      for j ← 0, 127 do
8          I(i, j) ← (buffer[buffer_index] >> shift) & 0xFF;
9          shift ← (shift + 8) & 0x1F;
10         if shift is 0 then
11             buffer_index ← buffer_index + 1;
12         end
13     end
14 end
```

De acordo com o algoritmo 7, dado um array contendo 4096 palavras de 32 bits, a aplicação utiliza *bitwise operators* de C para extrair os 4 pixels contidos em cada palavra de 32 bits, e atribuí-los à posição correspondente da estrutura de dados que representa a imagem no software.

É importante notar que, como as imagens são sempre redimensionadas para o tamanho 128 x 128 pixels, o array de entrada da tarefa *Receive Data FPGA-CPU* sempre terá 4096 palavras de 32 bits.

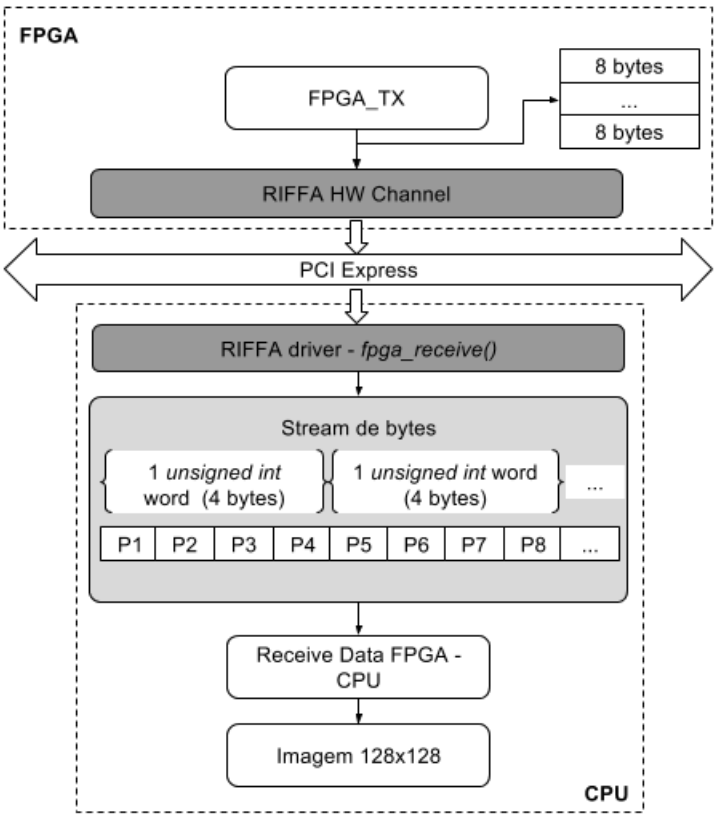


Figura 5.4: Arquitetura da comunicação, referente ao envio de dados da FPGA para a CPU.

5.2.2 Comunicação - Hardware

Do lado do hardware, o framewok RIFFA provê um conjunto de sinais de controle e dados que simplificam as transações no barramento PCI Express.

A largura de dados do barramento entre o processador e o FPGA é determinada pela configuração do link do PCI Express da plataforma-alvo e pelo framework RIFFA. Como utilizamos a placa DE2i-150 para prototipação do sistema proposto, a largura de dados do barramento utilizada nesse trabalho é 64 bits. Consequentemente, 8 pixels (i.e. 64 bits) podem ser recebidos ou enviados a cada ciclo de clock.

Desse modo, desenvolvemos dois módulos em hardware, $FPGA\_RX$ e $FPGA\_TX$, para controlar as transações de leitura e escrita no barramento, respectivamente, de acordo com os sinais do protocolo do framework RIFFA. Esses módulos operam em uma frequência de 125 MHz e realizam a interface da troca de dados entre o canal do RIFFA (RIFFA HW Channel nas imagens 5.3 e 5.4) e o módulo proposto BIA.

A arquitetura de cada um dos módulos $FPGA\_RX$ e $FPGA\_TX$ é composta por uma unidade de controle e um *buffer* do tipo FIFO assíncrono (FIFO_RX e FIFO_TX, respectivamente). As unidades de controle são descritas pelas máquinas de estado ilustradas nas figuras 5.5 e 5.6.

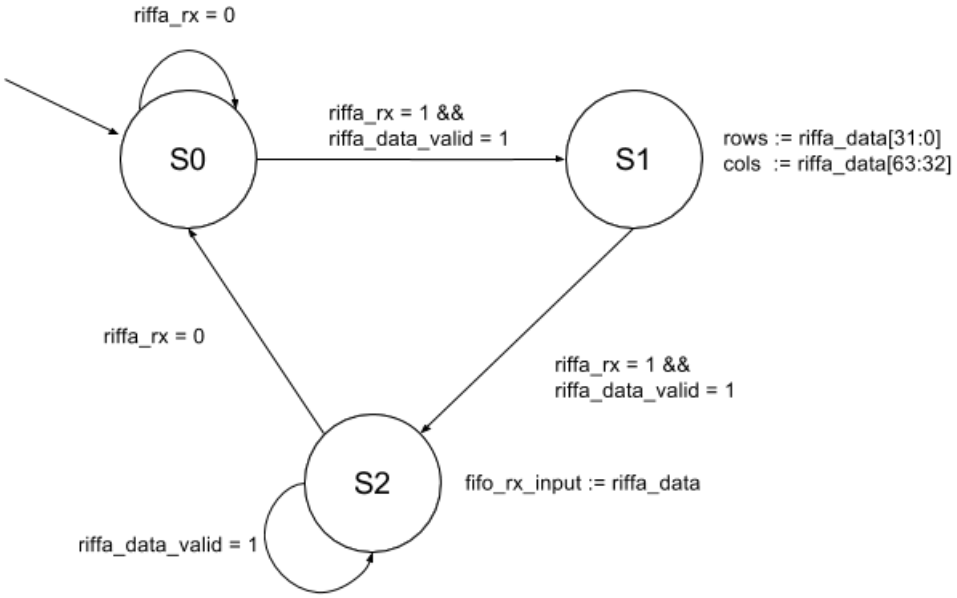


Figura 5.5: Máquina de estado da unidade de controle do módulo FPGA_RX.

FPGA_RX

A descrição de cada estado da unidade de controle do módulo $FPGA\_RX$ é feita a seguir:

- $S0$: o módulo inicia e permanece nesse estado enquanto não houver uma transação válida no barramento ($riffa\_rx = 0$). Quando houver dados válidos no barramento ($riffa\_data\_valid = 1$) provenientes de uma transação válida ($riffa\_rx = 1$), o sistema muda para o estado $S1$.

- $S1$: nesse estado é realizada a leitura da primeira palavra de 64 bits enviada pela aplicação em software (*riffa_data*). Como pode ser observado no algoritmo 6, os 32 bits mais significativos dessa palavra (*riffa_data[63:32]*) correspondem ao número de colunas da imagem de entrada e os 32 bits menos significativos (*riffa_data[31:0]*), correspondem ao número de linhas. Portanto, no estado $S1$, o número de linhas e colunas da imagem original são armazenados em dois registradores de 32 bits.

- $S2$: nesse estado, o módulo realiza a leitura das próximas palavas de 64 bits e as armazena no *buffer* FIFO_RX (*fifo_rx_input := riffa_data*). Cada palavra corresponde a 8 pixels consecutivos da imagem original. Quando o RIFFA indicar que a transação não é mais válida (*riffa_rx=0*), o sistema volta ao seu estado inicial.

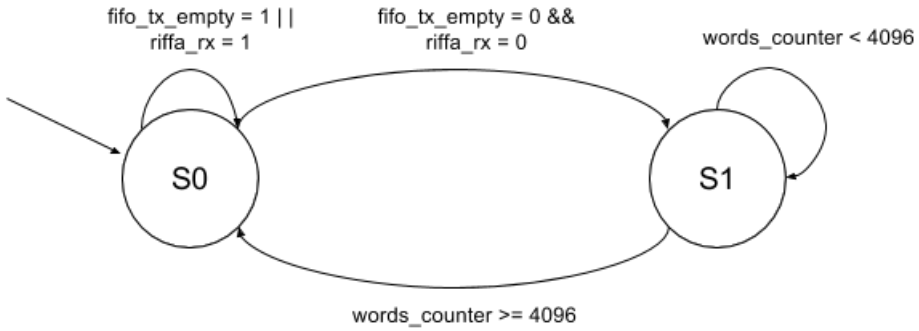


Figura 5.6: Máquina de estado da unidade de controle do módulo FPGA_TX.

FPGA_TX

A descrição de cada estado da unidade de controle do módulo $FPGA\_TX$ é feita a seguir:

- $S0$: o módulo $FPGA\_TX$ inicia nesse estado. Quando não houver mais nenhuma transação do tipo RX válida no barramento ($riffa\_rx = 0$) e se houver dados prontos na $FIFO\_TX$ ($fifo\_tx\_empty = 0$), o sistema está pronto para enviar dados para a CPU.
- $S1$: esse estado é responsável por ler as palavras armazenadas na $FIFO\_TX$ e enviá-las à CPU. Portanto, o módulo permanece nesse estado até que todas as 4096 palavras de 64 bits (todos os 128×128 pixels) forem enviadas com sucesso.

As FIFOs $FIFO\_RX$ e $FIFO\_TX$ utilizadas pelos módulos $FPGA\_RX$ e $FPGA\_TX$, respectivamentes, são necessárias no módulo projetado pelos seguintes motivos:

- **Evitar perda de dados**: uma vez que as transações no barramento são feitas em rajadas, a vazão de transmissão e recepção de dados geralmente não é contínua. Portanto, FIFOs assíncronas são uma boa solução para evitar que os dados se percam devido à inconsistência da vazão.
- ***Clock Domain Crossing:*** se não houver uma interface adequada entre dois sistemas interconectados que operam em frequências ou fases diferentes, o módulo em desenvolvimento pode se tornar assíncrono, levando a problemas como *metastability* e transferência de dados não-confiáveis. Uma solução robusta para esse tipo de problema é fazer a interface entre os módulos através de FIFOs assíncronas.

  Como os módulos $FPGA\_RX$ e $FPGA\_TX$ operam a uma frequência de 125 MHz e o módulo BIA opera a uma frequência 50 MHz, FIFOs são fundamentais para que o sistema não se torne assíncrono.

5.3 O Módulo BIA

O Bicubic Interpolation Accelerator é um IP-core para FPGA, responsável por redimensionar uma imagem de entrada de tamanho qualquer para o tamanho de 128 x 128 pixels, através do algoritmo de interpolação bicúbica.

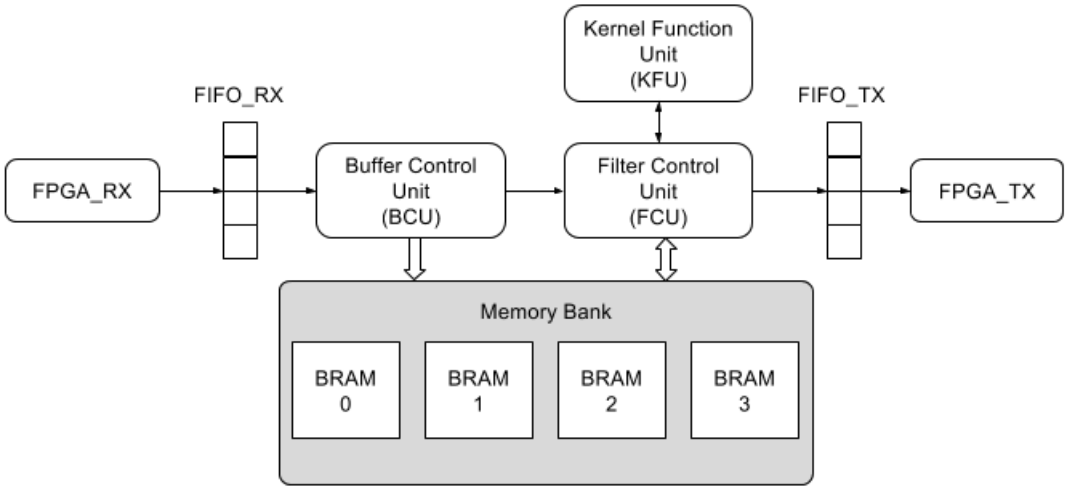


Figura 5.7: Arquitetura do Bicubic Interpolation Accelerator e sua integração com os módulos FPGA_RX e FPGA_TX.

A arquitetura proposta para o módulo BIA (figura 5.7) consiste em três unidades de processamento: (1) Buffer Control Unit (BCU), (2) Filter Control Unit (FCU) e (3) Kernel Function Unit (KFU) e 4 bancos de memória on-chip do tipo BRAM.

Essa arquitetura foi projetada de modo a ser integrada ao processador como uma unidade autônoma (standalone), flexível e capaz de se adaptar a variados tamanhos de imagens de entrada. Todas as unidades de processamento funcionam como estágios de um *pipeline* e são capazes de fornecer um pixel de saída a cada 6 ciclos de um clock de 50 MHz. O BIA explora o paralelismo do algoritmo de interpolação bicúbica processando 4 pixels (dos 16 pixels necessários para computar um pixel de saída) a cada ciclo de clock.

Para facilitar o entendimento de como funciona o módulo BIA, revisaremos o comportamento do algoritmo de interpolação bicúbica a seguir.

De acordo com o algoritmo 8, dada uma imagem de entrada I de tamanho qualquer, o método constrói a imagem de saída iterativamente (linhas 5 e 6), calculando cada pixel de saída, através da equação 5.1.

$$O(i,j) = \sum_{m=-1}^{+2} \sum_{n=-1}^{+2} I(x+m, y+n) r_{cub}(dx-m) r_{cub}(dy-n) \tag{5.1}$$

na qual (i,j) são as coordenadas de um pixel qualquer na imagem de saída, sendo $(x,y) = (\lfloor i * \frac{I_x}{O_x} \rfloor, \lfloor j * \frac{I_y}{O_y} \rfloor)$, calculados nas linhas 7 e 8 do algoritmo, $dx = i * \frac{I_x}{O_x} - x$ e $dy = j * \frac{I_y}{O_y} - y$, calculados nas linhas 12 e 13 do algoritmo e os somatórios indicados na equação 5.1 são

representados pelos laços das linhas 10 e 11.

Algoritmo 8: Redimensionamento de imagens, através do método de interpolação bicúbica

```
1  Entrada: I, a imagem original;
2  I_x e I_y, a quantidade de linhas e colunas da imagem original, respectivamente;
3  O_x e O_y, a quantidade de linhas e colunas da imagem de saída
4  Saída: O, a imagem redimensionada;
5  for i ← 0, O_x do
6      for j ← 0, O_y do
7          x ← ⌊i * I_x/O_x⌋ ;
8          y ← ⌊j * I_y/O_y⌋ ;
9          O(i, j) ← 0 ;
10         for m ← −1 : 2 do
11             for n ← −1 : 2 do
12                 dx ← i * I_x/O_x − y ;
13                 dy ← j * I_y/O_y − x ;
14                 if x + m < I_x and y + n < I_y then
15                     O(i, j) ←
                         O(i, j) + I(x + m, y + n) * r_cub(dx − m)|_a=0.5 * r_cub(dy − n)|_a=0.5
16                 end
17             end
18         end
19     end
20 end
```

Nessa perspectiva e conforme descrito no capítulo 4, o algoritmo de interpolação bicúbica pode ser visto como um processo de filtragem da imagem de entrada em relação a um conjunto de pesos definidos pelo *kernel* do algoritmo de interpolação, r_{cub}. Porém, existem duas diferenças fundamentais que aumentam a complexidade do projeto do módulo de hardware acelerador da interpolação bicúbica:

1. Na interpolação bicúbica os pesos do filtro, ou seja os valores de r_{cub}, são calculados dinamicamente para cada pixel de entrada, portanto não é possível utilizar otimizações baseadas em filtros estáticos. Nesse sentido, desenvolvemos a unidade *Kernel Function Unit*, que é descrina na subseção 5.3.2.

2. Para computar um determinado pixel da imagem de saída, o algoritmo de interpolação bicúbica acessa 16 pixels da imagem de entrada, que consiste na a vizinhança de 4×4 pixels descrita no capítulo 4 e indicada pelos laços das linhas 10 e 11 do algoritmo 8. A partir desse ponto, iremos nos referir à essa região como janela.

   Diferentemente de um processo de filtragem, as coordenadas dos pixels contidos na janela são calculadas dinamicamente para cada pixel de saída, uma vez que elas dependem das dimensões da imagem de entrada e das coordenadas do pixel de saída.

   Portanto, a utilização de mecanismos simples de armazenamento da imagem de entrada

(i.e *shift registers*), amplamente utilizados em operações de filtragem, não são suficientes para um algoritmo de interpolação.

Os detalhes da implementação de cada elemento da arquitetura do módulo BIA são descritos a seguir.

5.3.1 Buffer Control Unit

O Buffer Control Unit é responsável por controlar o armazenamento da imagem de entrada. Os pixels de entrada são armazenados em BRAMs, uma vez que essa estrutura de armazenamento permite acesso aleatório ao seu conteúdo e, por serem memórias *on-chip*, sua latência de leitura é de apenas um ciclo.

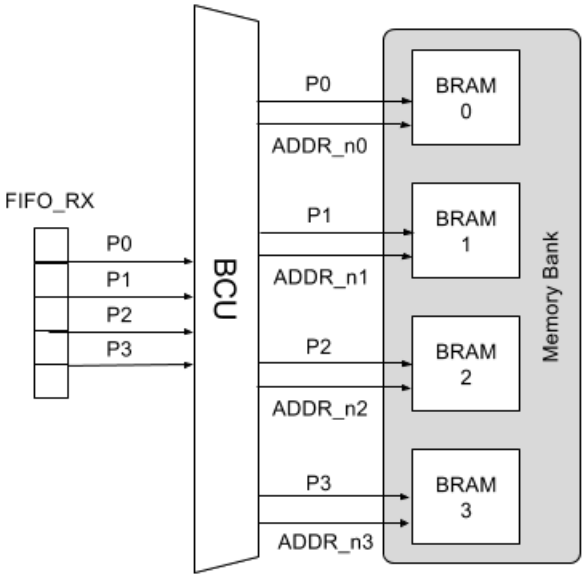


Figura 5.8: Arquitetura do Buffer Control Unit.

A figura 5.8 ilustra a arquitetura do BCU. A cada ciclo de clock, essa unidade recebe de entrada 4 pixels consecutivos da imagem de entrada, e armazena cada pixel em uma BRAM específica, de modo que, seja p_i o i-ésimo pixel da imagem de entrada, p_i é armazenado no endereço $addr_n = i/4$ da memória $BRAM_n$, onde $n = i \mod 4$.

Esse arranjo de armazenamento dos pixels permite que o módulo Filter Control Unit (descrito na subseção 5.3.3) acesse 4 pixels consecutivos da imagem de entrada em paralelo, uma vez que dois ou mais pixels consecutivos nunca estarão armazenados na mesma BRAM (imagem 5.9).

Como 4 pixels consecutivos da imagem de entrada compõem uma linha da janela, utilizada para computar um pixel de saída no método da interpolação bicúbica, o pararelismo do algoritmo pode ser explorado acessando esses 4 pixels em paralelo.

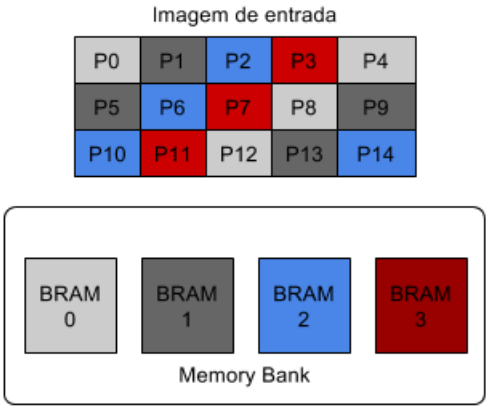


Figura 5.9: Exemplo de um mapeamento dos pixels da imagem de entrada nas BRAMs. Os pixels são armazenados na BRAM com a mesma cor. Os pixels consecutivos nunca estarão armazenados na mesma BRAM.

5.3.2 Kernel Function Unit

A Kernel Function Unit (KFU) é a unidade responsável por calcular os pesos do *kernel* dinamicamente para cada pixel de entrada, de acordo com a equação 4.3, onde $a = 0.5$. Ou seja, dada um valor de entrada (x, y), esse módulo retorna o valor de $r_{cub}(x) * r_{cub}(y)$.

Como pode ser visto no algoritmo 8, os valores de entrada da função r_{cub}, $dx - m$ e $dy - n$, são números reais. Portanto, no módulo em hardware, nós representamos os números no formato de ponto fixo $Q1.8.8$. Ou seja, registradores de 17 bits são utilizados para armazenar valores numéricos, nos quais o primeiro bit mais significativo é usado como bit de sinal, os próximos 8 bits mais significativos representam a parte inteira do número e os 8 bits menos significativos representam a parte fracionária.

Como estamos utilizando uma representação em ponto fixo, $dx - m \in [-2, 2]$ e $dy - n \in [-2, 2]$, todos os possíveis valores de entrada da unidade KFU pertencem ao intervalo inteiro $-512..512$ ($512 = 2 << 8$), portanto, existem apenas 1025 possíveis saídas de r_{cub} (1025 é a quantidade de números inteiros no intervalo $-512..512$). Por essa razão, é sensato utilizar uma *look-up table* (LUT) para implementar a função do *kernel*, $r_{cub}(x)$.

As principais vantagens de se utilizar uma LUT no desenvolvimento dessa unidade são: (1) evitar realizar uma grande quantidade de multiplicações e adições para cada valor de entrada e (2) a saída do módulo fica disponível com a latência de um ciclo de *clock*.

A arquitetura desse módulo é ilustrada na figura 5.10. Como o módulo BIA explora o paralelismo de 4 pixels em uma linha da janela, as entradas da unidade KFU são: (1) 4 pares ordenados, que correspondem aos valores de (dx, dy) para cada pixel e (2) o valor de $m \in [-1, 2]$, que depende de qual linha está sendo processada no momento. O valor de n é conhecido e fixo para todas as entradas, uma vez que estamos processando todos os pixels de uma linha em paralelo.

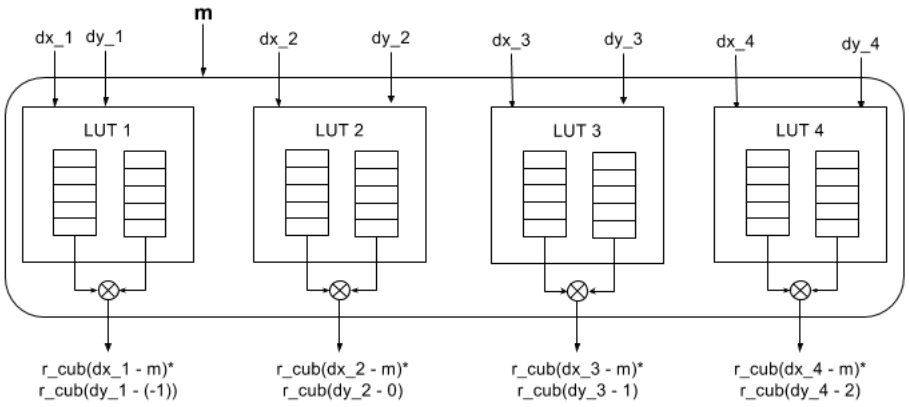


Figura 5.10: Arquitetura do KFU.

5.3.3 Filter Control Unit

A Filter Control Unit (FCU) é a unidade responsável pelas seguintes tarefas:

1. Calcular, para cada pixel de saída, as coordenadas dos pixels pertencentes à janela;
2. Ler os pixels da memória BRAM, baseado nas coordenadas calculadas em (1);
3. Calcular os valores de entrada da Kernel Function Unit, dx, dy e m;
4. Multiplicar os pixels da janela pelos pesos do kernel, para gerar o pixel de saída.

No projeto dessa unidade, nós exploramos o paralelismo do hardware computando uma linha (i.e. 4 dos 16 pixels) da janela de 4×4 pixels em um ciclo de *clock* (figura 5.11).

O funcionamento da FCU é descrito pela máquina de estado da figura 5.12. Ela descreve a leitura dos pixels da imagem de entrada, ou seja, como a FCU controla o acesso às memórias BRAMs e como é feito o controle da multiplicação entre os pixels lidos e os pesos do kernel correspondentes. A descrição de cada estado é feita a seguir:

1. IDLE: no estado inicial, o sistema calcula as coordenadas do centro da janela, x e y, e os valores de dx e dy (a maneira como esses valores são calculadas está descrita no algoritmo 8). O sistema permanece nesse estado enquanto as BRAMs ainda não tiverem armazenado os pixels da janela centrada em (x,y) (*bram_has_window_stored = 0*), necessários para computar o pixel de saída;
2. START: o sistema solicita às BRAMs a leitura dos 4 pixels da primeira linha da janela e solicita ao submódulo KFU os pesos do kernel correspondentes a essa linha ($m = -1$). Quando esses dados estiverem disponíveis (*data_bram_ready_row1 == 1*), o sistema muda para o estado ROW1.
3. ROW1: os pixels da primeira linha da janela são multiplicados pelos pesos do kernel correspondentes. Adicionalmente, o sistema solicita às BRAMs a leitura dos 4 pixels da segunda linha da janela e solicita ao submódulo KFU os pesos do kernel correspondentes a essa linha ($m = 0$). Quando esses dados estiverem disponíveis, (*data_bram_ready_row2 == 1*), o sistema muda para o estado ROW2;

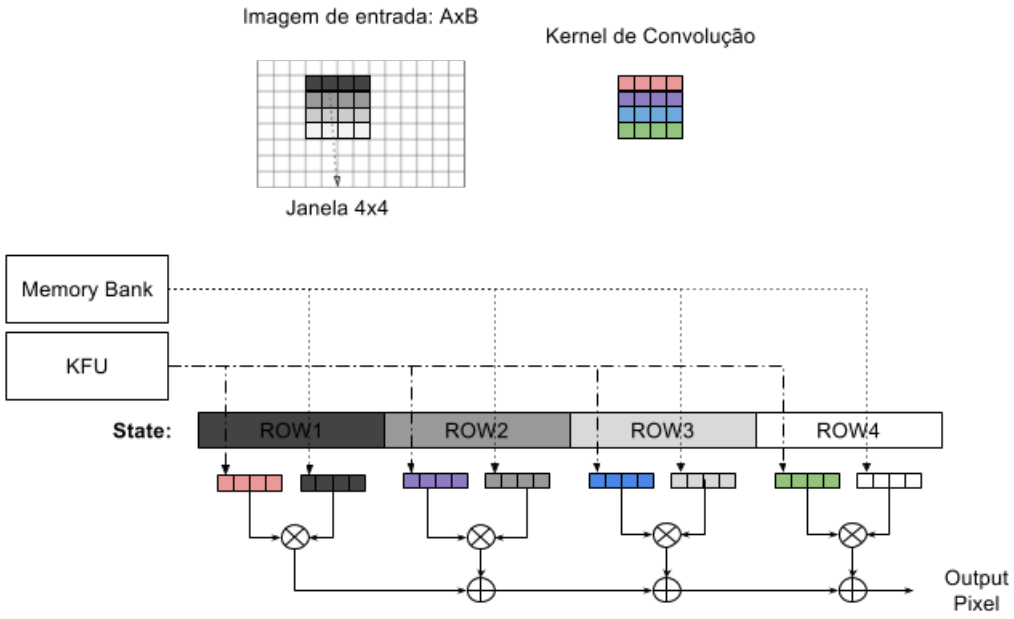


Figura 5.11: Paralelismo do algoritmo de interpolação bicúbica explorado pelo FCU: a cada ciclo de clock, 4 pixels da janela são processados em paralelo.

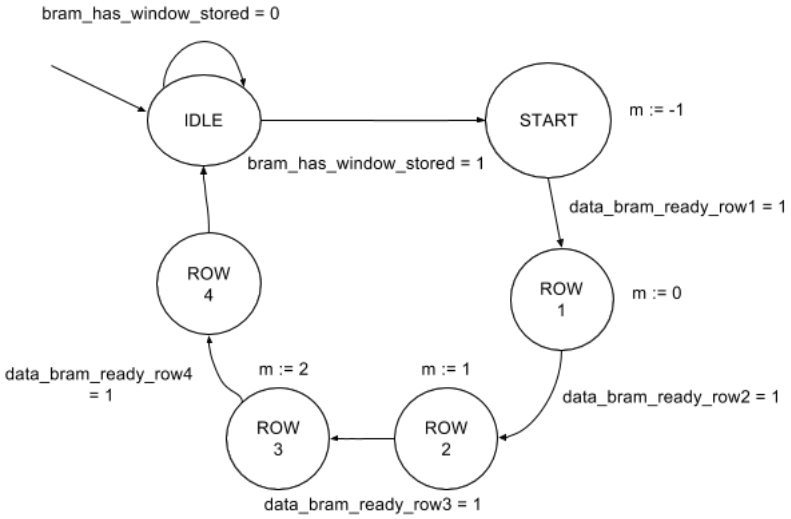


Figura 5.12: Máquina de estados da FCU.

4. ROW2 e ROW3: de maneira similar ao estado anterior, os pixels da segunda e terceira linhas da janela, respectivamente, são multiplicados pelos pesos do kernel correspondentes ($m = 1$ e $m = 2$, os valores de m para a segunda linha e terceira linha, respectivamente). Adicionalmente, os pixels da terceira e quarta linhas e os próximos pesos do kernel são solicitados;

5. ROW4: os pixels da quarta linha da janela são multiplicados pelos pesos do kernel correspondentes e a coordenada (i, j) do pixel de saída é atualizada para $(i, j+1)$ ou $(i+1, 0)$, se o pixel de saída atual for o pixel da última coluna. No próximo estado (IDLE), um pixel de saída válido estará disponível.

5.4 Implementação do módulo BIA

O fluxo geral do projeto do módulo BIA é ilustrado na figura 5.13. Nesse sentido, o desenvolvimento do módulo partiu de um modelo de referência, escrito em C++, do algoritmo de interpolação bicúbica aplicado ao redimensionamento de imagens digitais. Esse modelo de referência foi previamente validado, durante o processo de desenvolvimento do sistema de STCR em software. Portanto, pode-se assumir que ele atende aos requisitos funcionais e não-funcionais do projeto.

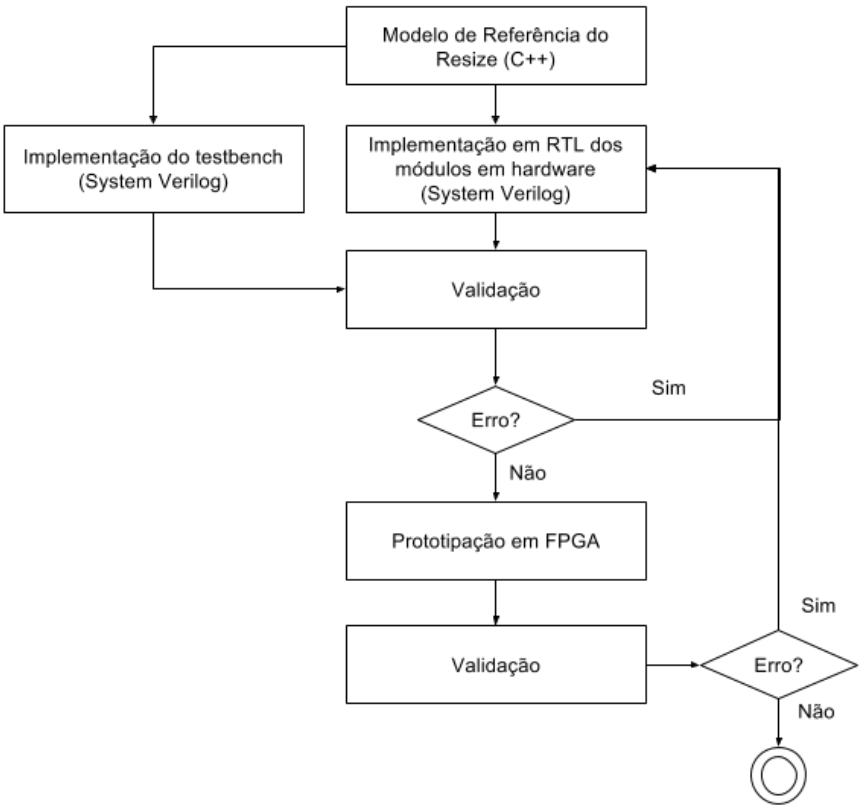


Figura 5.13: Fluxo geral do projeto do módulo BIA.

Em seguida, codificamos os módulos em hardware, FPGA_RX, FPGA_TX e BIA, em nível de abstração RTL (transferência de registrador), através da linguagem System Verilog. Para tornar o sistema flexível, facilitar os testes e tornar a codificação mais simples, dividimos o módulo BIA em três submódulos: Buffer Control Unit, Kernel Function Unit e Filter Control Unit. A tabela 5.1 sumariza os detalhes da codificação dos módulos em hardware e das tarefas *Send Data CPU - FPGA* e *Receive Data FPGA - CPU*.

Tabela 5.1: Linguagem utilizada e número de linhas de código de cada módulo em hardware e para as tarefas *Send Data CPU - FPGA e Receive Data FPGA - CPU*

Tarefa	Linguagem	Linhas
Send Data CPU - FPGA	C++	45
Receive Data FPGA - CPU	C++	32
FPGA_RX	System Verilog	123
FPGA_TX	System Verilog	110
Buffer Control Unit	System Verilog	50
Kernel Function Unit	System Verilog	1745
Filter Control Unit	System Verilog	407

Em paralelo à implementação dos módulo em hardware, desenvolvemos um *testbench*, com o objetivo de verificar se a implementação em RTL dos módulos de hardware estavam de acordo com a especificação, que nesse caso é o modelo de referência em software.

O *testbench*, cujo modelo é ilustrado na figura 5.14, foi codificado em System Verilog e foram escritas 183 linhas de código.

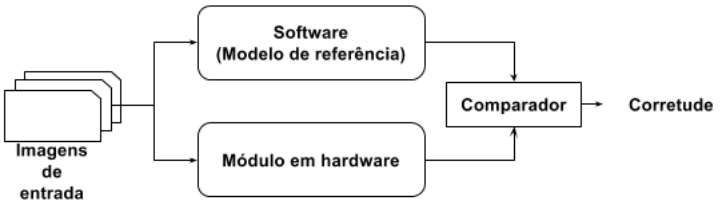


Figura 5.14: Modelo de *testbench* utilizado para validar o módulo em hardware implementado.

No modelo da figura 5.14, o estímulo de entrada são imagens que foram retiradas do benchmark Chars74K-15 e convertidas de RGB para Grayscale de acordo com o algoritmo de conversão de espaço de cores descrito no capítulo 4).

Depois de convertidas, as imagens foram enviadas em paralelo aos módulos de hardware e ao modelo de referência em software (a implementação em C++ da tarefa *Resize* utilizando ponto fixo). Nessa etapa, os módulos de hardware foram simulados, através de simulações funcionais e com timing, no ambiente *ModelSim*.

Por fim, as saídas dos módulos em software e hardware foram comparadas. Esse procedimento de validação foi realizado de maneira iterativa até que não houvesse erro em nenhuma saída. Nesse momento, consideramos que o módulo de hardware foi validado por simulação.

Na próxima etapa, o módulo foi validado através da prototipação em FPGA. O procedimento de validação realizado nessa fase é similar ao procedimento descrito na fase anterior. As duas principais diferenças são: (1) nessa etapa, utilizamos uma aplicação em C++ para gerar os estímulos de entrada e para comparar a saída do módulo em hardware com o modelo de referência em software, ao invés de um *testbench* em System Verilog e (2) os módulos em hardware não foram simulados, mas executaram diretamente na FPGA. Ao final dessa etapa, o módulo foi completamente validado.

A tabela 5.2 sumariza o resultado da etapa de síntese dos módulos de hardware. Os módulos em FPGA ocupam apenas 11% dos elementos lógicos da Cyclone IV. A memória é o recurso mais utilizado, uma vez que 4 BRAMs de 81000 palavras de 8 bits são utilizadas para armazenar a imagem de entrada (portanto, a arquitetura suporta imagens de entrada de até 324000 pixels).

Tabela 5.2: Percentual da utilização de recursos da FPGA Cyclone IV pelos módulos em hardware

Recurso	Utilização
Elementos lógicos	11%
Bits de memória	52%
Multiplicadores de 9-bits	12%
PLLs	25%
Potência estimada (TDP)	1 W

Capítulo 6

Resultados e Experimentos

Nesse capítulo, abordamos as ferramentas, experimentos e cenários que utilizamos para avaliar o tempo de execução e a acurácia do sistema de reconhecimento de caracteres em cenas naturais proposto no capítulo 4 e da arquitetura heterogênea CPU-FPGA proposta no capítulo 5.

6.1 Configuração dos Experimentos

O desempenho das soluções propostas foram avaliadas, baseadas na taxa de reconhecimento (acurácia) e no tempo de execução. Portanto, realizamos experimentos em duas configurações diferentes:

1. Na primeira configuração, executamos o sistema de reconhecimento de caracteres proposto no capítulo 4 apenas em software sob um Ubuntu 14.04, executando no processador da placa DE2i-150, um Intel ATOM N2600 dual-core 1.6GHz, com 2GB de memória RAM e 60GB de disco rígido da placa DE2i-150. O sistema foi implementado em C++. Para realizar uma comparação justa com o sistema híbrido proposto, a etapa *Resize* da implementação em software também foi implementada usando ponto fixo de formato Q1.8.8 e os valores de $r_{cub}(x)$ foram calculados através de uma *hash table*, para simular uma *look-up table*.

2. Na segunda configuração, executamos o sistema híbrido proposto no capítulo 5 na placa DE2i-150. A parte em software foi escrita em C++ e é executada no processador placa. Já a parte em hardware foi escrita em System Verilog e é executada na FPGA da placa, uma Cyclone IV EP4CGX150DF31. A comunicação entre o processador e a FPGA é feita através de um barramento x1 Gen PCI Express. Além disso, utilizamos o framework RIFFA [32] para realizar a interface entre a FPGA e a CPU.

Ambas as configurações foram avaliadas usando o *benchmark* Chars74K-15 [1], descrito no capítulo 2. Ele contém 930 imagens de treinamento e 930 imagens de teste, divididas em 62 classes. Portanto, em todos os testes subsequentes, é assumido que foi utilizado o *dataset* Chars74K-15.

6.1.1 Parâmetros da *Extreme Learning Machine*

Como visto no capítulo 4, os parâmetros da ELM que garantem o melhor desempenho (67.2% de acurácia) da estratégia de reconhecimento de caracteres em cenas naturais proposta são 18000 neurônios na camada escondida e função de ativação tangente hiperbólica. Porém, uma quantidade tão grande de neurônios na camada escondida pode penalizar o tempo de resposta do sistema, quando ele está executando em uma plataforma embarcada, como a placa DE2i-150.

Portanto, geramos o gráfico ilustrado na figura 6.1, para avaliar o impacto que a quantidade de neurônios na camada escondida exerce sobre o tempo médio que a rede neural ELM leva para classificar uma imagem do conjunto de teste do *benchmark* Chars74K-15. Nesse experimento, a função de ativação é a tangente hiperbólica.

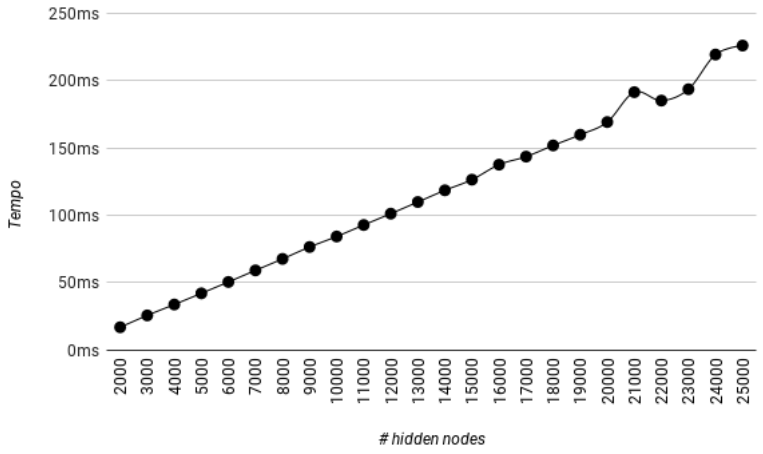


Figura 6.1: Número de neurônios da camada escondida em função do tempo médio que a rede neural ELM leva para classificar uma imagem do conjunto de teste.

De acordo com os resultados indicados no gráfico da figura 6.1, o tempo de classificação da rede neural ELM cresce de maneira aproximadamente linear com o número de neurônios na camada escondida.

Portanto, com o objetivo de manter um compromisso entre baixo tempo de execução e alta taxa de acerto, optamos por utilizar 6000 neurônios na camada escondida. Com essa configuração, o sistema classifica corretamente até 65.5% das imagens de entrada, resultado não somente superior aos resultados dos trabalhos recentes da área discutidos no capítulo 3, como também apenas 1.7 pontos percentuais mais baixo que a taxa de acerto considerando 18000 neurônios na camada escondida. Além disso, nessa configuração o tempo médio de classificação de uma imagem (50.7 ms) é 3 vezes menor que o tempo médio de classificação de uma rede neural ELM com $L = 18000$ (151.9ms).

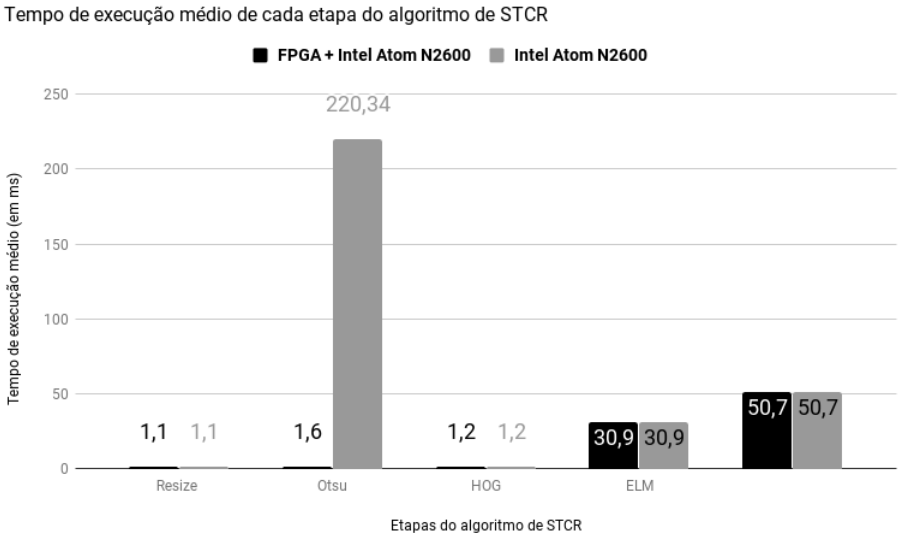


Figura 6.2: Duração média, em milissegundos, de cada etapa do algoritmo de STCR na arquitetura híbrida proposta em comparação com a abordagem apenas em software

6.2 Performance

6.2.1 Profiling

Para avaliar o tempo médio de execução da arquitetura híbrida para o modelo de STCR proposto, executamos o procedimento descrito no algoritmo 20 do capítulo 5. Os tempos individuais de cada tarefa foram medidos através da ferramenta de *profiling* Gprof [31] e funções da biblioteca *time.h*. Estimamos o tempo médio de execução da tarefa *Resize*, através da diferença entre o instante de tempo imediatamente anterior à tarefa *Send Data CPU - FPGA* e o instante de tempo imediatamente posterior à tarefa *Receive Data CPU - FPGA*. Portanto, o tempo indicado no gráfico da figura 6.2 relativo à tarefa *Resize* é o tempo da comunicação entre CPU e FPGA somado ao tempo de execução do BIA.

O resultado ilustrado na figura 6.2 indica que o módulo BIA é capaz de redimensionar imagens através do algoritmo de interpolação bicúbica até 137 vezes mais rápido que a implementação em C++ (ver imagem 5.1).

6.2.2 Comparação com Outros Métodos

Utilizando os parâmetros da ELM definidos na subseção 6.1.1 e os parâmetros do HOG, definidos na tabela 4.1, o sistema proposto foi comparado com trabalhos relevantes na área de reconhecimento de caracteres que foram avaliados na base de dados Chars74K-15.

Tabela 6.1: Comparação dos métodos de STCR no dataset Chars74K-15.

Método	Tempo de Execução (s)	Arquitetura	Acurácia (%)

SIFT+SVM [1]	-	-	21.4
GB+SVM [1]	-	-	52.6
MKL[1]	-	-	55.8
Tensor + NN [33]	-	-	57.1
Rank1 Tensor + I2CDML [2]	-	-	59.0
Keypoints + HOG + Adaboost [4]	1	ARM Cortex-A9 + GPU GeForce ULP	60.0
PCANet [3]	1	CPU	64.0
Proposto (configuração 1)	0.308	Intel Atom N2600	**65.5**
Proposto (configuração 2)	**0.085**	Intel Atom N2600 + FPGA	**65.5**

A tabela 6.1 indica que o trabalho proposto tem taxa de acerto superior à dos trabalhos comparados, ao mesmo tempo em que é capaz de processar até 11 *frames* por segundo. A arquitetura híbrida CPU-FPGA (configuração 2) é 3.6 vezes mais rápida que a implementação apenas em software (configuração 1) e até 11.7 vezes mais rápida que os métodos propostos em [4] e [3].

6.2.3 Ponto Fixo

Como utilizamos ponto fixo na representação dos números reais na etapa *Resize*, é normal que a acurácia do sistema seja diferente da implementação que utiliza ponto flutuante de dupla precisão (*double*). Para avaliar o impacto do uso de ponto fixo na acurácia do sistema, adicionamos ao gráfico ilustrado na figura 4.6 a curva equivalente à acurácia do sistema utilizando ponto flutuante de dupla precisão. O resultado, ilustrado na figura 6.3, indica que a precisão do sistema não teve impacto significativo com o uso de ponto fixo e que, na maioria dos casos, a precisão do sistema com ponto fixo foi superior à precisão do sistema com ponto flutuante.

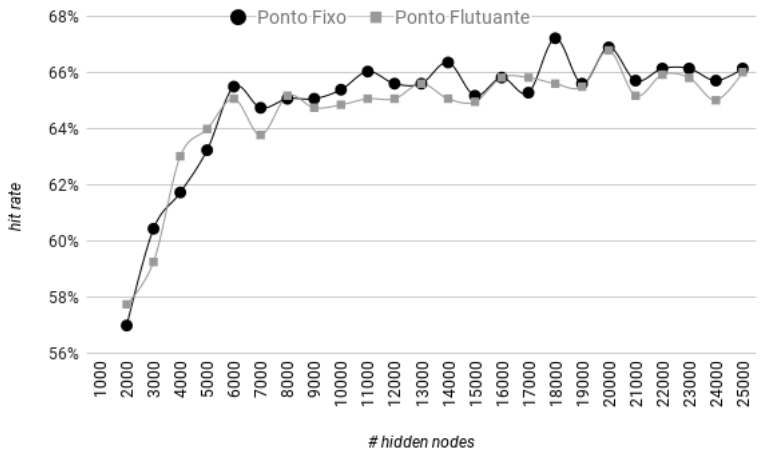


Figura 6.3: Número de neurônios da camada escondida em função da taxa de acerto da rede neural ELM com função de ativação tangente linear. A curva do sistema que utiliza ponto flutuante é indicada por quadrados, enquanto que o sistema que utiliza ponto fixo é indicado por círculos.

Capítulo 7

Conclusão

Nesse trabalho, apresentamos uma nova técnica de visão computacional baseado em *Histogram of Oriented Objects* (HOG) e redes neurais *Extreme Learning Machine* (ELM), para reconhecimento de caracteres em cenas naturais (STCR) e uma arquitetura híbrida CPU-FPGA para acelerar essa aplicação. O sistema foi prototipado na plataforma embarcada Terasic DE2i-150, que é composta por um processador embarcado conectado a uma FPGA através do barramento PCI Express.

Inicialmente, a técnica proposta transforma as imagens do espaço de cores RGB para o espaço de cores Grayscale. Em seguida, todas as imagens são redimensionadas para um tamanho fixo de 128×128 pixels, com o objetivo de padronizar as imagens de entrada e aumentar a variabilidade dos dados. Para extrair as características da imagem, a aplicação usa HOG, que gera um vetor com 1296 *features*. Finalmente, utilizamos uma rede neural ELM para classificar as imagens, baseada no vetor de características.

Essa técnica é capaz de atingir acurácia de 65.5% no dataset Chars74K-15, resultado superior aos resultados obtidos nos trabalhos comparados nesse documento. Para acelerar o tempo de execução da implementação em C++ do método proposto, realizamos um *profiling* e identificamos que o *hotspot* da aplicação era o redimensionamento da imagem.

Assim, apresentamos nesse documento o Bicubic Interpolation Accelerator (BIA), um IP-Core em FPGA, acelerador do algoritmo de Interpolação Bicúbica para redimensionamento de imagens digitais. O módulo funciona como uma unidade autônoma (standalone), flexível e capaz de se adaptar a variados tamanhos de imagens de entrada. Todas as unidades de processamento funcionam em pipeline e são capazes de fornecer um pixel de saída a cada 6 ciclos de clock de 50 MHz. O BIA explora o paralelismo do algoritmo de interpolação bicúbica processando 4 pixels (dos 16 pixels necessários para computar um pixel de saída) a cada ciclo de clock.

Adicionalmente, mostramos uma arquitetura híbrida CPU-FPGA que utiliza o BIA para acelerar o algoritmo de STCR proposto. O sistema heterogêneo é até 11.7 vezes mais rápido que os trabalhos comparados, é capaz de processar até 11 *frames* por segundo, ocupa apenas 11% da área da FPGA Cyclone IV. Portanto, é viável sua utilização em sistemas embarcados com restrições de tempo e área.

Embora tenhamos acelerado o algoritmo de STCR proposto, seria interessante desenvolver mais aceleradores em FPGA para as outras partes mais custosas do algoritmo, como para a Rede Neural ELM e o método HOG. Trabalhos futuros também podem integrar o sistema de STCR

proposto a um sistema de *Scene Text Localization*, para acelerar o reconhecimento de texto em cenas naturais fim-a-fim. Por fim, como o BIA é uma unidade bastante flexível, ele pode ser utilizado para acelerar qualquer sistema de visão computacional que utiliza redimensionamento de imagens.

Referências Bibliográficas

[1] T. E. de Campos, B. R. Babu, and M. Varma, "Character Recognition in Natural Images," *Visapp (2)*, pp. 273–280, 2009.

[2] M. Ali and H. Foroosh, "Character recognition in natural scene images using rank-1 tensor decomposition," in *Image Processing (ICIP), 2016 IEEE International Conference on*, pp. 2891–2895, IEEE, 2016.

[3] C. Chen, D.-H. Wang, and H. Wang, "Scene character recognition using PCANet," *Proceedings of the 7th International Conference on Internet Multimedia Computing and Service - ICIMCS '15*, pp. 1–4, 2015.

[4] C. Yi and Y. Tian, "Scene text recognition in mobile applications by character descriptor and structure configuration," *IEEE Transactions on Image Processing*, vol. 23, no. 7, pp. 2972–2982, 2014.

[5] X. Rong, C. Yi, and Y. Tian, "Recognizing Text-based Traffic Guide Panels with Cascaded Localization Network," *European Conference on Computer Vision*, pp. 1–14, 2016.

[6] A. Coates, B. Carpenter, C. Case, S. Satheesh, B. Suresh, T. Wang, D. J. Wu, and A. Y. Ng, "Text Detection and Character Recognition in Scene Images with Unsupervised Feature Learning," *International Conference on Document Analysis and Recognition*, 2011.

[7] M. Aggravi, A. Colombo, D. Fontanelli, A. Giannitrapani, D. Macii, F. Moro, P. Nazemzadeh, L. Palopoli, R. Passerone, D. Prattichizzo, and Others, "A Smart Walking Assistant for Safe Navigation in Complex Indoor Environments," in *Ambient Assisted Living*, pp. 487–497, Springer, 2015.

[8] K. Sanni, G. Garreau, J. L. Molin, and A. G. Andreou, "FPGA implementation of a Deep Belief Network architecture for character recognition using stochastic computation," in *2015 49th Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–5, 2015.

[9] H. Zho, G. Zhu, and Y. Peng, "A RMB optical character recognition system using FPGA," in *2016 IEEE International Conference on Signal and Image Processing (ICSIP)*, pp. 539–542, 2016.

[10] Terasic, "DE2i-150 FPGA Development Kit." Website: http://www.terasic.com.tw, 2016.

[11] J. Fan, L. Haotian, B. Bing, and Z. Xiaojin, "Automatic IC Character Recognition System for IC Test Handler Based on SVM," in *Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2016 8th International Conference on*, vol. 2, pp. 239–242, IEEE, 2016.

[12] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, pp. 886–893, IEEE, 2005.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[14] H. Hou and H. Andrews, "Cubic splines for image interpolation and digital filtering," *IEEE Transactions on acoustics, speech, and signal processing*, vol. 26, no. 6, pp. 508–517, 1978.

[15] R. Keys, "Cubic convolution interpolation for digital image processing," *IEEE transactions on acoustics, speech, and signal processing*, vol. 29, no. 6, pp. 1153–1160, 1981.

[16] C. Chen, D.-H. Wang, and H. Wang, "Scene character and text recognition: The state-of-the-art," in *International Conference on Image and Graphics*, pp. 310–320, Springer, 2015.

[17] C. Yi, X. Yang, and Y. Tian, "Feature representations for scene text character recognition: A comparative study," in *Document Analysis and Recognition (ICDAR), 2013 12th International Conference on*, pp. 907–911, IEEE, 2013.

[18] Z. R. Tan, S. Tian, and C. L. Tan, "Using pyramid of histogram of oriented gradients on natural scene text recognition," in *Image Processing (ICIP), 2014 IEEE International Conference on*, pp. 2629–2633, IEEE, 2014.

[19] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Deep Features for Text Spotting.," in *ECCV (4)*, pp. 512–528, 2014.

[20] A. Coates, B. Carpenter, C. Case, S. Satheesh, B. Suresh, T. Wang, D. J. Wu, and A. Y. Ng, "Text detection and character recognition in scene images with unsupervised feature learning," in *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pp. 440–445, IEEE, 2011.

[21] C. Shi, C. Wang, B. Xiao, Y. Zhang, S. Gao, and Z. Zhang, "Scene text recognition using part-based tree-structured character detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2961–2968, 2013.

[22] C. Shi, C. Wang, B. Xiao, S. Gao, and J. Hu, "End-to-end scene text recognition using tree-structured models," *Pattern Recognition*, vol. 47, no. 9, pp. 2853–2866, 2014.

[23] D. G. Bailey, *Design for embedded image processing on FPGAs*. John Wiley & Sons, 2011.

[24] N. Otsu, "A Threshold Selection Method from Gray-Level Histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, pp. 62–66, jan 1979.

[25] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: theory and applications," *Neurocomputing*, vol. 70, no. 1, pp. 489–501, 2006.

[26] C. Rafael Gonzalez and R. Woods, "Digital image processing," *Pearson Education*, 2002.

[27] W. Burger and M. J. Burge, *Digital image processing: an algorithmic introduction using Java*. Springer, 2016.

[28] O. Déniz, G. Bueno, J. Salido, and F. la Torre, "Face recognition using histograms of oriented gradients," *Pattern Recognition Letters*, vol. 32, no. 12, pp. 1598–1603, 2011.

[29] M. W. Tahir, N. A. Zaidi, R. Blank, P. P. Vinayaka, M. J. Vellekoop, and W. Lang, "Detection of fungus through an optical sensor system using the histogram of oriented gradients," in *SENSORS, 2016 IEEE*, pp. 1–3, IEEE, 2016.

[30] W.-K. Tsai, S.-K. Lo, C.-D. Su, and M.-H. Sheu, "Vehicle Detection Algorithm Based on Modified Gradient Oriented Histogram Feature," in *Advances in Intelligent Information Hiding and Multimedia Signal Processing: Proceeding of the Twelfth International Conference on Intelligent Information Hiding and Multimedia Signal Processing, Nov., 21-23, 2016, Kaohsiung, Taiwan, Volume 2*, pp. 127–134, Springer, 2017.

[31] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*, vol. 17, pp. 120–126, ACM, 1982.

[32] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 4, p. 22, 2015.

[33] M. Ali and H. Foroosh, "Natural Scene Character Recognition Without Dependency on Specific Features.," in *VISAPP (2)*, pp. 368–376, 2015.