Universidade Federal de Pernambuco
Centro de Informática

Graduação em Engenharia da Computação

Otimização de Técnica de Reconhecimento de Caracteres em Imagens de Cenas Naturais usando Hardware

Rodolfo Ivo Santos de Andrade

Trabalho de Graduação

Recife
21 de dezembro de 2017

Universidade Federal de Pernambuco
Centro de Informática

Rodolfo Ivo Santos de Andrade

Otimização de Técnica de Reconhecimento de Caracteres em Imagens de Cenas Naturais usando Hardware

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
21 de dezembro de 2017

Resumo

Algoritmos de reconhecimento de caracteres em cenas naturais (STCR) de alta precisão requerem alta capacidade de processamento. Para executar esses algoritmos em um sistema embarcado de tempo real de modo satisfatório é preciso uma arquitetura heterogênea composta por processadores de propósito geral e processadores de propósito único. Desse modo, as etapas da técnica de STCR são alocadas aos componentes de hardware (processadores de propósito único) e de software (processadores de propósito geral). Sendo que as etapas que mais exigem poder de processamento serão dedicadas ao componente de hardware do sistema, com o objetivo de diminuir o tempo total de execução do algoritmo. Dentre uma estratégia definida pelo grupo de pesquisa, uma das etapas de razoável custo computacional consiste na etapa de cálculo do HOG (*Histogram of Oriented Objects* (HOG)). Este trabalho tem como objetivo a implementação de um módulo para cálculo do HOG em hardware reconfigurável FPGA.

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

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

OCR Optical Character Recognition
STCR Scene Text Character Recognition
CPU Cental Processing Unit
FPGA Field Programmable Gate Array
HOG Histogram of Oriented Gradients
FPS Frames Per Second
NSCR Natural Scene Character Recognition
SC Shape Context
GB Geometric Blur
SIFT Scale Invariant Feature Transform
MR8 Maximum Response of filters
PCH Patch descriptor
NN Nearest Neighbor
SVM Support Vector Machine
MKL Multiple Kernel Learning
I2CDML Image-to-class Distance Metric Learning
PCANet Principal Component Analysis Network
CNN Convolutional Neural Networks
PCA Principal Component Analysis
BOW Bag-of-Words
GMM Gaussian Mixture Model
HD Harris Detector
MD MSER Detector
DD Dense Detector
RD Random Detector
ELM Extreme Learning Machine
RIFFA Reusable Integration Framework for FPGA Accelerators
FIFO First In First Out
BCU Buffer Control Unit
BRAM Block random-access memory
KFU Kernel Function Unit
LUT Look-up Table
FCU Filter Control Unit
RNA Rede Neural Artificial

BIA Bicubic Interpolation Accelerator

Capítulo 1

Introdução

Com a transição de aplicações analógicas para as digitais houve necessidade de conversão de documentos escritos e impressos. Para isso foram desenvolvidas técnicas de reconhecimento de caracteres a partir de imagens desses documentos. Uma das técnicas se chama *Optical Character Recognition* (OCR) e foi se aperfeiçoado com o tempo. Atualmente existe outro problema similar, porém de maior complexidade, o reconhecimento de caracteres em cenas naturais ou *Scene Text Character Recognition* (STCR) que busca reconhecer caracteres em imagens de um ambiente externo, como placas de trânsitos, cartazes, *outdoors* e em objetos, como os produtos industrializados encontrados nos supermercados. Alguns exemplos de cenas naturais podem ser visto na figura 1.1.

Figura 1.1 Exemplos de cenas naturais do banco de imagens Chars74K.

A tarefa de reconhecimento de caractere OCR está relacionada ao problema de análise e reconhecimento de documentos digitalizados por meio de câmera, a aplicação dessa técnica está limitada a imagens que representam um documento impresso ou que possui texto em sua maioria [2]. Diferente da técnica de OCR, técnicas para STCR não exigem um ambiente isolado com imagem manipulada em perfeitas condições para análise. Imagens em cenas naturais possuem diversas contingências que dificultam a análise e classificação dos símbolos.

STCR foca no problema de reconhecimento de caracteres extraídos de imagem em cenas reais. Essa tarefa lida com problemas derivados das diversas forma de representação de caracteres, fontes, tamanhos, inclinação e rotação, cor do plano de fundo e cor do caracteres e de outros fatores ambientais externos, como variação da iluminação e também da própria ferramenta de captura da imagem que pode causar distorção ou borramento da imagem. Além disso, caracteres não tem uma estrutura fixa e discriminante [3], existem diversas formas de representar um caractere e essa forma pode ser legível para um ser humano, mas uma máquina teria dificuldade de classificar.

Com o advento de carros autônomos e tecnologia embarcadas, há cada vez mais uma demanda por tecnologias capazes de reconhecer o ambiente externo, adaptar-se e atuar de acordo com o seu propósito. As aplicações para algoritmos de STCR são várias, um exemplo seria o reconhecimento de placas de trânsitos por veículos autônomos [4] [5] e também tecnologia assistiva para deficientes visuais [6]. Diversas aplicações dependem do desenvolvimento e

aperfeiçoamento dessas técnicas e com os avanços atuais na área de sistemas embarcados e inteligencia artificial isso se torna possível. Outro problema se deve ao requisito da aplicação que quando embarcada em dispositivos ou em circuito integrado é necessário considerar o tempo de resposta, o custo e o consumo de energia.

A solução proposta é uma implementação mais eficiente da técnica de STCR proposta anteriormente [1] em uma arquitetura heterogênea, composta por uma processador de propósito geral (CPU) e aceleradores de hardware (FPGA). Com objetivo de otimizar o algoritmo STCR, a técnica foi dividida em etapas, dentre as quais as maiores consumidoras de tempo de processamento e de fácil implementação em hardware foram otimizadas em forma de módulos em hardware implementadas em FPGA enquanto que as restantes foram implementados no processador de propósito geral. O módulo Resize que se destina ao redimensionamento da imagem já tinha sido implementado em SystemVerilog por Lima [1].

Na arquitetura proposta por Lima [1], o sistema tem uma velocidade de 11.7 vezes maior em comparação com outros sistemas e apresenta a melhor acurácia entre eles, de 65.5% para o banco de dados de imagens Chars74K-15. A figura 1.1 mostra exemplos desse *dataset*. Mas o tempo de execução do algoritmo não é suficientemente bom para uma aplicação em vídeo online, pois a frequência de processamento está abaixo de 24 fps, frequência perceptível aos olhos humanos.

1.1 Objetivos

Esse novo trabalho tem como objetivo desenvolver uma implementação em hardware para o módulo limiarizador Otsu e para o módulo que calcula o histograma de características HOG. Ambos os módulos foram especificados em SystemVerilog para síntese e prototipação em FPGA. A implementação em hardware possui vários benefícios, como redução da potencia, diminuição de área e custo além de alcançar um melhor desempenho.

Nesse trabalho foi proposta uma otimização de modo a contribuir com a implementação já realizada pelo autor Luiz Lima Júnior. Experimentos realizados no artigo anterior [1] comprovam que o tempo de processamento é reduzido quando o módulo Resize foi implementado em hardware em uma arquitetura híbrida hardware-software (CPU-FPGA).

1.2 Estrutura do Trabalho

O trabalho é organizado em seis capítulos. No capitulo 2 são apresentadas técnicas relacionadas ao sistema STCR, uma breve descrição de quais técnicas existem na literatura, a acurácia obtida por estes sistemas e para alguns, o tempo de processamento para o mesmo banco de imagens Chars74K-15. O terceiro capítulo foi dedicado à técnica usada nesse trabalho. No quarto capítulo será detalhada a implementação dos dois novos módulos em hardware, o Otsu e HOG. No capítulo 5 é descrito os experimentos para o validação e o resultado obtido. Por fim, o capítulo 6 finaliza com a conclusão e sugere possíveis trabalhos futuros.

Capítulo 2

Técnicas para Reconhecimento de Caracteres em Cenas Naturais

Neste capítulo serão abordadas as técnicas de reconhecimento de caracteres em cenas naturais que existem na literatura e um comparativo em termos de acurácia e tempo de processamento.

2.1 Scene Text Character Recognition

Uma das aplicações da visão computacional é a detecção de caracteres. Essa tarefa conhecida como *Optical Character Recognition* (OCR) busca a transcrição do texto presente em uma imagem em um documento digital de formato texto. Além da dificuldade associada ao reconhecimento da letra ou do número, em alguns casos, o plano de fundo não uniforme do carácter pode ser um problema devido ao contraste. Esse é o desafio atual de reconhecimento de caracteres em cenas naturais *Scene Text Character Recognition* (STCR) ou *Natural Scene Character Recognition* (NSCR). Não só o plano de fundo como a própria fonte de texto e o seu tamanho podem dificultar a classificação. A iluminação da imagem e a textura do objeto em que os caracteres estão posicionados e até mesmo a disposição desses caracteres acrescentam outro fator de complicação [7]. Esses diversos fatores somados permitem que caracteres diferentes se tornem similares nas imagens, como é ilustrado na figura 2.1. Por esse motivo que sistemas OCR falham, ocasionando a classificação errônea dos caracteres.

Figura 2.1 Exemplos de imagens de caracteres em cenas naturais e suas classes. A imagem demostra a semelhança entre caracteres de imagens de diferente classificação.

Esse trabalho cita alguma das técnicas de STCR presentes na literatura. Independente da técnica, geralmente sistemas STCR contêm uma sequência de três etapas principais. A primeira etapa de pré-processamento visa reduzir ruídos da imagem de entrada. A segunda etapa é dedicada à extração de características da imagem. Por último, a terceira etapa usa os dados da segunda para auxiliar na classificação da imagem. A figura 2.2 mostra um diagrama do sistema com três módulos representando cada etapa de processamento, tendo como entrada a imagem de um caractere e como saída a classificação do caractere.

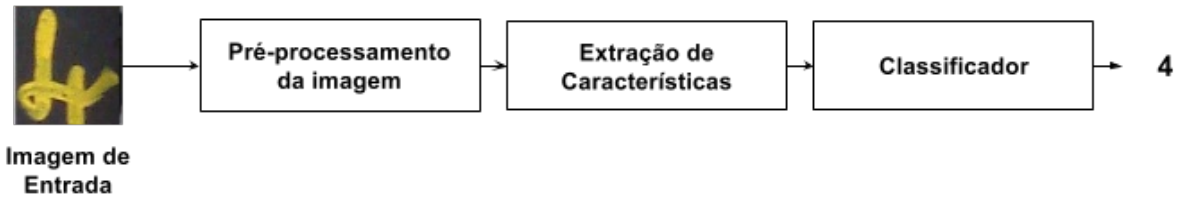

Figura 2.2 *Flowchart* genérico de STCR. Cada módulo representa uma etapa principal do sistema. Esse exemplo tem como entrada a imagem do caractere "4"e sua saída a classificação do caractere.

Técnicas de STCR variam e são compostas por outras tarefas que envolvem detecção de áreas de texto e correção seguidas pela técnica de OCR [2] ou então localização, segmentação e reconhecimento de caracteres ou de palavras usando um vocabulário como auxílio [5].

Recentemente, técnicas para extração de características do caracteres podem ser resumidas em três categorias: *Histogram of Oriented Gradients* (HOG) e suas variações, *mid-level character feature learning methods* e *deep learning* [8].

2.2 Trabalhos Relacionados

Dentro do contexto de STCR, ferramentas comerciais não tem boa acurácia e não servem ao propósito de reconhecimento de caracteres em cenas naturais como afirma de Campos [2]. No artigo de Campos, o autor também introduz um *dataset* que posteriormente se tornou popular, sendo usado nessa categoria de reconhecimento, e que foi adotado como referência nesse trabalho e no trabalho de Lima [1].

Entre as técnicas STCR existentes na literatura, as que conseguiram obter boa precisão para o banco de dados Chars74k-15 [5] [8] [3] falharam no desempenho devido ao seu tempo de reconhecimento. Para aplicações embarcadas com requisitos de tempo real essas técnicas não são adequadas.

Considerando a necessidade de uma boa acurácia e com bom desempenho temos técnicas que usam o hardware de forma a otimizar o processamento como foi citado por [9] e [10]. O problema é que devido ao uso de um *dataset* diferente do Chars74k-15 não há meio rápidos de se fazer uma comparação realista.

A seguir os trabalhos relacionados à esse projeto.

2.2.1 Character Recognition in Natural Images

Este trabalho [2] aborda o problema de reconhecer caracteres em inglês e em kannada em cenas reais. O kannada é um idioma do sul da Índia que usa o alfabeto kannada, muito diferente do alfabeto latino. Um exemplo de alfabeto kannada pode ser visto na imagem 2.3. O artigo usa um banco de dados próprio de imagens extraídas das ruas de Bangalore na Índia usando uma câmera comum. Todos os caracteres foram extraídos dessas imagens manualmente, compondo assim o *dataset* Chars74K. Esse *dataset* foi introduzido por esse trabalho. Totalizando 7705 imagens de números, letras maiúsculas e letras minusculas.

Figura 2.3 Exemplo de caracteres kannada extraídos do banco de dados Chars74k. Cada um dos caracteres representados nas imagens são de classes diferentes. No kannada, a combinação de consoantes e vogais resultam em mais de 600 classes distintas.

No banco de imagem de letras em inglês, cada imagem pertencente à uma das 62 classes de letras ou números, sendo que 10 classes representam cada algorismo numérico e 52 classes para cada letra tanto maiúscula como minúscula. Para que a aplicação seja robusta é recomendado utilizar uma subcategoria do banco de dados, o Chars74K-15, que possui 15 imagens por classe totalizando 930 imagens de treino e 930 de teste.

O trabalho citado conclui que ferramentas comerciais de OCR não são apropriadas para reconhecer caracteres em cenas naturais. Os autores obtiveram uma melhora de 25% com seu classificador em relação ao Sistema OCR de estado da arte, ABBYY FineReader 8.0.

Foi feito um *benchmark* usando o subconjunto Chars74k-15. Usando diversas técnicas de extração de características: *Shape Context* (SC), *Geometric Blur* (GB), *Scale Invariant Feature Transform* (SIFT), *Spin image*, *Maximum Response of filters* (MR8) e *Patch descriptor* (PCH) para a formação de um histograma de características *Bag-of-Visual-Words* e utilizando os seguintes classificadores: *Nearest Neighbor* (NN), *Support Vector Machines* (SVM) e *Multiple Kernel Learning* (MKL). Desses, o classificador MKL com a combinação de todos as técnicas de extração foi capaz de obter uma acurácia de 55.26% em comparação com o ABBYY de 30.77%. Os autores ainda afirmam que esse resultado poderia ser melhor caso desconsiderassem a diferença entre letras maiúsculas e minúsculas.

É importante ressaltar que a técnica foi executada exclusivamente em software e o tempo de execução não foi computado.

2.2.2 Character Recognition In Natural Scene Images Using Rank1 Tensor Decomposition

Esse artigo [5] utiliza tensor como extrator de características do caractere. Um tensor é uma representação de um objeto usando *arrays*. No caso do tensor do artigo, foi usado um tensor modo 3, que se resume à três *arrays* sendo dois deles representando o espaço x e y e o terceiro a rotação do caractere. Para a classificação foi usado o *Image-To-Class Distance Metric Learning* (I2CDML). Nesse artigo, para o dataset de Chars74K-15, foi obtido a acurácia de 59%.

Novamente, o algoritmo foi implementado em software e não foi informado o tempo de execução.

2.2.3 Scene Character Recognition Using PCANet

Principal Component Analysis Network (PCANet) é uma rede neural *deep learning* que usa um mecanismo de convolução similar às redes neurais convolucionais (CNN) só que baseado em PCA [8].

A acurácia resultante desse método foi de 64% para o Chars74K-15. Nesse artigo, o tempo de teste para cada imagem foi de aproximadamente 1 segundo.

2.2.4 Scene text recognition in mobile applications by character descriptor and structure configuration

O artigo [3] propõe um reconhecimento textual em cenas reais a partir da área da imagem que contém texto. Essa área textual é obtida usando um algoritmo anterior dos mesmos autores. O método atual, propõe dois métodos de reconhecimento de texto. O primeiro método, emprega o modelo *Bag-of-Words* (BOW) enquanto que o segundo o esquema *Gaussian Mixture Model* (GMM). Esses dois classificadores dependem de quatro detectores de pontos chaves da imagem ou extratores de características, *Harris detector* (HD), *MSER detector* (MD), *Dense detector* (DD) e *Random detector* (RD). No primeiro método, para cada um dos detectores, um vetor de características HOG é calculado. E para cada um dos histogramas HOGs foi criado um vocabulário que por fim foram organizados em um modelo BOW. Já no segundo método, o

GMM é um produto de histogramas derivados dos detectores DD e RD somente. Embora o artigo cite esses dois métodos, eles combinados resultam em uma melhor acurácia para o banco de dados Chars74K-15, atingindo 60% da taxa de acerto.

Outro ponto relevante é que esse trabalho é proposto para aplicações mobile. O modelo processa online as imagens da câmera a cada *frame* extraindo as informações textuais em cerca de um segundo.

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

Técnicas de STCR além de desafiadoras são computacionalmente custosas. Isso é verificado nos trabalhos relacionados. O método de Luiz Lima [1] propôs um sistema heterogêneo em hardware e software de acurácia de 65.5% de baixo tempo de processamento (0.085s) comparado com outras técnicas executadas em software para o banco de dados Chars74K-15 [2].

O algoritmo proposto por Luiz Lima pode ser resumido à três etapas, pré-processamento, extração de características e classificação.

A primeira etapa de pré-processamento busca filtrar a imagem e eliminar ruídos, melhorando a qualidade da imagem. Essa etapa visa isolar ao máximo os componentes que agregam maior informação em relação a sua verdadeira classe. No trabalho de graduação de Luiz Lima Júnior [1] foram usadas três técnicas de pré-processamento, a de conversão de espaço de cores para escala de cinza, redimensionamento da imagem e a limiarização de Otsu.

O algoritmo converte a imagem em escala de cinza na primeira etapa, redimensiona para a dimensão de 128x128 pixel que é limiarizada separando o caractere do plano de fundo.

A conversão de cores é essencial ao algoritmo. O espaço de cores de padrão RGB tem uma representação muito grande. Para cada pixel da imagem colorida ela é representada por 3 conjunto de 8 bits. Para cada pixel é preciso o valor em 8 bits de cada uma das cores, vermelho, verde e azul. Quando convertido para escala de cinza esse pixel tem tamanho de um único byte (8 bits) com capacidade de representar 256 tons de cinza, facilitando assim o uso de memória e reduzindo o custo computacional.

Em seguida, a imagem já convertida para escala de cinza é redimensionada para as dimensões 128x128. Alguns algoritmos de extração de características exigem imagens de tamanho especifico. Por isso é necessário um algoritmo de redimensionamento de imagem na etapa de pré-processamento. Essa dimensão de 128x128 é exigida pela técnica de extração de característica HOG que será explicada mais adiante.

Depois de redimensionada, o caractere é detectada pelo limiarizador de Otsu e o seu plano de fundo é excluído, resultando em uma imagem binária em que o pixel 1 representa o caractere e o 0 o plano de fundo.

Na etapa seguinte, chamada de extração de características, busca extrair informações presentes na imagem que auxiliem na classificação correta do caractere na etapa de classificação. Essas informações são armazenadas em forma de vetor e será usado na etapa seguinte de classificação. A técnica utilizada no trabalho citado foi o *histogram of oriented gradients* (HOG) para essa etapa.

A terceira e última etapa é a de reconhecimento do caractere. No sistema heterogêneo o classificador escolhido de menor custo computacional foi o *extreme learning machine* (ELM).

2.3 Comparação dos algoritmos STCR

A seguir é mostrado a tabela 2.1 comparando os métodos citados, inclusive o algoritmo de Lima. Nessa comparação foi levado em consideração o tempo de execução, quando informado no seu respectivo artigo, em segundos e a taxa de acerto em porcentagem.

Tabela 2.1 Performance dos métodos de STCR no dataset Chars74K-15.

Método	Tempo de Execução (s)	Arquitetura	Acurácia (%)
MKL [2]	-	-	55.76
Rank1 Tensor + I2CDML [5]	-	-	59
PCANet [8]	1s	CPU	64
Keypoints + HOG + Adaboost [3]	1s	CPU ARM Cortex-A9 1.2 GHz dual-core + GPU GeForce ULP	60
Resize + Otsu + HOG + ELM [1]	0.308	Intel Atom N2600	65.5
Resize (FPGA) + Otsu + HOG + ELM [1]	0.085	Intel Atom N2600 + FPGA	65.5

Capítulo 3

Algoritmo STCR

Nesse capítulo será detalhado o algoritmo STCR proposto para reconhecimento de caracteres em cenas naturais.

3.1 Visão Geral

Esse trabalho visa uma melhoria no trabalho de graduação de Luiz Lima em que foi desenvolvido um algoritmo de STCR com um módulo de hardware dedicado ao redimensionamento da imagem. Nesse trabalho será realizada a implementação de dois módulos em hardware, o módulo Otsu e o módulo HOG, que são executados posteriormente ao módulo Resize e antes da última etapa de classificação usando ELM. A figura 3.1 mostra o diagrama de blocos do algoritmo, suas etapas e o nome dos módulos.

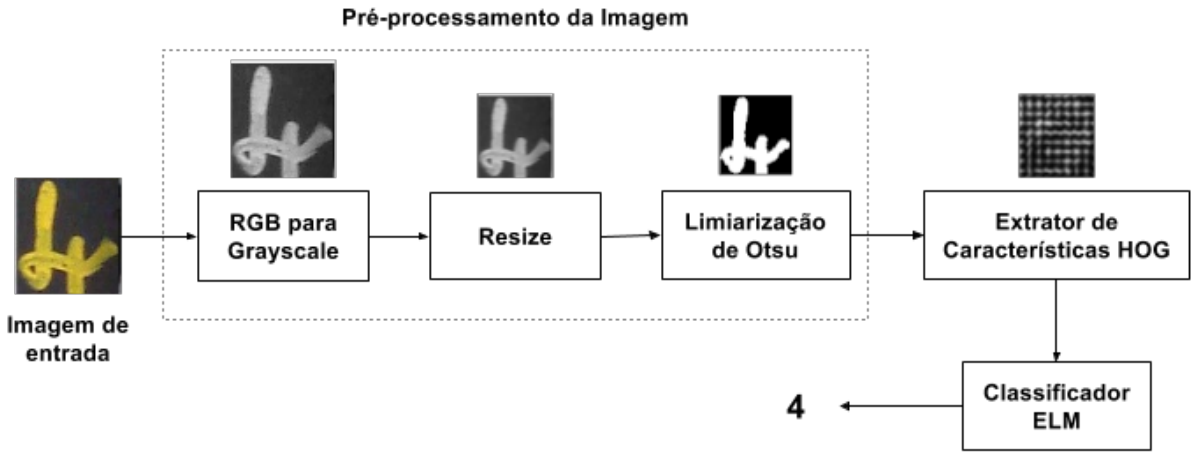

Figura 3.1 Flowchart do sistema STCR de Lima.

O objetivo é otimizar ainda mais o desempenho, introduzindo outros dois módulos de hardware após a etapa de redimensionamento para reduzir o tempo de execução total da técnica STCR.

No trabalho anterior foi feita a análise do tempo médio para todas as 930 imagens de teste do banco de dados Chars74K-15 de cada etapa do algoritmo em C++ no processador Intel Atom N2600. A medição do tempo se fez por meio da ferramente Gprof, que é uma ferramenta de análise de performance para ambiente Linux com o auxílio da biblioteca time.h. Sendo identificado a tarefa mais custosa computacionalmente, o Resize, foi determinado que essa tarefa consume 71.75% do tempo de execução do algoritmo, totalizando 220.34 ms. Por esse motivo foi imprescindível o desenvolvimento do módulo em hardware chamado de *Bicubid Interpolation Accelerator* (BIA). Esse era o único módulo em hardware do algoritmo proposto até então, formando assim uma arquitetura híbrida e heterogênea CPU-FPGA. Este componente BIA foi implementado em hardware usando a linguagem de descrição de hardware SystemVerilog HDL, enquanto que o restante do algoritmo em linguagem C++.

Nesse trabalho, foi desenvolvido dois módulos que passaram do software para o hardware, esses módulos são o Otsu e o HOG. O HOG é o terceiro módulo que mais consome tempo, cerca de 30.9 ms são dedicados à essa atividade. O motivo principal para a implementação do Otsu é para evitar o uso de barramento. O Otsu não consome muito tempo, somente 1.2 ms, mas ele é um módulo intermediário entre o Resize e o HOG. Implementando o Otsu em hardware, garante a sequência de processamento da imagem sem depender do barramento.

3.2 Algoritmo STCR

Inicialmente o algoritmo STCR tem como entrada uma imagem no espaço de cores RGB. Na primeira etapa de pré-processamento a imagem é convertida para a escala de cinza. Essa etapa é realizada em software na CPU.

A etapa seguinte à conversão necessita da transmissão de bytes da imagem pelo barramento PCI Express. Para isso foi criado dois módulos, um de envio e outro de recebimento dedicados ao envio da imagem em escala de cinza para o hardware e o de recebimento da imagem vinda do software.

Depois da imagem ser recebida pelo hardware. O módulo Resize que tem como entrada uma imagem de tamanho dinâmico, o *dataset* não tem um padrão de largura e altura para as imagens, ou seja, os tamanhos não são previsíveis e isso deve ser tratado já que o outro módulo da etapa de extração de características, HOG, necessita de uma imagem de tamanho padronizado limitado pela dimensão 128x128 pixel.

A arquitetura foi alterada nesta etapa, quando a imagem é redimensionada no hardware. Anteriormente a imagem voltava ao software, e assim executava todas as outras etapas de limiarização, HOG e ELM. Nessa nova arquitetura, o módulo Limiarizador Otsu e o módulo extrator HOG são executados no hardware na sequência depois do BIA, que redimensiona a imagem. Só então o vetor de características, saída do módulo HOG, é enviado de volta ao software com os módulos de envio do hardware e de recebimento no software.

De volta ao software o vetor de características é usado pelo classificador ELM para classificar a imagem.

3.3 GrayScale

Cada pixel da imagem em RGB tem três componentes, representando a intensidade das três cores primárias, Red, Green e Blue. Na conversão cada um dos seus componentes são multiplicados por determinados valores e somados. O resultado é a intensidade daquele pixel na escala de cinza. A seguir a função de conversão na equação 3.1.

$$Gray = 0.21 * R + 0.72 * G + 0.07 * B \tag{3.1}$$

Gray é a intensidade em escala de cinza. R, G e B são a intensidade das cores em vermelho (R), verde (G) e azul (B).

3.4 Comunicação Hardware-Software

A comunicação é feita pelo barramento PCI Express através da Interface *framework* RIFFA. No programa em C++, executado na CPU, o RIFFA usa duas principais funções, a função fpga_send() para enviar a imagem em *stream* de bytes e fpga_receive() para receber no software. Enquanto que no hardware FIFOs são usadas para a comunicação com a CPU. A interface com o barramento PCI Express é abstraída pela máquina de status do RIFFA.

O protocolo usado pelo RIFFA pede que dados estejam em formato de 32 bits. Por causa disso foi criado um módulo em software que empacota pixels em palavras de 32 bits. Cada pixel, representado por um inteiro de 32 bits é reduzido a 8 bits (1 byte) sem perda de informação, já que o valor do pixel vai de 0 até 255. A tarefa percorre a imagem, selecionando 4 pixels adjacentes e concatena esses valores. A concatenação de 4 pixels de 8 bits cada gera uma palavra de 32 bits. Esse protocolo transmite uma imagem convertida em um *streaming* de bytes para o hardware.

A quantidade de bits do barramento depende do *framework* RIFFA e da plataforma de destino. Sendo que para a placa DE2i-150, 64 bits foi a quantidade de bits usada no barramento.

3.5 Resize

O módulo Resize usa o IP-core *Bicubic Interpolation Accelerator* (BIA). Esse módulo é baseado no algoritmo de interpolação bicúbica. Esse algoritmo é eficiente em termos de processamento e de qualidade da imagem. A seguir uma representação desse algoritmo na figura 3.2.

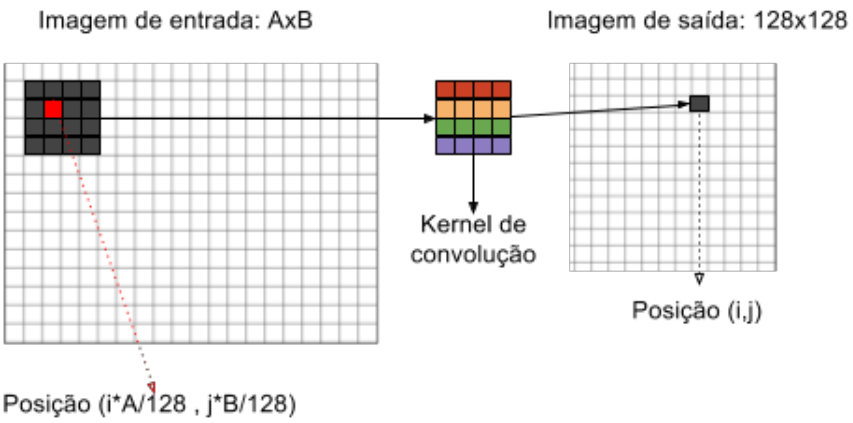

Figura 3.2 Visualização do algoritmo de interpolação bicúbica. Cada quadrado representa um pixel. O pixel vermelho é o selecionado, a região cinza são os pixels vizinhos.

Uma forma de entender o algoritmo é imaginar a imagem de entrada dividida por *grids* em linhas e colunas de acordo com o tamanho desejado, nesse caso 128x128, 128 linhas e 128 colunas espaçadas uniformemente pela imagem. Os *grids* na imagem auxiliam na visualização do processo. No local onde há uma intercepção dos *grids* é extraído o pixel desejado para formar a nova imagem. Desse pixel, é calculado um valor a partir de seus vizinhos e o resultado é o pixel da imagem de saída. O cálculo é feito com a equação 3.2.

$$I_{out}(x',y') = \sum_{m=-1}^{+2} \sum_{n=-1}^{+2} I(x+m,y+n) r_{cub}(dx-m) r_{cub}(dy-n) \tag{3.2}$$

Sendo $I_{out}(x',y')$ o pixel da imagem de saída. x e y são as coordenadas do respectivo pixel da imagem de entrada. m e n estão limitados a [-1,2], ou seja, $I(x+m,y+n)$ representa o pixel dos quatro vizinhos calculados para obter o pixel de saída. O *kernel* de convolução $r_{cub}(x)|a$ gera o peso para aquele pixel. Foi definido que o valor de a é 0.5. A seguir a equação 3.3 que representa o *kernel*.

$$r_{cub}(x,a) = \begin{cases} (-a+2)|x|^3 + (a-3)|x|^2 - 1, & \text{se } 0 \leq |x| < 1, \\ -a|x|^3 + 5a|x|^2 - 8a|x| + 4a, & \text{se } 1 \leq |x| < 2, \\ 0, & \text{se } |x| \geq 2 \end{cases} \tag{3.3}$$

A seguir o pseudo-código 1 [1] que exemplifica o algoritmo.

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

Entrada: I, a imagem original;
I_x e I_y, a quantidade de linhas e colunas da imagem original, respectivamente;
O_x e O_y, a quantidade de linhas e colunas da imagem de saída
Saída: O, a imagem redimensionada;
for $i \leftarrow 0, O_x$ **do**
  for $j \leftarrow 0, O_y$ **do**
    $x \leftarrow \lfloor i * \frac{I_x}{O_x} \rfloor$;
    $y \leftarrow \lfloor j * \frac{I_y}{O_y} \rfloor$;
    $O(i,j) \leftarrow 0$;
    for $m \leftarrow -1 : 2$ **do**
      for $n \leftarrow -1 : 2$ **do**
        $dx \leftarrow i * \frac{I_x}{O_x} - y$;
        $dy \leftarrow j * \frac{I_y}{O_y} - x$;
        if $x+m < I_x$ ***and*** $y+n < I_y$ **then**
          $O(i,j) \leftarrow$
          $O(i,j) + I(x+m, y+n) * r_{cub}(dx-m)|_{a=0.5} * r_{cub}(dy-n)|_{a=0.5}$
        end
      end
    end
  end
end

Na implementação em hardware, o *Buffer Control Unit* é responsável por armazenar a imagem em BRAMs. Essas memórias tem latência de um ciclo. São usados quatro memórias, seu armazenamento é feito de acordo com mod 4 da sequência do pixel da imagem de entrada. Existe outro módulo *Kernel Function Unit* dedicado ao cálculo dos pesos utilizando *look-up table* (LUT). O *Filter Control Unit* é o módulo central, usa uma máquina de estados para calcular as coordenadas dos pixels, se comunica com as memórias BRAMs, se comunica com o *Kernel Function Unit* para obter os pesos e faz as operações aritméticas para obter o pixel de sáida.

3.6 Limiarização de Otsu

Tendo obtido a imagem já redimensionada por 128x128 pixel vindo do módulo Resize. Essa imagem é segmentada separando o caractere do plano de fundo. A saída da limiarização de Otsu é uma imagem binária, sendo que o valor de 1 destina-se a cor branca enquanto que 0 à cor preta. Essa etapa atua como uma limiarização, quando o pixel da imagem de entrada for maior ou igual que o limiar calculado esse pixel passa a ser branco, caso contrário preto, como mostra a equação 3.4. Na qual o pixel de saída é representada por I_{bin}, pertencendo a linha e coluna (i, j) e τ é o limiar.

$$I_{bin}(i,j) = \begin{cases} 1, & \text{se } I_{gray}(i,j) \geq \tau \\ 0, & \text{se } I_{gray}(i,j) < \tau \end{cases} \tag{3.4}$$

Existe a possibilidade de usar um limiar fixo porém há uma diminuição da acurácia de segmentação. O limiar dinâmico, que foi usado no algoritmo, é calculado a partir do histograma de cada imagem tendo como objetivo maximizar a variância entre os pixeis mais escuros dos mais claros e que minimize o espalhamento desses pixeis no seu respectivo conjunto. O valor de τ é o valor do pixel que satisfaz essas condições.

Inicialmente é preciso para o cálculo desse valor o histograma de contagem de intensidade de pixel da imagem 128x128 em escala de cinza. Para obter esse valor a equação 3.5 é calculada para cada valor do histograma começando do 0 até o 255, sendo t o pixel da interação.

$$\sigma_w^2(t) = q_b(t)\sigma_b^2(t) + q_f(t)\sigma_f^2(t) \tag{3.5}$$

O valor de $\sigma_w^2(t)$ é o que se deseja maximizar, ele representa o espalhamento entro o plano de fundo e o caractere. Ele é o somatório da variância dentro dessas classes levando em consideração seus respectivos pesos. Seus pesos são calculados pela sua respectivas probabilidade, como é mostrado na equação 3.6 e 3.7:

$$q_b(t) = \sum_{i=0}^{t} P(i) \tag{3.6}$$

$$q_f(t) = \sum_{i=t+1}^{255} P(i) \tag{3.7}$$

Enquanto que os valores da variância do conjunto do plano de fundo e do caractere são computados a seguir respectivamente:

$$\sigma_b^2(t) = \sum_{i=0}^{t} [i - \mu_b(t)]^2 \frac{P(i)}{q_b(t)} \tag{3.8}$$

$$\sigma_f^2(t) = \sum_{i=t+1}^{255} [i - \mu_f(t)]^2 \frac{P(i)}{q_f(t)} \tag{3.9}$$

No qual as variáveis $\mu_b(t)$ e $\mu_f(t)$ representam seus valores médio respectivamente e que são calculados pela fórmula a seguir:

$$\mu_b(t) = \sum_{i=0}^{t} \frac{iP(i)}{q_b(t)} \tag{3.10}$$

$$\mu_f(t) = \sum_{i=t+1}^{255} \frac{iP(i)}{q_f(t)} \tag{3.11}$$

O pseudo-código 2 a seguir [1] explica melhor a sequência de passos para obter o limiar:

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

Entrada: imagem em *grayscale* I_{gray}
Saída: limiar de Otsu τ
$\tau = 0$;
$max\_variance = 0$;
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 $\sigma_w^2(t) < max\_variance$ **then**
        $max\_variance = \sigma_w^2(t)$;
        $\tau = t$;
    end
end

Depois de determinado o limiar, a imagem binária é gerada comparando todos os pixels da imagem com o limiar, se a intensidade do pixel for maior ou igual ao limiar então esse pixel passa a ser 1, caso contrário ele passa a ser 0. A imagem binária obtida, em seguida, é enviada para o próximo módulo, o módulo HOG.

3.7 Extrator de Características HOG

O histograma da imagem é obtido a partir da intensidade dos gradientes dos pixels e da orientação desses gradientes. Esse histograma representa as características da imagem.

Inicialmente a imagem é divida por células de tamanho 18x18 pixels. Em cada pixel de cada célula são calculados os gradientes g_x e g_y, que consistem na diferença de intensidade entre o pixel anterior e o posterior na direção x e y do eixo horizontal e vertical, respectivamente. As equações 3.12 e 3.13 demostram esse cálculo.

$$g_x(x,y) = I(x+1,y) - I(x-1,y) \tag{3.12}$$

$$g_y(x,y) = I(x,y+1) - I(x,y+1) \tag{3.13}$$

$I(x,y)$ representa a intensidade do pixel na posição (x,y) na imagem em escala de cinza.

Dos valores dos gradientes é possível obter a magnitude $m(x,y)$ e a direção $\theta(x,y)$ de cada pixel. Como mostram as equações 3.14 e 3.15.

$$m(x,y) = \sqrt[2]{g_x^2(x,y) + g_y^2(x,y)} \tag{3.14}$$

$$\theta(x,y) = arctan\frac{g_y}{g_x} \tag{3.15}$$

Cada célula tem um histograma de 9 intervalos divididos entre 0º e 180º. Esses intervalos tem comprimento de 20 graus e estão centrados nos valores de 10, 30, 50, 70, 90, 110, 130, 150 e 170 graus. Um exemplo de histograma de uma célula pode ser visto na figura 3.3. Cada pixel acrescenta a magnitude no intervalo de sua orientação e também no intervalos vizinhos para reduzir o efeito do ruído. Os pesos nos vizinhos estão diretamente relacionados às distâncias da orientação em relação ao valor central do intervalo do histograma. Por exemplo, se a orientação do pixel for 50°, ele contribuiria 100% ao intervalo 50° do histograma, mas se fosse 80°, então teria uma contribuição de 50% nos intervalos de 70° e 90°.

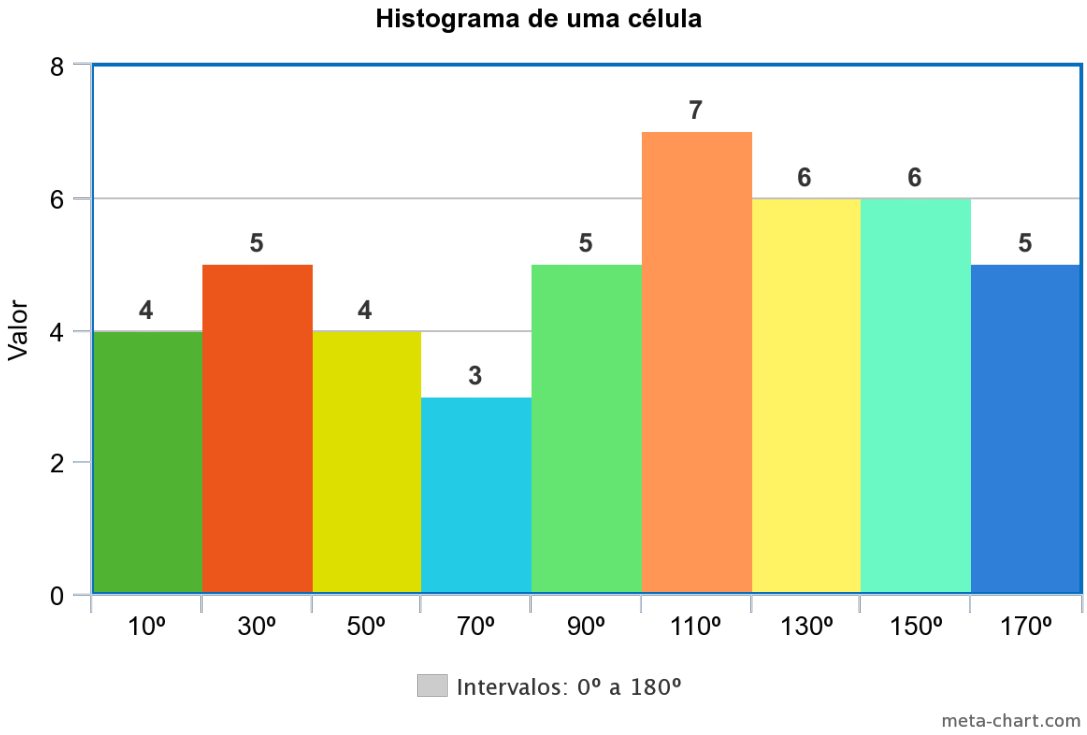

Figura 3.3 Histograma HOG de uma célula. Cada um dos 9 intervalos tem comprimento de 20 graus. O ângulo central de cada intervalo pode ser visto no eixo x. O eixo y consiste no somatório das magnitude de todos os pixels de uma célula.

Em seguida, os histogramas de cada bloco (2x2 células) são concatenados, formando um vetor de 36 componentes (9 intervalos de um histograma por 2x2 células).

Depois disso, o bloco 2x2 move uma célula seguindo a sequência e o processo se repete, a figura 3.4 mostra esse processo. Todos os vetores de cada bloco são concatenados resultando em um histograma de 1296 componentes (uma imagem de 128x128 é percorrida por um bloco 6 vezes por coluna e 6 vezes por linha, como cada bloco tem um histograma de 36 componentes então o histograma final tem 6x6x36 componentes).

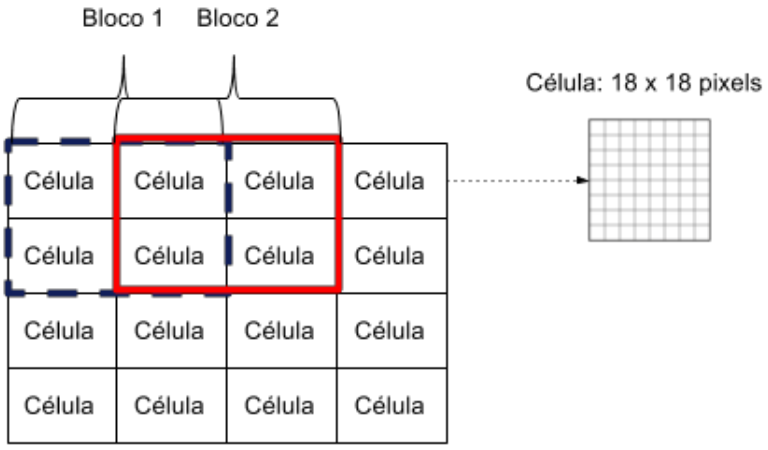

Figura 3.4 A divisão da imagem por células, cada célula tem dimensões 18x18. Também mostra o tamanho dos blocos de 2x2 células. O algoritmo percorre a imagem usando blocos, o bloco 1 de linha azul pontilhada pertence à primeira interação e o bloco 2 de linha vermelha pertence a segunda interação.

Esse vetor de cada bloco (36 componentes) é normalizado usando a equação 3.16 e assim o vetor total (1296 elementos) está pronto para ser usado em um classificador.

$$\text{L2-Norm: } v_{i-norm} = \frac{v_i}{\sqrt[2]{||v_i||^2 + \varepsilon^2}} \tag{3.16}$$

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

A seguir o pseudo-código 9 da técnica HOG [1]:

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

Entrada: I, uma imagem em *grayscale* de 128×128 pixels
Saída: H, vetor com as HOG *features* de I
Divida I em células de tamanho $cell\_size \times cell\_size$
for $block \leftarrow$ *each block* **do**
    for $cell \leftarrow$ *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 da célula, onde θ determina em quais intervalos do histograma aquele gradiente deve contribuir e M indica o quanto o gradiente contribui para aquele intervalo;
    end
end
Normalize todos os histogramas por bloco;
O vetor de saída, H, é a concatenação de todos os histogramas normalizados no passo anterior;

3.8 Classificador Extreme Learning Machine

Extreme Learning Machine ELM é uma rede neural com uma camada intermediária [11]. Esse classificador é uma variação da rede neural artificial RNA. Nas redes RNA a corretude da classificação depende da minimização do error de forma interativa, isso é custoso em termos de processamento diferente da ELM que usa inversão de matrizes para reduzir o error.

A primeira etapa é a escolha aleatória de pesos e biases para a camada intermediaria. O novo conjunto de treinamento os pesos da camada intermediária é obtido pelo produto da matriz invertida de saída da camada escondida com a matriz das saídas esperadas. Assim a rede ELM aprende mais rapidamente em comparação com outras redes neurais *feedforwards*.

Com uma rede ELM de uma camada escondida, a saída é o produto de todas a saída dessa camada escondida multiplicada com o peso respectivo da aresta que liga ao nó de saída. A figura 4.1 mostra a representação de uma rede ELM. Além do produto já citado é preciso considerar o bias, um limiar de cada nó de saída. O produto somado com esse bias passa por uma função de ativação. Foi feito testes empíricos com esse algoritmo variando a função de ativação do ELM e o número de neurônios. O melhor resultado foi obtido com a função de ativação da tangente hiperbólica em comparação com a ativação linear e com 18000 neurônios da camada escondida resultando na maior acurácia de 67.2% [1]. A equação seguinte representa a tangente hiperbólica:

$$g(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.17}$$

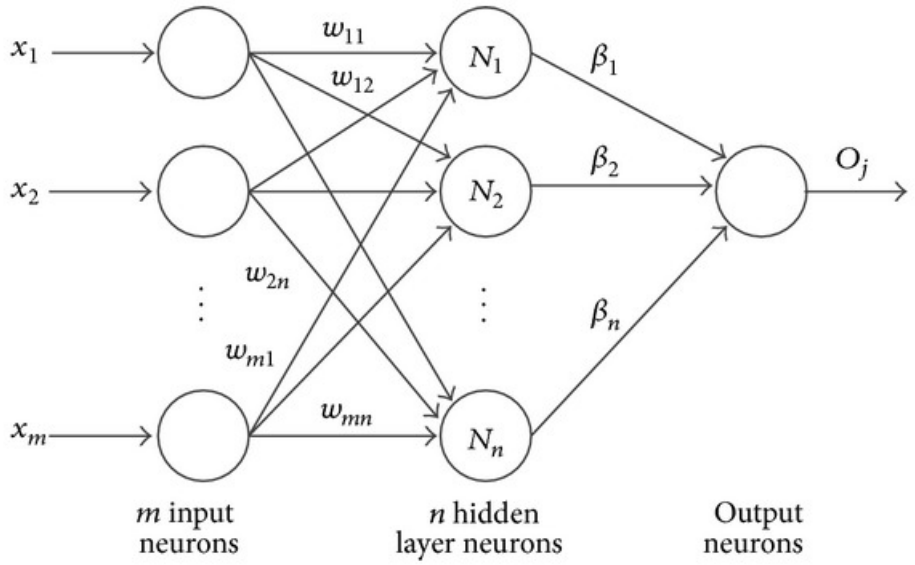

Figura 3.5 Visualização de uma rede neural do tipo Extreme Learning Machine. Onde as entradas são representadas por x, a saída por O, os pesos da entrada por ω e os pesos da camada intermediária por β. Os nós m são os neurônios de entrada e a camada escondida é nomeada de n.

Capítulo 4

Arquitetura de Hardware-Software proposta para STCR

Nessa capítulo é apresentada a arquitetura heterogênea de hardware-software, uma otimização da arquitetura anterior [1]. Na arquitetura presente foram implementados dois novos módulos em hardware com objetivo de reduzir o tempo de processamento do algoritmo.

4.1 Visão Geral

O objetivo atual é melhorar a performance obtida pelo trabalho anterior [1]. É preciso garantir um tempo de processamento de um *frame* acima ou igual à frequência de 24 fps, para que a aplicação funcione com aplicações de vídeos online em dispositivos embarcados. O projeto anterior introduziu a arquitetura híbrida de hardware-software (CPU-FPGA) como meio de acelerar o processamento de reconhecimento de caracteres alfanuméricos em cenas naturais. O projeto cumpriu com objetivo estabelecido de prototipar na placa Terasic DE2i-150. A aplicação em software foi dedicada ao processador Intel ATOM N2600 e a parte de hardware foi designada ao hardware reprogramável FPGA Cyclone IV da Altera. O barramento PCI Express foi abstraído usando o *framework* RIFFA que é mais eficiente na transmissão de dados.

O algoritmo STCR primeiro foi desenvolvido em C++, o tempo médio de cada etapa foi medido usando a ferramenta de Gprof [12], no ambiente Linux, e a biblioteca time.h. O exemplo

do algoritmo modificado para medição do tempo pode ser verificado com o algoritmo 4.

Algoritmo 4: 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;
```

Depois de identificado a tarefa Resize como a que mais exige tempo de processamento, essa foi analisada considerando sua dificuldade de implementação em hardware e por fim projetada com a linguagem SystemVerilog HDL. Foi implementado no trabalho anterior o módulo *Bicubic Interpolation Accelerator* (BIA) dedicado a tarefa do Resize. Antes o módulo Resize em software consumia 220.34 ms, cerca de 71.75% do tempo, como mostra a figura 4.1, no trabalho passado utilizando uma arquitetura híbrida passou para 1.6 ms.

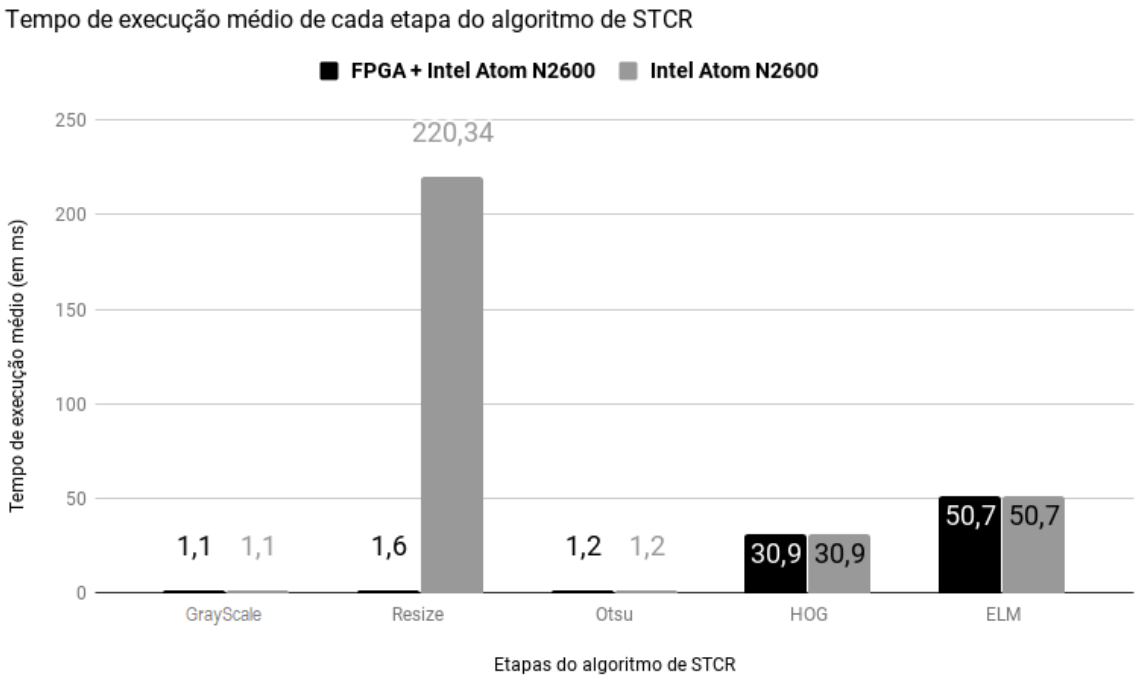

Figura 4.1 Tempo de execução do algoritmo STCR por etapa, em milissegundos, antes e depois da arquitetura híbrida proposta.

A arquitetura proposta por Lima é apresentada na figura 4.2. Na imagem, dois grandes blocos estão representando o processador ATOM (CPU) e o hardware FPGA. O barramento PCI Express está representado entre eles. Os módulos no bloco da CPU foram implementados em software na linguagem C++. Os módulos implementados no FPGA foram descritos usando a linguagem HDL SystemVerilog.

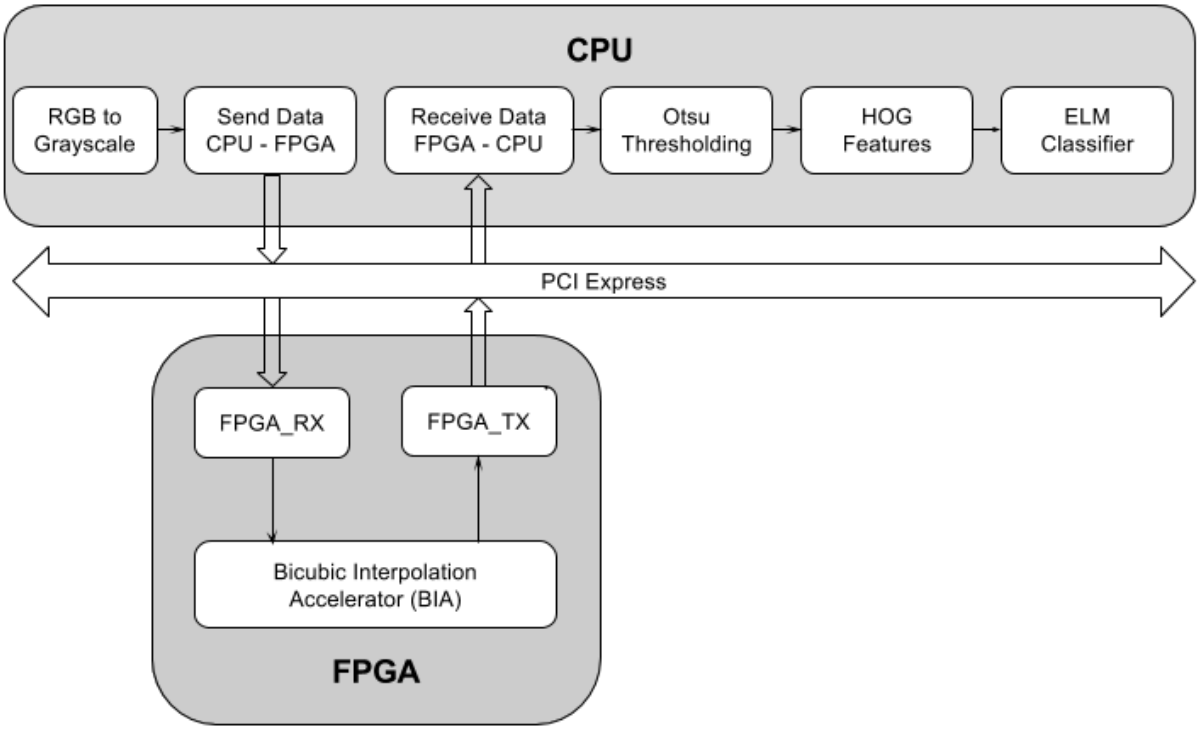

Figura 4.2 Arquitetura heterogênia hardware-software proposta anteriormente [1].

A mesma lógica foi aplicada, dessa vez, com o módulo Otsu e o HOG. De todos os módulos, o módulo HOG é o segundo que mais consome tempo depois do módulo ELM na arquitetura proposta como mostra a figura 4.1. Por esse motivo, esse módulo foi escolhido para ser otimizado em hardware. O Otsu tem um tempo de execução relativamente pequeno, mas por ser uma etapa intermediaria entre o Resize e o HOG foi preferível que também fosse otimizado para que o barramento fosse poupado e consequentemente evitasse a adição de tempo de comunicação extra. Será mantido a execução em software do módulo GrayScale, que converte para escala de cinza, e o módulo de classificação ELM.

O módulo ELM, apesar de ser o segundo módulo que mais consome tempo, sua implementação em hardware comprometeria a redução do tempo com o uso adicional do barramento. Primeiro a imagem iria para o hardware seria redimensionada, passaria para o software para ser processada pelo Otsu e HOG e depois voltaria ao hardware para ser classificada pelo ELM e o resultado voltaria ao software.

A nova arquitetura proposta é apresentada na figura 4.3. O sistema apresentado tem uma sequência linear de processamento, todos os blocos são executados uma vez, um depois do outro. Inicialmente, o sistema executa em software e tem como entrada uma imagem de dimensões não definidas, essa imagem é convertida à coloração preto e branco pelo módulo "RGB to Grayscale". Após isso, a imagem convertida é direcionada ao hardware pelo bloco "Send Data CPU - FPGA", que envia um *stream* de bytes pelo barramento até o módulo "FPGA_RX". Depois de recebida a imagem, o módulo BIA tem a tarefa de converter a imagem de tamanho qualquer á dimensões fixas de 128x128 pixels. Tendo redimensionado a imagem, então o módulo "Otsu Thresholding"converte a imagem em cores binárias, somente preta ou branca. Em seguida

o módulo "HOG Features"extrai as características da imagem binária. Na sequência, o hardware envia de volta à CPU a imagem em *stream* de bytes com o auxilio da "FPGA_TX"que é recebido pelo "Receive Data FPGA - CPU". E por fim, no software, o módulo "ELM Classifier"classifica a imagem em uma das 62 classes que foram treinadas previamente.

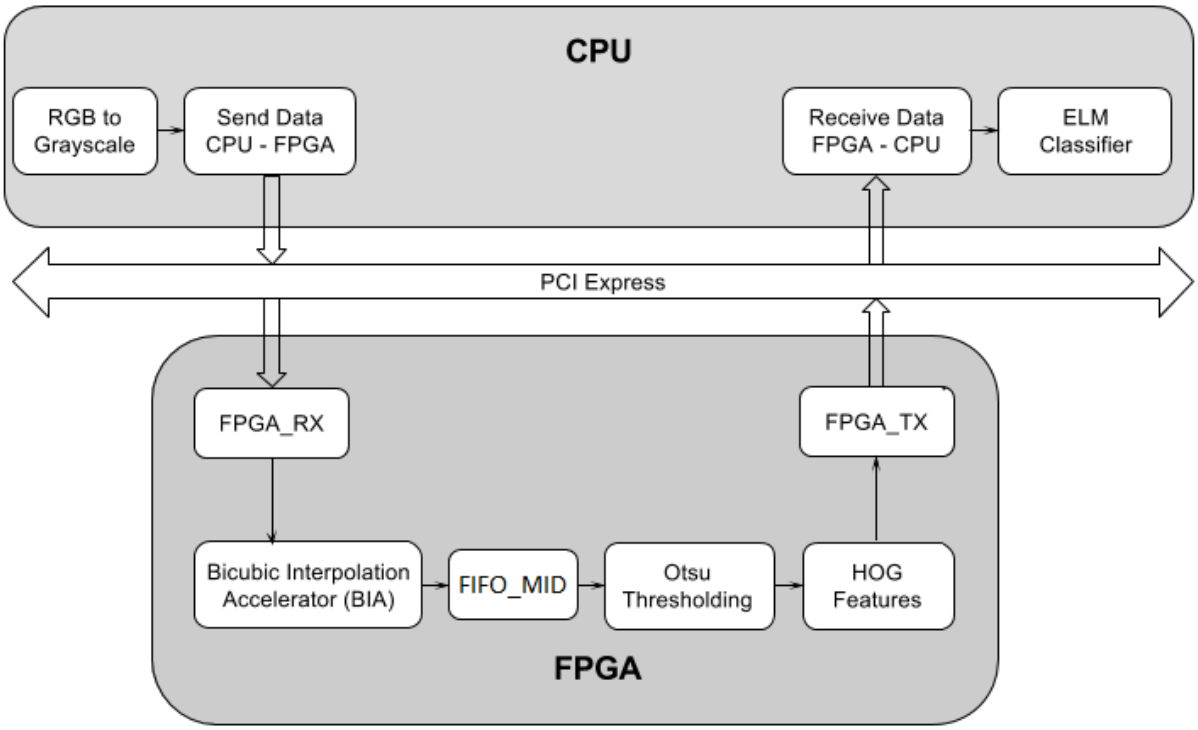

Figura 4.3 Nova arquitetura heterogênica hardware-software proposta.

4.2 Barramento PCI Express

Outro detalhe importante é a comunicação entre os módulos. O barramento PCI Express é abstraído com o *framework* RIFFA dentro do software. Enquanto que no hardware é usado módulos FIFOs. O software envia *stream* de bytes ao hardware do mesmo modo que o trabalho passado [1]. Para a comunicação de volta do sistema, foi preciso criar uma nova FIFO de saída utilizando a ferramenta de criação do Quartus. A FIFO de saída anterior recebia um byte e enviava 4 bytes. A nova FIFO foi projetada para enviar e receber 64 bits. Além disso, foi feita uma FIFO intermediária, para ajustar a entrada do módulo Otsu com a saída do módulo Resize. Outro componente é o RIFFA que é capaz de simplificar a interface com o barramento PCI Express usando sinais de controle protocolares.

As FIFOs assíncronas tem como função evitar perda de dados devido a diferença de fluxo entre a CPU e a FPGA. A transferência do módulo em software ao de hardware e vice-versa não é fluida. As FIFOs foram projetadas para garantir essa transição. Outro problema solucionado pelas FIFOs assíncronas é a diferença de frequências entre os módulos que necessitou de logica assíncrona no sistema, o que não é desejada em um algoritmo sequencial. Isso leva a problemas

como *metastability* e dados inteligíveis obtidos no barramento. Os módulos FPGA_RX e FPGA_TX tem frequência de 125 MHz e os IP-Cores de 50 MHz.

Nas seções seguintes serão explicados com maiores detalhes os módulos de comunicação em software e hardware desse trabalho, tanto para recebimento de *stream* de bytes como para envio. É importante considerar que não houve modificações nos módulos "Send Data CPU - FPGA"e "FPGA_RX", esses módulos enviam bytes do software ao hardware. A arquitetura atual complementa a antiga arquitetura somente depois do módulo BIA, ou seja, o sistema continua recebendo bytes da imagem no hardware do mesmo modo que o sistema antigo. A única mudança quanto a comunicação será a implementação dos novos módulos "FPGA_TX"e "Receive Data FPGA - CPU", que tiveram de ser modificados pois a saída do hardware passou a ser de outro tipo então o tamanho dos bytes que são manipulados por esses dois módulos foi alterado. Também existe uma FIFO intermediária que auxilia no tratamento dos dados para o modelo de entrada exigido pelo módulo Otsu.

4.2.1 Software

Send Data CPU-FPGA é um módulo dedicado ao empacotamento de 4 bytes da imagem em palavras de 32 bits. Esse tamanho é requisitado pelo protocolo RIFFA para a transmissão do *stream* de bytes ao hardware.

A imagem tem tamanho não fixo, por isso é enviado primeiro o número de linhas e depois de colunas, só então o conjunto de 4 pixels por palavra de 32 bits. Cada pixel originalmente era do tipo inteiro, ou seja, 32 bits. Porém esse tamanho pode ser reduzido já que o valor do pixel varia de 0 a 255, em binário a variação seria de 0000 0000 até 1111 1111. Então a representação pode ser feita com 1 *unsigned* byte (8 bits menos significativos dos 32 bits originais) sem perda de informação.

O pseudo-código 5 descreve o procedimento de conversão de imagem em *stream* de bytes e sua transferência ao hardware. Um *buffer* de zeros é criado de tamanho MxN/4 + 2 palavras (32 bits). Os dois primeiros 32 bits armazenam o número de linhas M e de colunas N. O restante do *buffer* contem os 4 bytes (4*8 bits = 32 bits) sequencias da imagem concatenada em palavras de

32 bits.

Algoritmo 5: 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)
```

.

A figura 4.4 mostra esse comportamento do processo de comunicação.

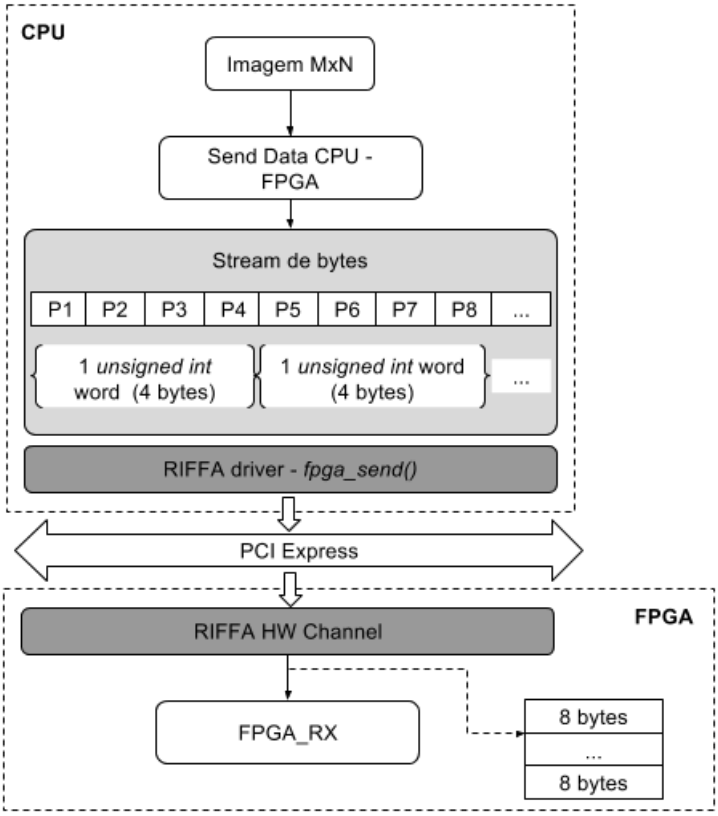

Figura 4.4 Arquitetura do processo de comunicação CPU-FPGA. A imagem de entrada de dimensões MxN é enviada ao módulo Send Data CPU-FPGA que converte-a em *stream* de 4 bytes sequenciais. A função fpga_send() envia os bytes ao barramento PCI Express que é recebido pelo módulo FPGA_RX no hardware.

O outro lado da comunicação de recebimento é descrito no algoritmo 6. Esse algoritmo mostra o recebimento do histograma de características HOG. A conversão é simples, o *buffer* tem 1.296 componentes de 64 bits de tamanho que são transferidos em um vetor de mesmo tamanho do tipo *long int* (64 bits) em software.

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

```
1 Entrada: buffer, um buffer de 1296 palavras de 64 bits;
2 Saída: HOG, um histograma de 1296 componentes long int
3 fpga_receive(buffer)
4 for i ← 0,1296 do
5 |   HOG(i) ← buffer[i];
6 end
```

A figura 4.5 mostra a nova arquitetura de comunicação dedicada ao recebimento do vetor HOG calculado em hardware.

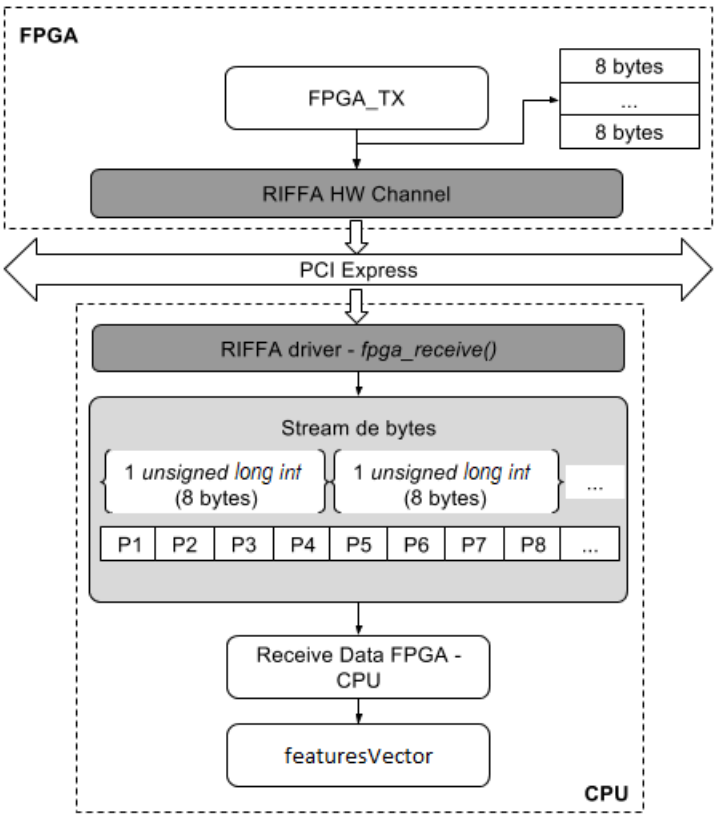

Figura 4.5 Nova arquitetura de comunicação dedicada ao recebimento do vetor HOG calculado em hardware ao software.

4.2.2 Hardware

No hardware, o *framework* RIFFA facilita a comunicação com o barramento através de sinais de controle. O link do PCI Express da placa DE2i-150 e o RIFFA determina a largura de transmissão de dados que é de 64 bits.

Com o auxílio do sistema de sinais do RIFFA foram desenvolvidos os módulos, FPGA_RX e FPGA_TX, o módulo FPGA_RX permanece o mesmo em relação a arquitetura anterior. Enquanto que o módulo FPGA_TX precisou ser modificado. Ambos módulos tem uma frequência de 125 MHz. O módulo FPGA_RX funciona como uma interface do canal RIFFA e o módulo BIA. Já o módulo FPGA_TX é uma interface entre o módulo HOG e o canal RIFFA.

A figura 4.6 mostra a máquina de estados do módulo FPGA_RX. No estado inicial S0, o controle passa para o estado seguinte S1 somente quando tiver transação válida no barramento e se o módulo informar, por meio do bit valid, que aquela informação é válida. Quando no estado S1, a primeira palavra de 64 bits é lida sendo que os primeiros 32 bits (riffa_data[31:0]) são o número de linhas e os outros 32 bits (riffa_data[63:32]) o número de colunas, seus respectivos valores são armazenados nos registradores rows e cols. No estado seguinte S2, é lida as palavras de 64 bits que contem 8 bytes, cada um representando um pixel da imagem. Esse estado se repete até a interface RIFFA finalizar indicando que a transmissão não é mais valida com o bit

riffa_rx = 0 e assim volta ao estado inicial S0.

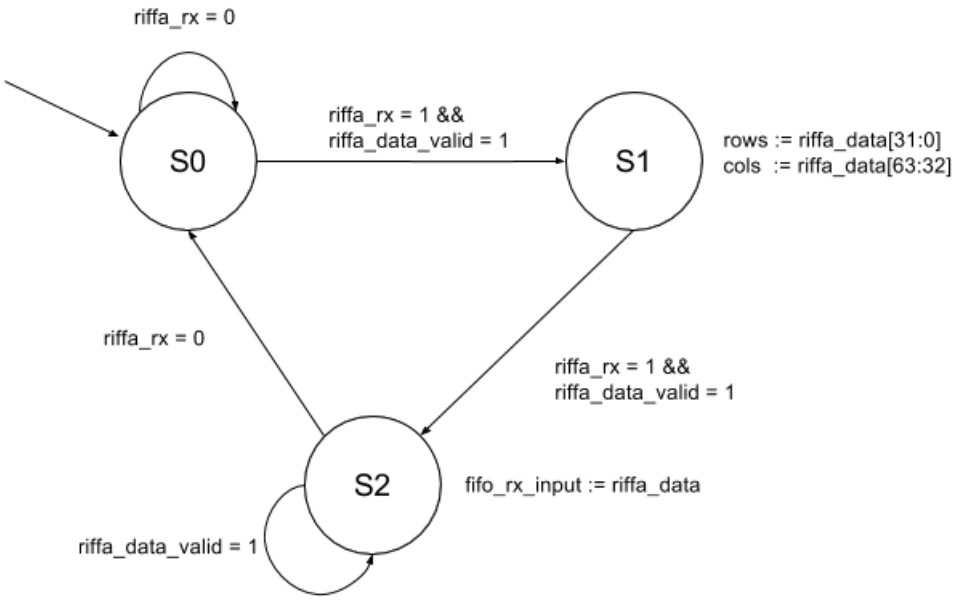

Figura 4.6 Máquina de estado da unidade de controle do módulo FPGA_RX.

A máquina de estados do módulo FPGA_TX é mais simples que a do módulo FPGA_RX. O módulo permanece no estado inicial quando o módulo FPGA_RX estiver usando o barramento, ou seja quando estiver recebendo dados. O estado só muda quando existir dados no FPGA_TX para serem enviados. O estado S1 é dedicado a enviar os 1296 componentes de 64 bits do vetor de características do HOG à CPU, um componente por vez. Depois de enviados todos os componentes o estado volta ao estado inicial S0. A imagem 4.7 resume essa descrição.

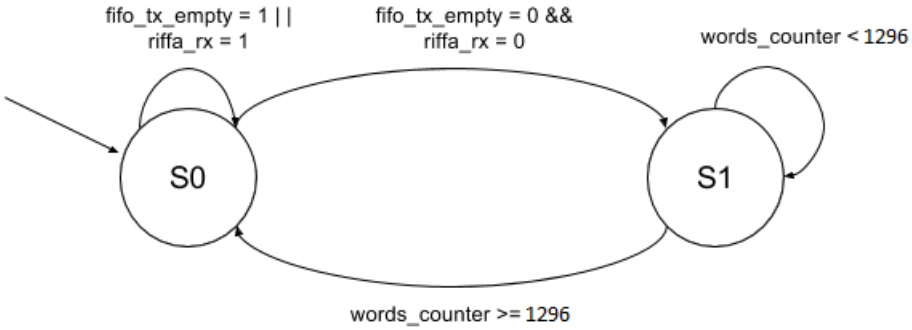

Figura 4.7 Máquina de estado da unidade de controle do módulo FPGA_TX.

4.3 Arquitetura proposta

Foi implementado os módulos Otsu e HOG em SystemVerilog. Esses módulos são representados por três arquivos. O histogramGrayScale calcula o limiar do Otsu e o otsuThreshold converte a

imagem em binário. Enquanto que o histogramBins se dedica a tarefa do HOG. A imagem 4.8 a seguir ilustra como esses módulos estão relacionados.

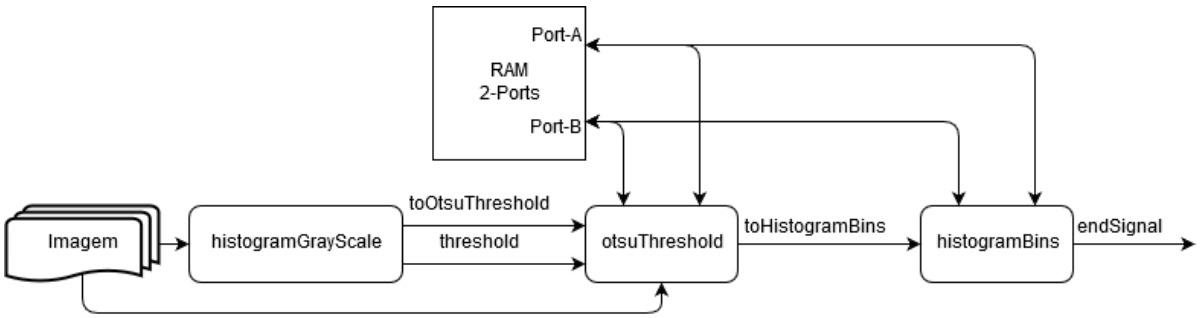

Figura 4.8 Diagrama de blocos da nova arquitetura com seus módulos implementados em hardware.

4.4 Módulo histogramGrayScale

histogramGrayScale é um módulo dedicado ao cálculo do histograma da imagem redimensionada de saída do módulo BIA. A entrada do módulo é uma palavra de 32bits dessa imagem, cada palavra tem 4 pixels. O módulo tem um vetor interno denominado histograma e um contator que garante o número certo de palavras da imagem recebida. Num primeiro momento o contador está zerado e o histograma também. A entrada do módulo é dividida em quatro bytes usando, operações bit a bit e *shift* e cada um desses 4 pixel é usado como índice do histograma e acrescentado uma unidade no seu respectivo componente do histograma. O histograma serve como um contador da quantidade de pixels de mesma intensidade, sendo cada componente do histograma de 0 a 254 representa uma intensidade diferente. Sabendo desse propósito, o histograma por essa razão possui um tamanho de 255, já que é o tamanho suficiente para armazenar todas as intensidade do pixel. Quando a contagem for maior ou igual a 4096 (128x128 / 4) significa que o módulo recebeu todos os pixels da imagem e passa para a próxima etapa. A imagem 4.9 mostra o diagrama de blocos do Otsu em hardware.

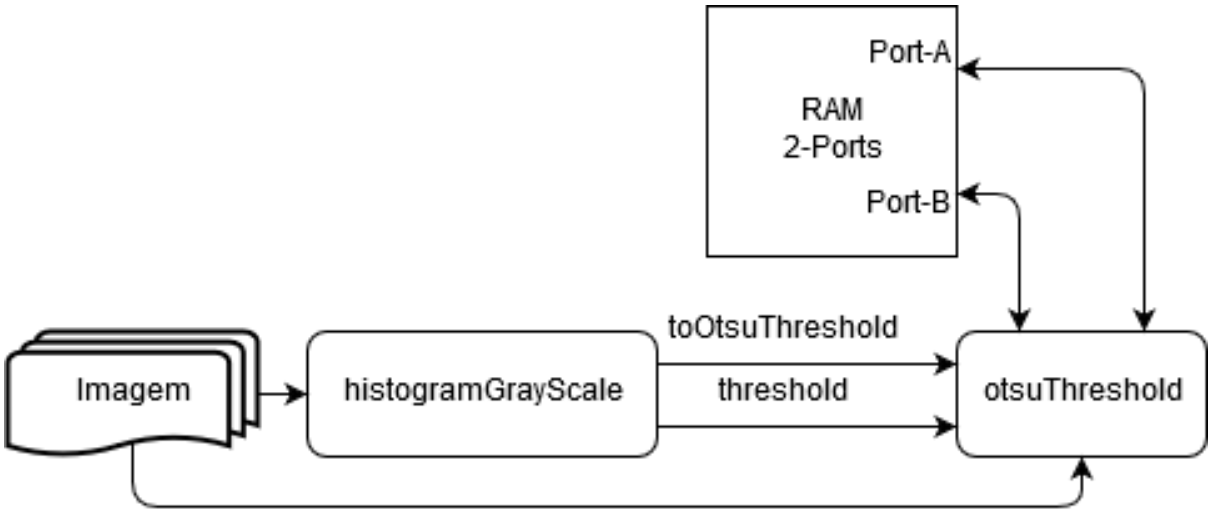

Figura 4.9 O módulo histogramGrayScale tem como input uma image e como saída o bit de ativação toOtsuThreshold e o limiar threshold.

Na etapa seguinte, cada valor do histograma é multiplicado por um peso, esse peso é proporcional ao seu índice divido por 255. Sendo assim, como cada pixel varia de 0 à 255, esses pesos variam de zero até 1. Depois de calculado cada um desses valores, ele é somado a uma variável chamada de weightedSum. Essa variável representa o somatório de todo o histograma multiplicado pelo seu respectivo peso.

Já calculado o somatório, o vetor histograma é percorrido novamente. Dessa vez para obter o limiar que maximiza a variância entre o plano de fundo e o caractere. Para cada componente do histograma, partindo do primeiro, é calculado o valor multiplicado do componente pelo seu peso como citado na etapa anterior. Esse valor é somado à variável backgroundSum e dividido pela quantidade de pixels acumulados até este índice, computando a média do plano de fundo. O resultado é atribuído a uma variável chamada de meanBackground. A sua variável oposta, meanForeground, é a média dos outros pixels restantes. A variância entre essa duas variáveis é o que se deseja maximizar, e o limiar é o índice da interação quando o valor dessa variância é máximo, dividido por 255. O limiar está entre o intervalo real [0, 1]. uma vez obtido esse limiar ele é disponibilizado na saída do módulo, mas somente quando o algoritmo percorrer todo o histograma é que o bit toOtsuThreshold é ativado executando o próximo módulo. A seguir o

pseudo-código 7.

Algoritmo 7: Calculo do limiar usado na limiariazação de Otsu

Entrada: *in*, a palavra de 32 bits contendo 4 pixels da imagem redimensionadas;
Saída: *threshold*, limiar;
$histogram = 0$
$backgroundSum = 0$
$maxVariance = 0$
for $count = 0,4096$ **do**
    $i1 = in >> 24$;
    $i2 = (in >> 16)\&hFF$;
    $i3 = (in >> 8)\&hFF$;
    $i4 = in\&hFF$;
    $histogram[i1]++$;
    $histogram[i2]++$;
    $histogram[i3]++$;
    $histogram[i4]++$;
    $count++$;
end
for $i = 0,255$ **do**
    $weightedSum = weightedSum + (i/255) * histogram[i]$;
end
for $j = 0,255$ **do**
    $wB = wB + histogram[j]$;
    $wF = 16384 - wB$;
    $backgroundSum = backgroundSum + (j/255) * histogram[i]$;
    $meanBackground = backgroundSum/wB$;
    $meanForeground = (weightedSum - backgroundSum)/wF$;
    $betweenClassVariance = wB * wF * (meanBackground - meanForeground)^2$;
    if $betweenClassVariance > maxVariance$ **then**
        $maxVariance = betweenClassVariance$;
        $threshold = (j/255)$;
    end
end
$toOtsuThreshold = 1$;

4.5 Módulo otsuThreshold

Enquanto o módulo histogramGrayScale recebe as palavras de 32 bits contendo 4 pixel da imagem redimensionada o módulo otsuThreshold grava as palavras na memória de duas portas RAM usando a porta B.

Quando o toOtsuThreshold é ativado o módulo otsuThreshold executa um *loop* de 4096 iterações, esse numero de *loops* é justificado pelo tamanho padronizado da imagem 128x128

dividido por 4. Em consequência disso, o módulo busca na memória 4 pixels por vez usando a porta A da memória RAM, sendo que esses pixels foram gravados inicialmente pelo otsuThreshold enquanto que o histogramGrayScale estava em execução. Esse módulo é dedicado a comparar cada pixel com o limiar, caso o pixel seja maior ou igual que o valor do limiar então a saída será 1, caso contrario 0. A saída dos 4 pixels são concatenados e atribuídos à saída que

será salvo na memória usando a porta B. Uma melhor descrição está no algoritmo 8.

Algoritmo 8: Retorna uma imagem binária

Entrada: *in*, a palavra de 32 bits contendo 4 pixels da imagem redimensionadas;
threshold, limiar calculado no histogramGrayScale;
Saída: *out*, imagem binária;
for $i = 0,4096$ **do**
    $address_b = i$;
    $data_b = in$;
    $wren_b = 1$;
end
for $j = 0,4096$ **do**
    $i1 = q_a >> 24$;
    $i2 = (q_a >> 16)\&hFF$;
    $i3 = (q_a >> 8)\&hFF$;
    $i4 = q_a\&hFF$;
    if $i1/255 \leq threshold$ **then**
        $aux = 1$;
    else
        $aux = 0$;
    end
    $out = out|aux << 24$;
    if $i2/255 \leq threshold$ **then**
        $aux = 1$;
    else
        $aux = 0$;
    end
    $out = out|aux << 16$;
    if $i3/255 \leq threshold$ **then**
        $aux = 1$;
    else
        $aux = 0$;
    end
    $out = out|aux << 8$;
    if $i4/255 \leq threshold$ **then**
        $aux = 1$;
    else
        $aux = 0$;
    end
    $out = out|aux$;
    $address_b = j$;
    $data_b = out$;
    $wren_b = 1$;
end
$toHistogramBins = 1$;

4.6 Módulo HistogramBins

O módulo HistogramBins foi implementado em hardware e se encarrega de obter o histograma de características HOG, o diagrama de blocos pode ser visto na figura 4.10.

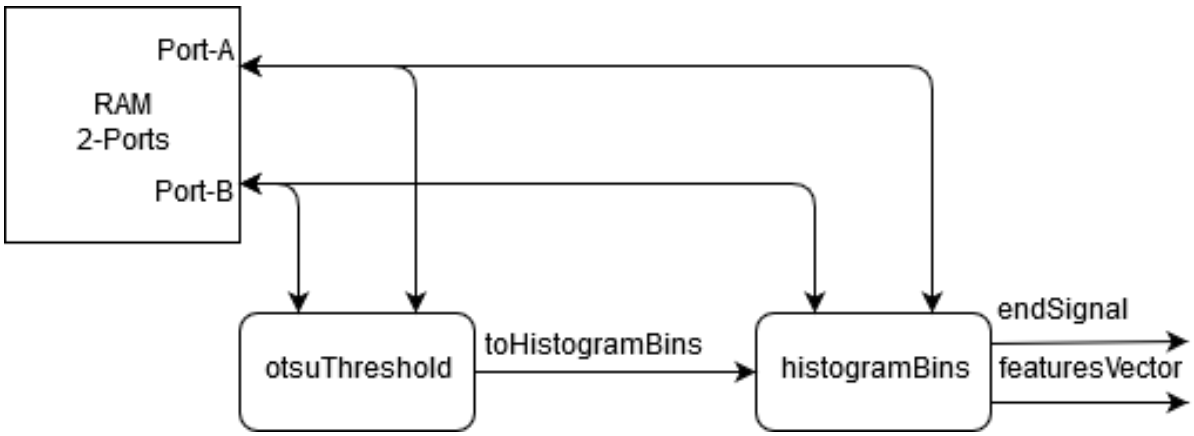

Figura 4.10 O módulo HistogramBins tem como input a image binária e como saída o vetor característica e seu bit que informa a saída válida.

O módulo divide a imagem binária de dimensão 128x128 em células de 18 pixels por 18. Para cada uma dessas células um histograma é criado. A memória RAM é usada para obter os valores anteriores e posteriores de cada pixel nas direções x e y para calcular o gradiente gx e o gy. Com gx e gy são obtidos a orientação e a magnitude. A magnitude serve para indicar a contribuição daquele pixel no histograma e também para contribuir aos intervalos vizinhos. Depois disso, todas as células são agrupadas por blocos contendo 2x2 células, para cada bloco os histogramas de cada célula são concatenados, resultando em um vetor de 36 componentes (9

intervalo do histograma para cada 2x2 células). O algoritmo é descrito pelo pseudo-código 9.

Algoritmo 9: Retorna um vetor de características HOG

```
1  Saída: featuresVector, um vetor de características descrevendo a imagem;
2  for each block do
3      for each cell do
4          for each pixel do
5              angle = getAngle(dx, dy);
6              weight = getMagnitude(dx, dy);
7              histogram = calculateCellHistogram(histogram, angle, weight);
8          end
9      end
10 end
11 for each block do
12     for each angle interval do
13         acumulador+ = histogram[block][angle]^2;
14     end
15     for each angle interval do
16         featuresVector = histogram[block][angle] * Q_r sqrt(acumulator);
17     end
18 end
```

O histograma de todos os blocos resulta em um vetor de 1296 componentes que são normalizados usando uma técnica de normalização de vetor para hardware chamada de *Fast inverse square root*. Essa formula é equivalente a equação 4.1 e afirma que para normalizar um vetor basta multiplicar todos os seus componentes pelo inverso da raiz quadrada da magnitude do vetor.

$$\text{Vetor normalizado: } \hat{v} = v\frac{1}{\sqrt[2]{||v||^2}} \tag{4.1}$$

$$\text{O inverso da raiz quadrada de } ||v||^2\text{: } \frac{1}{\sqrt[2]{||v||^2}} \tag{4.2}$$

em que, $||v||^2 = \sum_{i=1}^{36} v_i^2$

O algoritmo 10 mostra como foi implementado a função dedicada ao cálculo desse inverso.

Algoritmo 10: Calcula o inverso da raiz quadrada de um número

1 **Entrada:** *number*, um número real;
2 **Saída:** y, o inverso da raiz quadrada do número de entrada;
3 $x2 = number * 0.5$;
4 $y = number$;
5 $i = 0x5f3759df - (i >> 1)$;
6 $y = y * (1.5 - (x2 * y * y))$;
7 $y = y * (1.5 - (x2 * y * y))$;
8 $y = y * (1.5 - (x2 * y * y))$;

4.7 Memória de Duas Portas RAM

A memória usada foi uma memória RAM de duas portas sintetizada pela ferramenta Quartus II 14.1. As portas são chamadas de A e de B e cada uma das portas tem capacidade de ler e escrever. A capacidade de armazenamento total é de 131.072 bits (uma imagem de entrada tem dimensões 128x128 pixel e cada pixel um bytes, totalizando 131.072 bits). A memória trata os dados como palavras de 32 bits. A imagem 4.11 ilustra o módulo em hardware da memória RAM.

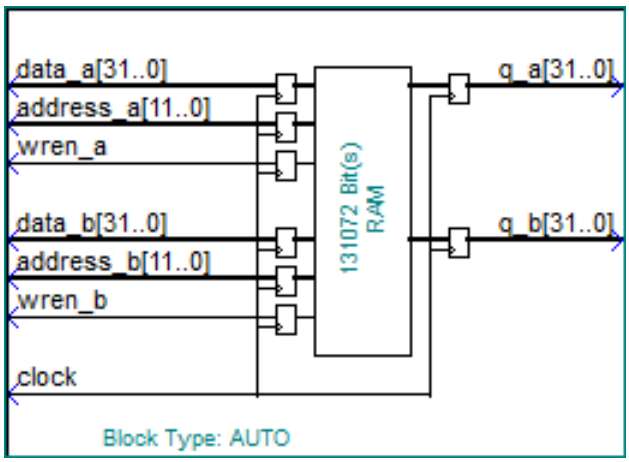

Figura 4.11 Módulo de memoria usada para armazenar uma imagem de 128x128x8 bits.

4.8 Ponto Fixo

Operações aritméticas em hardware são custosa, necessitando de muitos ciclos de clock ou um componente dedicado conhecido como Unidade de Ponto Flutuante. Para contornar esse problema foi adotado uma nova representação dos números reais. Essa representação utiliza partes dos bits de um tipo *int* (32bits) ou *long int* (64bits) para representar a parte real do número e os bits seguintes representam a parte decimal.

Para criar um tipo ponto fixo a partir de um número *int* é preciso fazer um *shift* para a esquerda. O número de *shift* depende da precisão desejada. Nos teste realizados, um *shift* de 24 resulta numa precisão de até 6 casas decimais enquanto que um *shift* de 27, uma precisão de até 8 casas. Certas operações aritméticas, como divisão e multiplicação deslocam ainda mais os valores para a esquerda, por isso é importante considerar o número de bits total da variável, para que os valores deslocados não sejam perdidos. Na imagem 4.12 abaixo mostra o resultado dos testes para diferentes *shift* no cálculo da variável weightedSum usado no módulo histogramGrayScale.

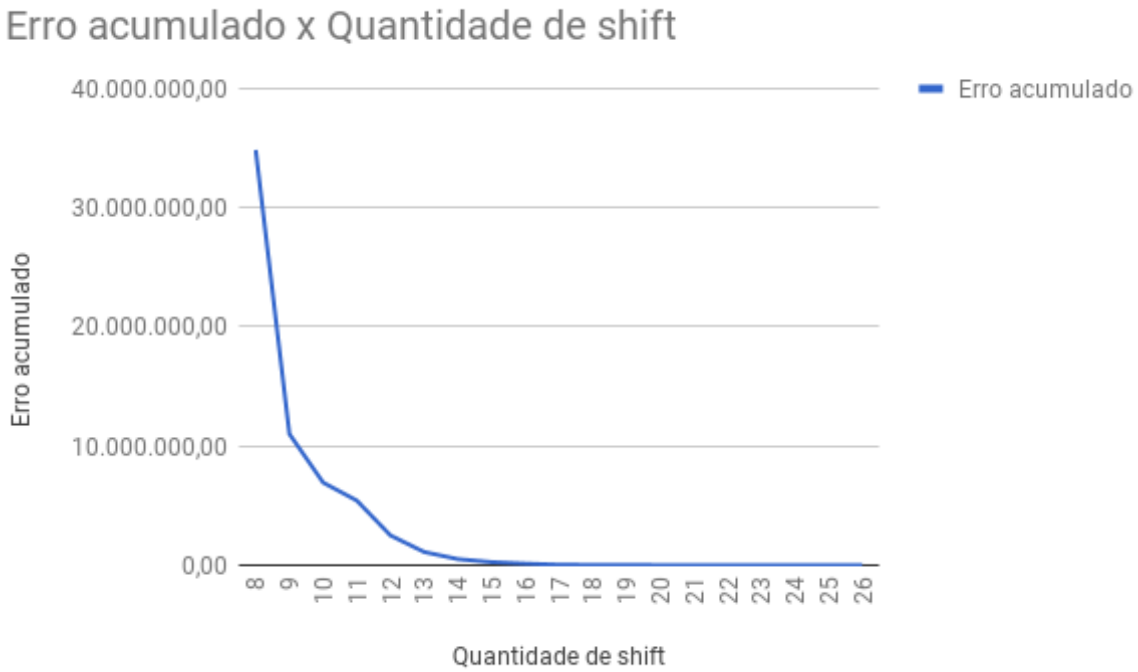

Figura 4.12 Gráfico que mostra o erro acumulado para diferentes *shift* no cálculo da variável weightedSum do histogramGrayScale.

As operações de adição e subtração são as mesmas que para os inteiros.

Para explicar a divisão segue um exemplo. Para dividir 1 por 2, primeiro convertermos os dois tipos para ponto fixo, adotando um *shift* de 8 bits. Obtendo para o 1, 256 e para o 2, 512. Para fazer a divisão, o numerador é deslocado novamente pela mesma quantidade (8 bits). Esse novo *shift* resulta 65536. Em seguida a divisão é feita. 65536 / 512 = 128. Como pode ser vista na equação 4.3.

$$\begin{aligned} \text{Para calcular:} 1/2 &= 0.5 \\ 1 << 8 &= 256 \\ 2 << 8 &= 512 \\ 256 << 8 &= 65536 \\ 65536/512 &= 128 \end{aligned} \tag{4.3}$$

128 representa 0.5 em ponto fixo. A seguir é explicado a conversão para obter o número real.

Para verificar se a operação está correta convertermos o valor para o número real. A parte inteira é obtida com o *shift* para a direita na mesma quantidade do *shift* adotado. No exemplo acima, 128 » 8 = 0. Ou seja, a parte inteira é 0. Para calcular a parte decimal calcula-se o total que é possível representar, se foi usado um *shift* de 8, então esses últimos 8 bits são dedicados a parte real. O máximo que se pode representar é o número 255 nessa parte. No exemplo acima, a representação em binário é 1000.0000 ou 128 em decimal. Dividindo 128 / 256 resulta em 0.5. Ou seja, 128 é a representação de 0.5 em ponto fixo.

Outra peculiaridade é a forma como é realizada a multiplicação. Usando outro exemplo, 0.5 * 0.5. Sabemos que 128 é 0.5 em ponto fixo. Primeiro se faz a multiplicação normalmente, 128 * 128 = 16384. E depois faz o *shift* para a direita, 16384 » 8 = 64. O número 64 está na parte decimal, convertendo para ponto flutuante: 64 / 256 = 0.25.

$$\begin{aligned} \text{Para calcular:} 0.5 * 0.5 &= 0.25 \\ 128 * 128 &= 16384 \\ 16384 >> 8 &= 64 \\ 64/256 &= 0.25 \end{aligned} \tag{4.4}$$

Existem casos de multiplicação que o algoritmo trata para garantir o máximo possível de precisão. É o caso do exemplo 4.5 a seguir. Fazendo multiplicar 0.0625 por 0.03125, sendo suas representações em ponto fixo, 16 e 8 respectivamente, é obtido 128. A etapa seguinte seria fazer o *shift* para a direita, 128 » 8, isso resulta em 0. Era esperado o número real 0.001953125 em representação de ponto fixo. A menor representação, além do zero, é a 1 / 256 = 0.00390625. Para melhorar a acurácia, nos casos de multiplicação que resultarem no sétimo bit ativo, é deslocado esse bit para a esquerda. Com o objetivo de que na etapa seguinte, esse bit acrescente 0.00390625 ao invés de 0.001953125 que não pode ser representado em ponto fixo.

$$\begin{aligned} \text{Para calcular:} 0.0625 * 0.03125 &= 0.001953125 \\ 16 * 8 &= 128 \\ 128 >> 8 &= 0(!) \\ 128 + (128 << 1) &= 128 + 256 = 384 \\ 384 >> 8 &= 1 \\ 1/256 &= 0.00390625 \end{aligned} \tag{4.5}$$

Capítulo 5

Resultados

O resultados foram obtidos usando o sistema operacional Windows 10. As ferramentas utilizadas foram o Quartus II versão 14.1 e o ModelSim 10.3c.

Inicialmente foi proposto uma arquitetura híbrida usando a placa DE2i-150. Parte da aplicação seria executada no processador ATOM N2600 dual-core 1.6GHz e outra parte na FPGA da placa, de modelo Cyclone IV EP4CGX150DF31.

Os arquivos SystemVerilog compilam e executam com o ModelSim, mas não foi possível sintetizar o arquivo .sof com o módulo Riffa no Quartus.

Para validação do algoritmo em SystemVerilog foi feito um *testbench* usando arquivo texto dos pixels das imagem do conjunto Chars74K-15. Nesses arquivos foram escritas as sequências de pixels obtidas de cada imagem depois do módulo Resize. Ou seja, com imagens já redimensionadas e convertidas para a escala de cinza. Esse conjunto de imagens tem no total 1860 imagens, 930 de treino e 930 de teste. Todas essas imagens foram redimensionadas e convertidas em escala de cinza e os seus 128x128 pixels impressos em um arquivo de saída. Esse arquivo foi usado como estrada para o *testbench*. A figura 5 a seguir ilustra o *testbench*.

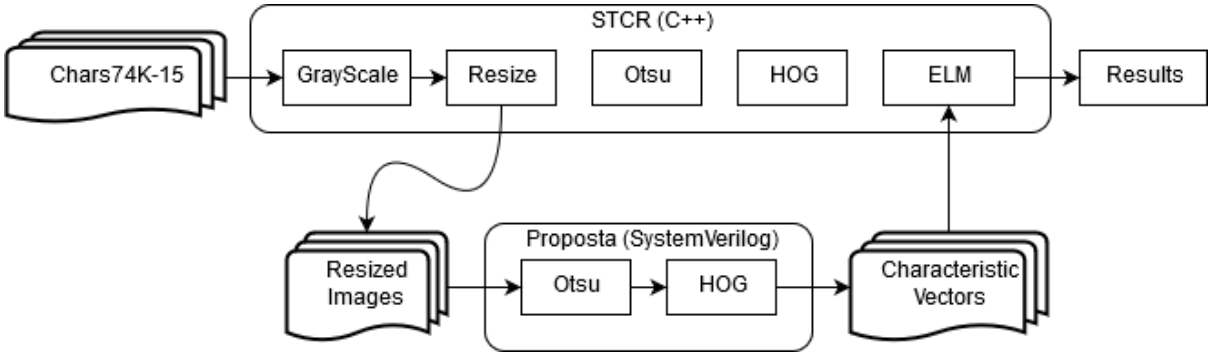

Figura 5.1 Representação do test bench realizado.

O módulo Otsu computa uma imagem binária e em seguida HOG calcula um histograma de características. Esse histograma é impresso em um arquivo de saída. Cada imagem de entrada tem um histograma que é um vetor de 1296 componentes. Cada componente é um ponto fixo de *shift* 27, de tamanho 128 bits para garantir maior precisão de casas decimais quando comparado

com o tipo *double*.

No Otsu, o ponto fixo teve *shift* de 24 bits para não interferir no calculo do limiar. Caso fosse 27 bits no Otsu, as variáveis de 128 bits usadas para o cálculo do limiar não seriam suficientes para conter o valor deslocado depois de algumas operações aritméticas.

Essa saída é comparada com os vetores de características impressos pelos respectivos módulos em C++. Foi criada uma aplicação em Python que lê todos os pontos fixos de um arquivo de saída e converte em ponto flutuante. Outro programa em Python compara cada saída desse arquivo com o resultado do algoritmo em C++ e calcula o erro. A diferença para cada elemento é somada em uma variável e impressa no final do arquivo. No *testbench* a diferença absoluta total entre todos os componentes foi de aproximadamente 1388 para todas as 1830 imagens. Diferença acumulada de 0.75 para cada imagem. Isso prova que o ponto fixo de 128 bits com *shift* de 27 resulta em uma boa precisão.

Esse arquivo que contém os vetores de saída foi usado no programa em C++. Cada elemento foi convertido em ponto flutuante e acrescentado na rede ELM, as 930 primeiros vetores foram inseridos como treinamento e o restante como teste. A acurácia do algoritmo permaneceu a mesma que a acurácia obtida testando somente o algoritmo em C++. A seguir o resultado na tabela 5.1, a primeira coluna é o número de neurônio na camada intermediária, a segunda a taxa de acerto e a terceira coluna é o tempo de execução para aquela configuração. A segunda tabela 5.2 mostra os resultados originais do algoritmo em C++.

Tabela 5.1 Performance do método de STCR com o vetor HOG calculado no SystemVerilog.

Número de neurônio	Acurácia (%)	Tempo de Execução (s)
1000	16.666667	1.452
2000	57.849462	2.855
3000	61.182796	5.006
4000	63.655914	5.638
5000	63.225806	7.124
6000	63.763441	8.499

Tabela 5.2 Performance do método de STCR com o vetor HOG calculado em C++.

Número de neurônio	Acurácia (%)	Tempo de Execução (s)
1000	19.139785	1.437
2000	57.526882	2.936
3000	62.043011	4.691
4000	63.118280	6.777
5000	62.795699	7.711
6000	63.763441	9.362

Outro teste foi realizado, como mostra a figura 5. Foi testados em conjunto a FIFO de entrada usada pelo módulo Resize, o próprio Resize, a FIFO intermediário entre o Resize e o Otsu, o Otsu, o HOG e a FIFO de saída. Foram comparadas a saída do ModelSim com a saída da aplicação em C++ para os valores obtidos do Resize em hardware. Os erros acumulados foram de 0.010 para a primeira imagem do banco de dados.

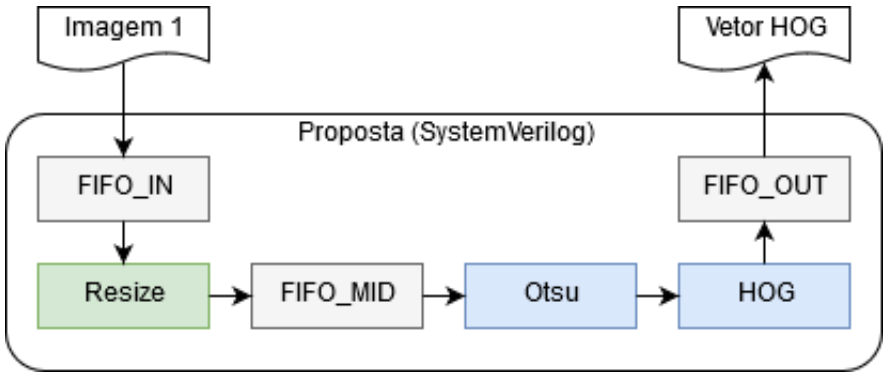

Figura 5.2 Teste para validação do Resize e das FIFOs.

Capítulo 6

Conclusão

6.1 Conclusão

Neste trabalho foram implementados os algoritmos de Limiarização de Otsu e a técnica de HOG em SystemVerilog. Os módulos foram testados no ModelSim para todas as imagens e conseguiram resultados desejados. Como foi visto na tabela 5.1 e 5.2 acurácia permaneceu a mesma. O número de neurônios testado foi de até 6000, no trabalho de Lima [1] foi concluído que esse número garante melhor acurácia em baixo tempo de execução.

No segundo teste, foi validado a integração dos novos módulos com os outros módulo implementados no trabalho de Luiz. Porém não foi possível ser executado na placa, os novos módulos usam declarações bloqueantes (=) numa lógica sequencial, sendo que para hardware é recomendado declarações não-bloqueantes (<=) numa lógica de máquina de estados. Em função disso não foi possível compilar no Quartus e em consequência não foi possível comparar a otimização com o software de referência e também não foi possível avaliar o tempo de processamento médio.

6.2 Trabalhos futuros

Como trabalho futuro temos a readequação do algoritmo para prototipação na placa junto com o projeto já implementado em hardware do módulo Resize. Outra sugestão seria implementar o ELM, esse módulo é o segundo que mais consome tempo do algoritmo depois do Resize.

Referências Bibliográficas

[1] L. Júnior, "Hardware Acelerador da Técnica de Reconhecimento de Caracteres em Imagens de Cenas Naturais," in *Programa de graduação em engenharia da computação*, Universidade Federal de Pernambuco, Centro de Informática, 2017.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.