NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

DIOGO MIGUEL RODRIGUES FERREIRA

Licenciado em Ciências da Engenharia Eletrotécnica e de Computadores

BIBLIOTECA GENÉRICA DE PRÉ-PROCESSAMENTO DE IMAGEM EM FPGA APLICADA A SISTEMAS DE VISÃO INDUSTRIAL

MESTRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

Universidade NOVA de Lisboa
Novembro, 2021

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

BIBLIOTECA GENÉRICA DE PRÉ-PROCESSAMENTO DE IMAGEM EM FPGA APLICADA A SISTEMAS DE VISÃO INDUSTRIAL

DIOGO MIGUEL RODRIGUES FERREIRA

Licenciado em Ciências da Engenharia Eletrotécnica e de Computadores

Orientador: Filipe de Carvalho Moutinho
Professor Auxiliar, Universidade NOVA de Lisboa

Coorientador: Pedro Deusdado
Hardware Supervisor Engineer, Introsys SA

Júri:

Presidente: Rui Manuel Leitão Tavares
Professor Auxiliar, Universidade NOVA de Lisboa

Arguente: Luís Filipe dos Santos Gomes
Professor Associado com Agregação, Universidade NOVA de Lisboa

Vogal: Filipe de Carvalho Moutinho
Professor Auxiliar, Universidade NOVA de Lisboa

MESTRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

Universidade NOVA de Lisboa
Novembro, 2021

Biblioteca Genérica de Pré-Processamento de Imagem em FPGA aplicada a Sistemas de Visão Industrial

Agradecimentos

Durante a realização da presente dissertação contei como o apoio de diversas pessoas e instituições às quais estou bastante grato.

Agradeço à Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa pelos 5 anos inesquecíveis, pelas amizades que ganhei, pelas experiências que vivi e pelos conhecimentos que adquiri. Não podia estar mais orgulhoso pelo desfecho deste ciclo incrível.

Agradeço à Introsys SA pela oportunidade de realizar a dissertação num clima empresarial. Foi um desafio enorme devido à presente pandemia e estou bastante grato pela confiança depositada em mim.

Agradeço ao meu orientador Professor Filipe Moutinho por todo o apoio que me deu durante mais de um ano, pela disponibilidade, pela boa disposição e pelas múltiplas e longas horas de reuniões para desenvolver este projeto. Não consigo agradecer o suficiente por tudo o que me ensinou, aprendi mesmo imenso consigo e saio desta etapa melhor profissional graças a si. Agradeço também ao Professor João Pedro Carvalho pela implementação dos algoritmos em GPU e medição dos tempos de execução (secção 4.15). Obrigado.

Agradeço a toda a equipa do departamento de inovação da Introsys SA. Em especial ao meu coorientador Pedro Deusdado pelas sugestões e pelos comentários assertivos, à Raquel Caldeira pela confiança depositada em mim e ao Fábio Miranda pela ajuda nos testes em aplicações reais. Obrigado a todos.

Agradeço o apoio do financiamento no âmbito do projeto CheckMate, com a referência n.º 045077 (POCI-01-0247-FEDER-045077), financiado pelo Compete, Portugal 2020. Feder – Projetos de I&D Individuais.

Um agradecimento especial aos meus pais e ao meu irmão pelo apoio incondicional, por se preocuparem constantemente comigo, pelas condições que me proporcionaram, pelas dicas e sugestões, por me animarem quando eu mais precisava, pelos sorrisos e risos, pela força que me deram e principalmente por todos os ensinamentos que fazem de mim o que sou hoje. Muito obrigado.

Agradeço à minha tia e aos meus avós pelos telefonemas, pela preocupação e pelo apoio constante. Foram peças fulcrais na minha determinação para acabar este projeto. Agradeço também a todos os meus amigos que me acompanharam durante este período,

em especial ao Ricardo Margarido, Diogo Banha e Tiago Castelo que me proporcionaram ótimos momentos e experiências ao longo destes anos. Foram uma das partes mais importantes durante todo o meu percurso académico. Obrigado a todos.

Resumo

A evolução tecnológica e a necessidade da sociedade atual possuir produtos de maior qualidade, provocou um aumento de complexidade nos sistemas de visão industrial implicando a integração de mais *hardware* na sua constituição com a finalidade de melhorar a sua eficiência. Na presente dissertação, o objetivo consiste na implementação de uma biblioteca genérica de métodos para execução em FPGA, em tempo real, com a finalidade de diminuir o tempo de processamento de imagem em sistemas de visão. O seu desenvolvimento teve como base a especificação de filtros em VHDL e a sua implementação na plataforma *Zybo Z7-20*. As *frames* processadas são transmitidas para CPU via *Ethernet* (UDP), possibilitando a integração do projeto em aplicações reais e a validação dos métodos. Foi realizada uma comparação entre o tempo de processamento de todos algoritmos desenvolvidos em duas plataformas (CPU e FPGA), assim como, a integração num sistema de visão industrial. Os resultados obtidos demonstraram ser positivos, visto que, a execução do pré-processamento em FPGA em tempo real acrescenta um atraso à imagem original na ordem dos nanossegundos, enquanto que, em CPU existe um acréscimo de tempo na ordem dos milissegundos para processar uma *frame*. Por fim, foi também realizada a comparação de tempos com uma solução baseada em GPU, na qual, se verificou que quando executado o pré-processamento em FPGA são obtidos melhores resultados.

Palavras-chave: Sistemas de visão industrial, Pré-processamento de imagem, FPGA, VHDL

Abstract

The technological evolution and society need to own the best quality products induced an increase in industrial vision systems complexity requiring more hardware to improve its efficiency. The objective of this work is the development of a generic pre-processing FPGA library, to accelerate real time industrial vision systems. Its development was based on the design of VHDL filters, implemented on a Zybo Z7-20 platform. The processed frames are transmitted to a CPU by Ethernet protocol (UDP) to enable the project integration in real applications and the methods validation. The execution time of all filters was compared in two platforms (FPGA and CPU) followed by the project integration in an industrial vision system. The obtained results were positive, where the FPGA solution in real-time only adds a nanoseconds range delay to the execution time of the original image, while the CPU solution adds a milliseconds range delay to process a frame. Lastly, a comparison of execution times with a GPU-based solution was also performed, in which it was conluded that the FPGA pre-processing algorithms achieve better results.

Keywords: Industrial vision systems, Image pre-processing, FPGA, VHDL

Índice

Índice de Figuras

Índice de Tabelas

Siglas

AXI4	Advanced Extensible Interface 4
BRAM	Block RAM
CLB	Configurable Logic Block
CPU	Central Process Unit
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
GIL	Generic Image Library
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HDMI	High-Definition Multimedia Interface
HLS	High-Level Synthesis
IOB	Input/Output Block
IP	Intellectual Property
OpenCL	Open Computing Language
OpenCV	Open Source Computer Vision Library
RGB	Red Green Blue
RTL	Register Transfer Level
SAD	Sum of Absolute Differences
SB	Switch Box
SoC	System on Chip

UART Universal Asynchronous Receiver-Rransmitter
UDP User Datagram Protocol
USB Universal Serial Bus

VHDL VHSIC Hardware Description Language

Capítulo 1

Introdução

Este é um capítulo de carácter introdutório para contextualizar o leitor quanto ao tema da presente dissertação. Na secção 1.1 é apresentada a motivação e o contexto do trabalho, na secção 1.2 a contribuição ao problema encontrado, e por último, na secção 1.3 é apresentada a estrutura do documento.

1.1 Contexto e Motivação

O conceito de visão foi evoluindo ao longo do tempo, este começou por estar apenas ligado à capacidade do ser humano observar o meio em seu redor. Conceitos como, noção de perspetiva, orientação e análise do meio envolvente estão relacionados não só com a visão humana, mas também, com todos os seres vivos. Várias tarefas do quotidiano humano são realizadas através desta capacidade, nas quais, se encontram algumas semelhanças com os sistemas de visão implementados atualmente. Uma simples travessia de estrada através de uma passadeira por parte de um peão requer o auxílio da capacidade de visão. Primeiramente, a passadeira é localizada e é feita uma análise do meio envolvente, de seguida, é tomada a decisão de atravessar ou não a estrada, consoante a análise anterior.

Um dos passos na evolução do conceito de visão deu-se quando este foi relacionado com câmaras e computadores, ou seja, com tecnologia. Uma câmara através da captação de uma fotografia ou vídeo, consegue simular a perspetiva humana. A deteção de objetos, contagem de peças e deteção de defeitos foram algumas das aplicações realizadas com a combinação destes conceitos. A partir destas soluções, implementadas em ambientes pouco complexos, o uso deste tipo de sistemas foi relacionado com a indústria, para assim, surgirem os sistemas de visão industrial [1]. Áreas como a robótica e automação têm cada vez mais impacto na atualidade. A elevada industrialização a nível mundial, implica que exista uma competição crescente entre empresas para alcançarem o sucesso. A qualidade

do produto é no momento atual um fator decisivo na satisfação dos consumidores, daí, a automação ter tido um papel tão decisivo na revolução industrial [2, 3].

A inspeção visual na indústria é uma das fases mais importantes no processo de manufatura de um produto. Este tipo de inspeção era realizada por humanos especializados, que apesar de especialistas na área, são propensos a erros quer pelo cansaço, repetibilidade ou complexidade exigida nas tarefas. Com o objetivo de os diminuir, a automação na indústria promoveu uma maior qualidade do produto através da introdução de sistemas de visão industrial, onde as tarefas de inspeção são agora executadas repetidamente por máquinas, diminuindo a ocorrência de erros nas linhas de montagem [4].

Durante o processo de funcionamento de um sistema de visão, é captada uma imagem via câmara, na qual, são efetuadas modificações com o intuito de facilitar a extração de informação relevante. Neste tipo de sistemas, as características de uma imagem são obtidas através de processamento de imagem.

1.2 Problema e Contribuição

A dissertação apresentada neste documento pretende relacionar a área de processamento de imagem através de *hardware* com os sistemas de manufatura industrial. Devido à necessidade de maximizar a produção com produtos de melhor qualidade, existem aplicações industriais cujos processos de fabrico requerem tempos de ciclo cada vez mais apertados. Estes sistemas apresentam complexidade elevada, o que implica a integração de mais *hardware* na sua constituição de forma a cumprir com os requisitos pretendidos.

Tendo em conta este problema, a presente dissertação tem como **objetivo o desenvolvimento de uma biblioteca genérica de filtros de pré-processamento para execução em Field-Programmable Gate Array (FPGA)**. A presente dissertação, proposta pela empresa Introsys, procura aumentar a eficiência de sistemas de visão, através da diminuição do tempo de processamento de imagem.

Na atualidade, existem vários sistemas de visão industrial que contêm FPGA na sua constituição. O potencial menor consumo de energia e a elevada capacidade de processamento, são alguns dos argumentos a favor da utilização de *hardware* nestes sistemas.

Alguns exemplos de sistemas onde esta biblioteca poderá ser aplicada são: *See-Q* [5] - análise dos parâmetros de aplicação de cordões de cola estruturais, em tempo real, durante a sua deposição em painéis automóveis; *CheckMate* [6] – análise de características após a montagem final do produto; *CONTIGO* [6] – contagem do número de peças presentes num contentor para extração. A validação da presente dissertação, será feita através da comparação de tempos de processamento entre um dos sistemas referidos (CPU) e a solução híbrida proposta (CPU e FPGA). Futuramente é pretendido que a carga de processamento seja divida entre Central Process Unit (CPU) e FPGA, de forma a aumentar a eficiência destes sistemas.

1.3 Estrutura do Documento

O documento tem a seguinte estrutura:

Capítulo 2 - Descrição do estado da arte relativo ao processamento de imagem através de FPGA aplicado aos sistemas de visão industrial. É apresentada uma pequena introdução à técnica de processamento de imagem, assim como, os *softwares* utilizados para o conseguir (2.1). De seguida, é introduzida a plataforma FPGA e o seu contributo para o processamento de imagem (2.2). É apresentada uma introdução à sua história e composição, de seguida, são analisadas as vantagens e desvantagens de várias plataformas que possibilitam o processamento de imagem, e por último, são apresentados três métodos de configuração de FPGA. A secção seguinte deste capítulo é destinada aos sistemas de visão industrial (2.3), onde é feita uma introdução aos mesmos e a apresentação de algumas soluções existentes bem como o contributo que as FPGA podem ter neste tipo de sistemas. Por último, é feita uma análise (2.4), onde são expostos os filtros mais utilizados e as bibliotecas já existentes, tendo em conta as secções anteriores.

Capítulo 3 - Descrição do processo de desenvolvimento da biblioteca de métodos de pré-processamento de imagem. Em primeiro lugar é realizada uma introdução ao projeto (3.1). De seguida, é descrito o material utilizado (3.2) e a arquitetura base do projeto, sendo explicitados os módulos constituintes da mesma, assim como, o protocolo de comunicação (3.3). Posteriormente, são explicitados os métodos de implementação dos filtros desenvolvidos (3.4), e por fim, a transmissão (3.5) e receção (3.6) de *frames* via *Ethernet* (UDP), de modo a ser possível integrar o projeto em aplicações reais, assim como, a validação dos filtros.

Capítulo 4 - Descrição do processo de validação dos métodos desenvolvidos e integração do projeto em aplicações reais. Em primeiro lugar, é apresentado o processo de validação onde são explicitadas as duas abordagens utilizadas para a extração de *frames*, assim como, os métodos de medição do tempo de processamento e erro absoluto entre implementações em FPGA e CPU (4.1). De seguida, são apresentados os resultados da validação de cada método (4.2-4.11). Posteriormente, é explicitada a integração em aplicações reais, na qual se destacam a sequência de filtros utilizada (4.12) e os resultados obtidos (4.13). Por fim, é apresentada uma secção de discussão dos resultados (4.14) e a comparação dos mesmos com uma solução baseada em Graphics Processing Unit (GPU) (4.15).

Capítulo 5 - Conclusão acerca dos resultados obtidos e possível trabalho futuro.

Capítulo 2

Estado da Arte

O foco deste capítulo é a apresentação dos métodos de processamento de imagem utilizados em sistemas de visão industrial através de FPGA. Para uma melhor perceção dos conceitos abordados, este capítulo foi dividido em quatro partes. Primeiramente, é apresentada uma introdução às tecnologias utilizadas em aplicações de processamento de imagem (2.1), de seguida, é analisado o contributo da plataforma FPGA neste tipo de aplicações (2.2), posteriormente, é exposta a relação existente entre os sistemas de visão industrial e o processamento de imagem através de FPGA (2.3), por último, são apresentadas algumas considerações acerca deste capítulo tendo em conta o objetivo da dissertação, assim como, bibliotecas de funções para FPGA existentes (2.4).

2.1 Processamento de Imagem e suas Tecnologias

Esta secção tem como foco contextualizar o leitor acerca do conceito de processamento de imagem. Em 2.1.1, é descrito o conceito de imagem e o porquê destas serem processadas. Em 2.1.2, 2.1.3 e 2.1.4 são apresentadas algumas ferramentas de *software* utilizadas para o desenvolvimento de aplicações nesta área.

2.1.1 Introdução ao Processamento de Imagem

O conceito de imagem digital pode ser definido como um conjunto de informação codificada. É composta por um conjunto de píxeis agrupados em linhas e colunas, nas quais, cada pixel tem um valor finito definido. Possui características como: resolução e tipo de imagem. A resolução indica a quantidade de detalhe da informação presente numa imagem. O tipo influencia a tipologia de informação presente na mesma. Apesar de uma imagem digital ter a possibilidade de dispor de uma vasta quantidade de informação,

pode ser necessário estudar uma parte especifica da mesma. A técnica que possibilita a extração de uma determinada característica de uma imagem digital é denominada de processamento de imagem.

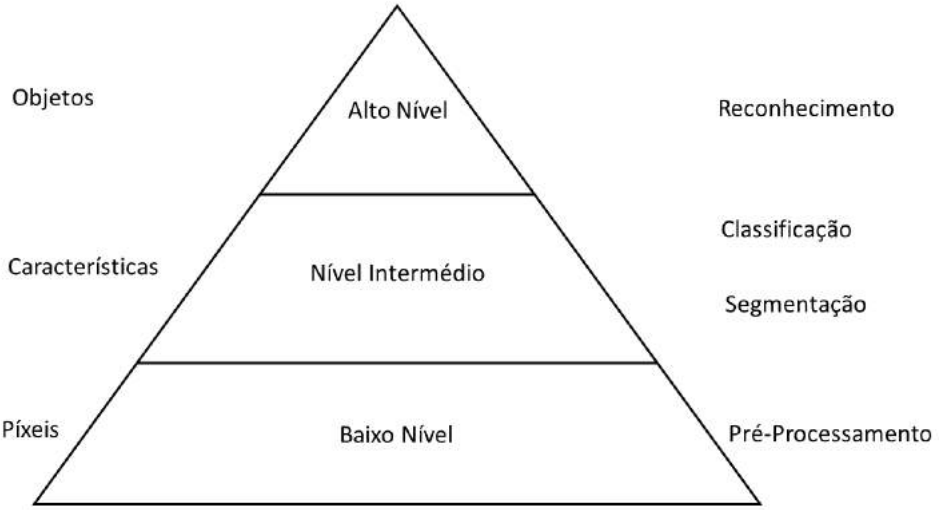


Figura 2.1: Esquema de operações de processamento de imagem (Adaptado de: [7])

Para uma melhor perceção do conceito desta técnica, na figura 2.1 estão representadas hierarquicamente algumas das suas operações possíveis. No nível mais baixo, são representadas as tarefas destinadas a realçar a informação considerada relevante na imagem, exemplo disso é a remoção de ruído. No nível intermédio o foco centra-se na extracção de características - segmentação e classificação de objetos. Por fim, no nível mais elevado, encontram-se as operações de reconhecimento de objetos [7].

2.1.2 C++

Nesta secção, vai ser introduzida a linguagem *C++*, o seu contributo para o processamento de imagem (2.1.2.1) e alguns exemplos de aplicações existentes (2.1.2.2).

2.1.2.1 Introdução ao C++

A linguagem *C++* dispõe de enumeras classes para diversos tipos de áreas de programação. Uma dessas áreas é o processamento de imagem, na qual pode ser usada a classe *Boost*. Esta é uma coleção de bibliotecas em linguagem *C++* que contém na sua constituição vários tipo de estruturas de dados e algoritmos. Uma das bibliotecas incluídas é a *Generic Image Library (GIL)*, que possui funções genéricas de processamento de imagem como: leitura e escrita de imagens, conversão de cor, transformações morfológicas e cálculo de histogramas. Esta biblioteca suporta imagens do tipo *.jpeg* e *.png* [8].

2.1.2.2 Aplicações de Processamento de Imagem baseadas em C++

A passagem de uma imagem em tons de cinzento para preto e branco pode ser importante na exclusão de informação não relevante de uma imagem. A realização desta operação requer a definição de um nível de referência. No varrimento dos píxeis, se o seu valor inicial for inferior ao valor de referência teremos um pixel branco como resultado, caso contrário é obtido um preto. Uma variante desta operação é dada pelo filtro *Otsu*. Numa operação de binarização comum, a referência é escolhida previamente, contudo, o algoritmo do filtro *Otsu* utiliza uma imagem à qual é aplicado o cálculo do melhor nível de referência, proporcionando assim, melhores resultados em relação à operação anterior. Este cálculo é realizado a partir da análise de um histograma da imagem a filtrar. Em [9], um exemplo deste tipo de filtro foi implementado em linguagem *C++*. Os resultados obtidos pelos autores, mostram que é possível implementar este algoritmo através desta linguagem de programação, e que o desempenho do mesmo depende da distribuição dos valores no histograma da imagem inicial.

A contagem de veículos presentes numa *frame* de um vídeo pode ser realizada a partir de processamento de imagem baseado em linguagem *C++*. Um desses exemplos é o algoritmo sequencial implementado em [10]. Este pode ser descrito pela seguinte sequência:

1. Conversão Red Green Blue (RGB) para tons de cinzento
2. Aplicação de um filtro de realce de contornos (*Sobel*)
3. Definição da zona de deteção (engloba o máximo de veículos possível)
4. Aplicação de um filtro *Kalman* (estima a posição de um determinado veículo numa *frame* posterior do vídeo)
5. Aplicação de um algoritmo de identificação do tipo de veículo
6. Contagem do número de unidades

Os resultados obtidos pelos autores mostram que através desta sequência de filtros implementados em linguagem *C++*, foi possível obter o número de veículos presentes numa *frame* de um vídeo com percentagens de erro baixas (cerca de 5%).

2.1.3 OpenCV

Nesta secção, irá ser introduzida a plataforma *Open Source Computer Vision Library (OpenCV)*, os métodos que constituem a sua biblioteca (2.1.3.1) e a sua aplicação em projetos existentes (2.1.3.2).

2.1.3.1 Introdução ao OpenCV

Uma das plataformas mais utilizadas no desenvolvimento de aplicações de processamento de imagem é o OpenCV. Esta foi criada em 2000 e o seu reconhecimento foi aumentando ao longo dos anos, devido ao impacto positivo sentido pela comunidade cientifica. O crescente aumento de popularidade da biblioteca, primeiramente implementada em *C*, originou a sua conversão para *C++*. Atualmente, está disponível em todos os sistemas operativos mais utilizados, tendo acesso a várias linguagens de programação na sua biblioteca (*Java*, *Python*, *MATLAB*) [11].

O OpenCV possibilita a manipulação de imagens em diferentes extensões (*.jpeg*, *.png*, *.webp*, etc) e disponibiliza vários métodos para processamento (operações aritméticas, lógicas e morfológicas) presentes na tabela 2.1.

Tabela 2.1: Exemplos de métodos de processamento de imagem através da plataforma OpenCV

	Função	**Descrição**
Funções Primárias	*imread*	Ler uma imagem para uma variável
	imwrite	Guardar uma imagem num ficheiro
Operações Lógicas	*bitwise_AND*	Aplicação da operação lógica AND a duas imagens
	bitwise_OR	Aplicação da operação lógica OR a duas imagens
Operações Geométricas	*warpAffine*	Translação de uma imagem
	resize	Escalamento de uma imagem
Operações Morfológicas	*dilate*	Operação de dilatação aplicada a uma imagem
	erode	Operação de erosão aplicada a uma imagem

Para além das operações descritas, a plataforma OpenCV fornece outros métodos importantes na caracterização de imagens presentes em [11]. Um desses métodos cria um histograma de uma imagem em tons de cinzento. Este pode ser considerado importante pois, é possível ter acesso ao número de píxeis com um determinado valor de cinza e realizar a equalização da imagem caso se pretendam novos níveis de contraste na mesma. A remoção de ruído é uma das técnicas utilizadas para realçar a informação relevante de uma imagem antes desta ser processada (pré-processamento). Filtros gaussiano, média e mediana são exemplos de métodos que fazem parte da biblioteca OpenCV .

2.1.3.2 Aplicações de Processamento de Imagem através do OpenCV

Atualmente, a plataforma OpenCV é utilizada em diversas áreas da ciência, como por exemplo a medicina. Em [12], foram realizadas experiências através de uma imagem de teste (*Shepp–Logan phantom*) com intuito de verificar a possibilidade de utilização desta biblioteca na área referida. Foi usada a plataforma *OpenCV* em linguagem *Python*, a partir da qual foram testadas as seguintes funções: modificação do nível de brilho, deteção de contornos, binarização, dilatação e erosão. Os resultados obtidos foram positivos, confirmando assim a possível utilização desta biblioteca na área da medicina.

O setor alimentar também recorre a esta biblioteca nomeadamente na análise dimensional e no estado de conservação da fruta através de um algoritmo com base na biblioteca *OpenCV*. Foi utilizado o algoritmo *k-means* que baseia a sua identificação na cor dominante, através de histogramas presentes na biblioteca. De seguida, foram calculadas as dimensões do objeto através da distância euclidiana entre extremos. Os resultados obtidos, confirmam que o sistema possibilita a análise dimensional da fruta e a verificação do seu estado de maturação [13].

No setor automóvel, os veículos autónomos necessitam de percecionar o ambiente que os rodeia para se deslocarem corretamente no trajeto sem colidirem com obstáculos. Em [14], foi implementado um algoritmo baseado em processamento de imagem que deteta os limites das faixas de rodagem.

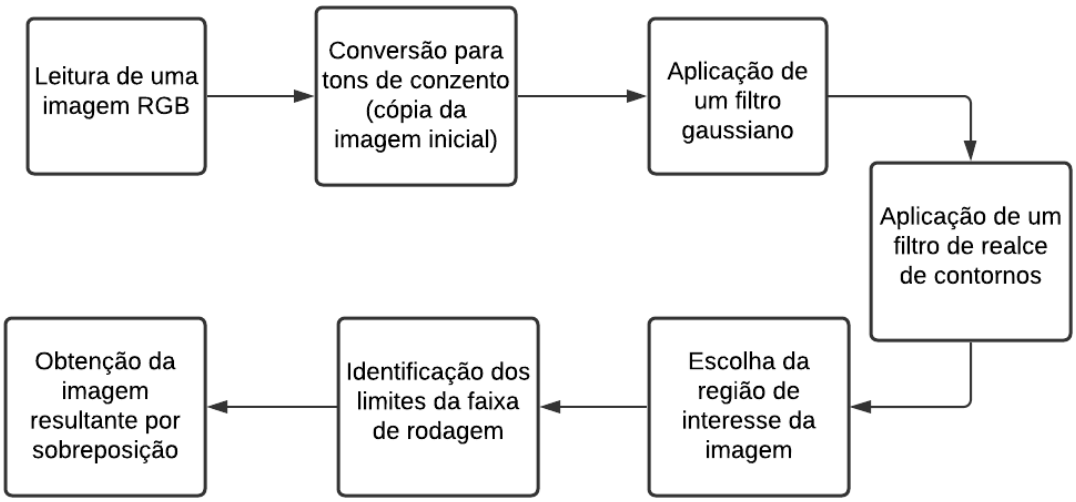


Figura 2.2: Diagrama representativo do algoritmo de deteção dos limites de faixas de rodagem para veículos autónomos

Na figura 2.2, é possível observar o processo realizado na implementação do algoritmo referido em [14]. A partir da função *cv2.imread* é lida uma imagem *RGB*, de seguida, é criada uma imagem cópia e esta é convertida para tons de cinzento (*cv2.cvtColor*). Para diminuir o seu ruído, é aplicado um filtro gaussiano através do método *cv2.GaussianBlur*. A operação seguinte procede ao realce dos contornos (*cv2.Canny*), seguida de uma interseção entre a imagem cópia e uma máscara que marca a região a processar (*cv2.bitwise_and*). Por último, as linhas que delimitam a faixa de rodagem são destacadas a azul (*cv2.HoughLinesP*) e a imagem inicial é sobreposta com a imagem filtrada (*cv2.addWeighted*). De notar que, todas as funções utilizadas estão presentes na biblioteca OpenCV. Os resultados apresentados pelos autores, mostram que esta biblioteca possibilitou o desenvolvimento do sistema de deteção dos limites das faixas de rodagem para veículos autónomos com sucesso.

2.1.4 MATLAB

Nesta secção, irá ser introduzido o processamento de imagem através do *software MATLAB*, as suas características (2.1.4.1) e aplicações em projetos existentes (2.1.4.2).

2.1.4.1 Introdução ao Software MATLAB

A ferramenta de *software MATLAB* tem como foco a análise de dados através da manipulação de matrizes. Uma variável simples é representada por uma matriz com uma coluna e uma linha, enquanto que, um vetor é representado através de várias linhas e várias colunas. Esta ferramenta dispõe da sua própria linguagem de programação, elevada capacidade gráfica e executa comandos listados em ficheiros com extensão *.m*. Devido a estas características, uma das aplicações deste tipo de *software* é o processamento de imagem [15].

Tabela 2.2: Funções de processamento de imagem e exemplos de filtros disponíveis a partir da ferramenta de *software MATLAB*

		Parâmetros	
Função	**Descrição**	**Entrada**	**Saída**
imread	Guardar uma imagem numa variável (matriz)	Diretoria da imagem	Matriz com a imagem guardada
imshow	Visualização de uma imagem	Imagem que irá ser visualizada	-
imwrite	Guardar uma imagem num ficheiro	Matriz que contém a imagem; Diretoria do ficheiro	-
imfinfo	Apresentar informação de uma imagem	Diretoria da imagem	Estrutura referente à imagem composta por nome, dimensão, tipo de formato, etc...
rgb2gray	Conversão de uma imagem RGB para tons de cinzento	Imagem RGB	Imagem em tons de cinzento
imadd	Adição de duas imagens	Duas imagens que irão ser somadas	Matriz com a soma das duas imagens
imabsdiff	Cálculo da diferença absoluta entre duas imagens (pixel a pixel)	Duas imagens que irão ser comparadas (matrizes)	Diferença entre as duas imagens (matriz)
imcomplement	Aplicação do filtro negativo a uma imagem	Imagem que irá ser filtrada	Matriz com a imagem depois da aplicação do filtro

Na tabela 2.2, é possível visualizar algumas das principais funções de processamento de imagem disponíveis na biblioteca do *software MATLAB*, tais como: armazenamento de imagem numa variável (*imread*) ou ficheiro (*imwrite*), aceder à informação da mesma (*imfinfo*) e proceder à sua visualização (*imshow*). Também são apresentadas duas operações

aritméticas e dois filtros possíveis de executar através desta ferramenta. As operações aritméticas expostas são a adição (soma de duas imagens pixel a pixel) e o cálculo da diferença absoluta (cada pixel resultante terá o valor da diferença absoluta dos píxeis correspondentes das imagens iniciais). Apesar de não apresentadas na tabela 2.2, a biblioteca do *software MATLAB* dispõe de diversas operações como a subtração, multiplicação e divisão de duas imagens [15]. São apresentados dois filtros na tabela 2.2: *rgb2gray*, realiza a conversão de uma imagem RGB para tons de cinzento; *imcomplement*, modificação da imagem para o seu complemento (filtro negativo).

Para além dos filtros mencionados, o *software MATLAB* oferece outras funções de auxílio ao processamento de imagem. Em [16], os autores utilizaram as funções *imfilter* (realiza a convolução de uma imagem com uma máscara), *imnoise* (acrescenta ruído a uma imagem) e *medfilt* (aplica um filtro de mediana a uma imagem), evidenciando assim, as diversas funcionalidades desta ferramenta de *software*. Os autores também implementaram um filtro de binarização com um nível de referência fixo e um filtro de deteção de contornos.

2.1.4.2 Aplicações do Software MATLAB em Projetos de Processamento de Imagem

O controlo de qualidade na indústria alimentar é uma técnica utilizada na garantia de conformidade do produto a ser produzido. Em [17], foi desenvolvido um algoritmo, constituído por uma sequência de filtros implementados no *software MATLAB*, que identifica se determinados tipos de fruta possuem defeito ou não. É referido ainda, que os autores obtiveram resultados na ordem dos 80%.

Em [18], foi implementado um sistema idêntico ao exemplo anterior, mas, adaptado ao controlo de qualidade de folhas de plantas. Um algoritmo sequencial foi também desenvolvido, constituído por vários funções: conversão de RGB para tons de cinzento, segmentação, extração de características e classificação. Os resultados obtidos demonstram a viabilidade na detecção de possíveis doenças na fase inicial de crescimento das plantas

Por último, o projeto de um parque de estacionamento foi implementado em [19]. Uma imagem do parque, captada por uma câmara, é sujeita a vários processos de filtragem (conversão para tons de cinzento, seguida de uma binarização e remoção do ruído) de modo a ser possível verificar do número de lugares livres existentes através de um algoritmo de deteção. Os resultados obtidos pelos autores, mostram que é possível detetar o número de lugares vazios num parque de estacionamento através do sistema desenvolvido no *software MATLAB*.

2.2 Processamento de Imagem baseado em FPGA

Nesta secção irá ser abordado o processamento de imagem através de *FPGA*. Primeiramente, é apresentada uma breve introdução da história e principais componentes desta

plataforma (2.2.1), na secção seguinte, é comparado o desempenho de FPGA, GPU e CPU no âmbito do processamento de imagem (2.2.2). De seguida, é apresentado um tipo de sistema que integra duas das plataformas analisadas anteriormente (FPGA e CPU) (2.2.3), e por último, são expostas secções que têm como foco métodos de configuração de FPGA: Hardware Description Language (HDL) (2.2.4), *High Level Synthesys* (2.2.5), e *OpenCL* (2.2.6).

2.2.1 Introdução e História da FPGA

As portas lógicas *AND* (conjunção lógica), *OR* (disjunção lógica) e *NOT* (inversão lógica) são a base dos circuitos digitais. No início da década de 60 foi desenvolvido o primeiro circuito integrado. A sua estrutura era bastante simples possuindo menos de cem transístores. Na década de 70, os circuitos integrados evoluíram de modo a conter na sua constituição milhares de transístores. Na década seguinte, foram desenvolvidos os *programmable logic devices*, que contêm vetores com elevadas quantidades de portas lógicas e podem ser caracterizados por uma função predefinida (*Programmable Logic Array* e *Programmable Array Logic*) ou por serem reprogramáveis (*Generic Logic Array*). O agrupamento destes circuitos num único *chip* deu origem aos *complex programmable logic devices* [20].

Em meados da década de 80, foi desenvolvida uma arquitetura que utiliza *lookup table* baseadas em memória para implementar lógica combinatória em vez de portas lógicas (FPGA) [20] . Na sua constituição podem ser identificados os *Configurable Logic Block (CLB)*, os *Input/Output Block (IOB)* e os conetores.

Os CLB providenciam os elementos fundamentais para construir a lógica definida pelo projetista. Na sua constituição, estão presentes alguns blocos lógicos simples, sendo que, a sua quantidade varia em função do modelo de FPGA. Geralmente, estes blocos são constituídos por uma *lookup table*, um *flip-flop* tipo *D* e um *multiplexer*. Os IOB são grupos de elementos básicos que implementam as funções de entrada e saída em FPGA. Os conetores têm a função de definir ligações entre os CLB e os IOB [21].

Com a evolução tecnológica, as FPGA viram a sua complexidade aumentar. Para além dos CLB, IOB e conetores mencionados anteriormente, são identificados componentes como: *Switch Box (SB)* (localizadas entre CLB), multiplicadores, *Block RAM (BRAM)* e blocos referentes ao *Digital Signal Processor (DSP)*. Os vários blocos de BRAM disponibilizam espaço de armazenamento dentro da FPGA. Os blocos referentes ao DSP têm como função o processamento de dados. Estes podem ser utilizados nas áreas tecnológicas em que é necessário executar enormes quantidades de operações aritméticas, sendo integrados quatro blocos numa única FPGA [22, 23]. Na figura 2.3, encontra-se representada uma arquitetura genérica de FPGA na qual é possível observar alguns dos componentes mencionados nesta secção.

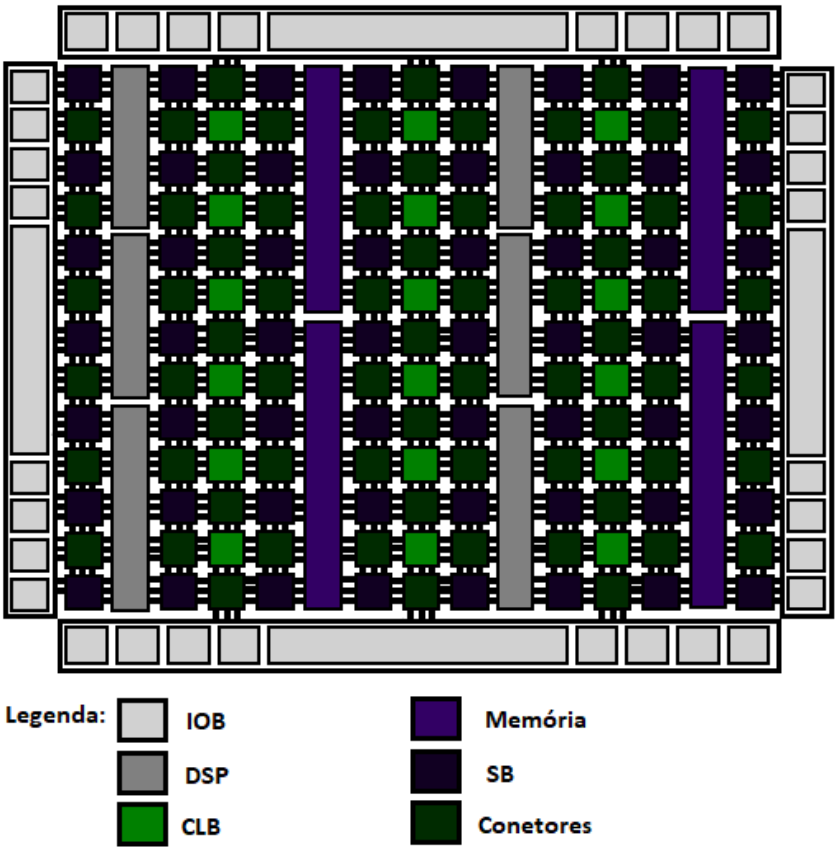


Figura 2.3: Arquitetura genérica de uma FPGA (Adaptado de: [22])

2.2.2 FPGA vs CPU vs GPU no Processamento de Imagem

A constante evolução da tecnologia ao longo das últimas décadas e a necessidade da sociedade atual em possuir novos produtos, implicou uma crescente complexidade nos sistemas de manufatura.

A necessidade de um processamento mais rápido, despoletou a utilização de plataformas com a capacidade de aceleração de *hardware*. Nesta secção, irão ser consideradas a FPGA, o *CPU* e o *GPU*. Diversos estudos realizados mostram a existência de vantagens e desvantagens da sua utilização [24].

Nas duas últimas décadas, o uso de FPGA como solução para acelerar o tratamento de dados dos sistemas de visão tem vindo a aumentar. Em [24], foi efetuado um estudo acerca das soluções existentes a partir de CPU, GPU e FPGA neste período. O elevado grau de portabilidade e paralelismo, assim como, o comportamento flexível ao longo da implementação e uma potencial maior velocidade de processamento de dados são algumas das características das FPGA.

De seguida, serão apresentados resultados específicos da comparação entre as três plataformas no processamento de imagem.

2.2.2.1 Análise do Desempenho da FPGA no Processamento de Imagem

O paralelismo é um dos conceitos de maior importância na comparação do desempenho de processamento de dados em CPU, GPU e FPGA. Este pode ser definido como uma técnica onde diversas tarefas são executadas simultaneamente (em paralelo), comum nas três plataformas referidas, caracterizadas por diversas unidades de processamento.

Um estudo do desempenho entre FPGA e GPU no âmbito do processamento de imagem foi efetuado em [25]. Os autores referem que a plataforma FPGA oferece um elevado nível de paralelismo e portabilidade, o que possibilita a sua aplicação em sistemas de alto rendimento. A sua baixa energia dissipada e a potencial maior flexibilidade em relação aos seus competidores diretos são características cruciais para o seu uso na visão industrial atual. É possível ter um circuito específico para o sistema a desenvolver a partir desta plataforma, devido a ser reconfigurável. Algoritmos como a redução do ruído, correção de imagem e deteção de contornos (*Sobel*), são alguns exemplos de soluções implementadas em FPGA. Em relação aos GPU, a sua produção em massa tem como consequência o menor custo relativamente às FPGA, pode até ser considerado que o rácio preço/velocidade de processamento é, em diversas aplicações, o melhor entre as alternativas em estudo. Como estes são criados com a finalidade de processar imagens e vídeos, a sua programação é menos complexa e mais amigável para o programador. A partir das características mencionadas, as potenciais aplicações destas duas plataformas encontram-se representadas na tabela 2.3.

Tabela 2.3: Características de aplicações adequadas a FPGA e GPU (Adaptado de : [25])

Plataformas	**Características e Potenciais Aplicações**
FPGA	Algoritmos complexos ou simples Elevado processamento de dados Elevada portabilidade Baixas potências
GPU	Algoritmos complexos ou simples Elevado processamento de dados Baixo período de desenvolvimento Baixo custo

O desempenho de FPGA, CPU e GPU foi comparado a partir do algoritmo *Sum of Absolute Differences (SAD)* e da convolução de duas dimensões entre uma imagem e uma máscara [26]. O algoritmo SAD pode ser utilizado para verificar a similaridade entre duas imagens, um exemplo da sua aplicabilidade são os sistemas de vídeo vigilância. A convolução entre uma imagem e um *kernel* pode ser descrita como a aplicação de um filtro à mesma. Neste artigo foram utilizados as seguintes plataformas:

1) FPGA: *GiDEL ProcStar III board* integrada com *65 nm Altera Stratix III E260*;
2) GPU: *55 nm EVGA GeForce GTX 295 PCIe x16*;
3) CPU: *Intel Xeon 4-core W3520*;

Ao analisar os resultados obtidos pelos autores de [26], a FPGA tem um consumo energético bastante inferior aos seus competidores diretos. Contudo, no caso do algoritmo SAD, o *GPU-FFT* (plataforma cuja implementação é independente da dimensão do *kernel*) apresenta consumo equivalente ao da FPGA. Os resultados obtidos permitem concluir que quanto maior a dimensão do filtro, maior será o consumo de energia por parte dos CPU e GPU, excepto o caso referido em cima. A FPGA destaca-se das outras plataformas positivamente, pois apresenta o melhor desempenho na aplicação dos dois algoritmos.

Em [27], foi realizado um estudo de comportamento entre CPU, GPU e FPGA em duas aplicações: um sistema de visão e a aplicação de filtro bidimensional a uma imagem em tons de cinza. Em relação ao sistema de visão, duas câmaras captam imagens da mesma área permitindo obter a distancia entre objetos, tal é conseguido através de projeções e do algoritmo SAD. Neste estudo foram utilizados os seguintes dispositivos: *Xilinx XC4VLX160* (FPGA), *XFX GeForce 280 GTX 1024MB DDR3 standard*, *CUDA version 2.1* (GPU) e *Intel Core 2 Extreme QX6850*, *Intel C++ Compiler 10.0* (CPU).

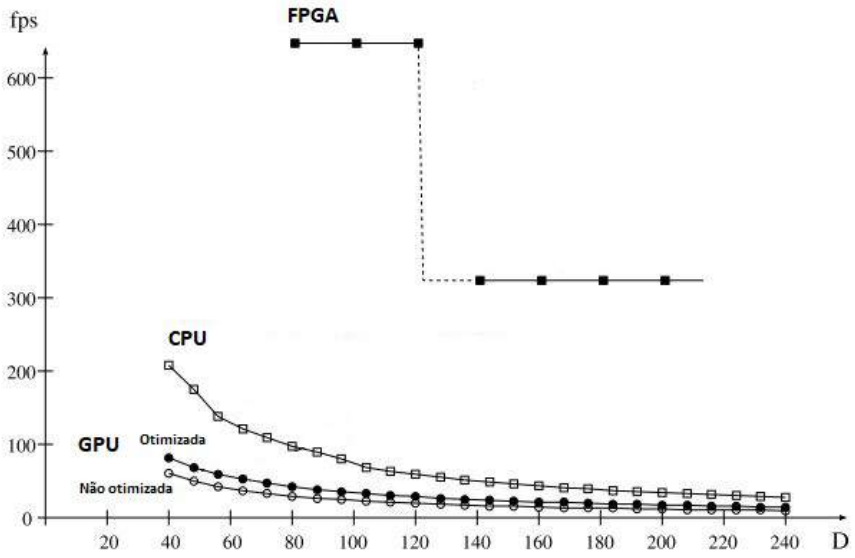


Figura 2.4: Comparação do comportamento de um sistema de visão utilizando diferentes plataformas (Adaptado de: [27])

Por análise da figura 2.4, é possível verificar que o desempenho do sistema baseado em FPGA é bastante mais elevado do que através das outras plataformas. Duas implementações de GPU foram utilizadas nesta comparação, uma solução otimizada e uma não otimizada. Apesar de apresentarem resultados equivalentes, a versão otimizada consegue valores de *Frames Per Second (FPS)* ligeiramente superiores. O CPU tem um desempenho superior comparativamente aos GPU, mas, com o aumento do valor *D* (variável responsável pelo número de filtros que serão comparados com a imagem em questão) os valores obtidos aproximam-se dos GPU.

Em relação ao filtro a duas dimensões, foi considerada a variável *W* que representa o dimensão do filtro a aplicar a uma imagem em tons de cinzento (640x480 píxeis de

resolução). Na figura 2.5, é possível verificar os resultados obtidos [27] da comparação entre as três plataformas quando utilizadas na aplicação de um filtro de duas dimensões. Através do gráfico, conclui-se que o GPU se destacou quando o filtro possui dimensões mais pequenas. À medida que a dimensão deste vai aumentando, os FPS resultantes diminuem. Os valores de FPS apresentados para CPU e FPGA são similares (variando entre 0 e 1000), contudo, (ao contrario da FPGA), o primeiro evidencia um ligeiro decréscimo. De notar que, para filtros de maior dimensão os valores obtidos entre FPGA e GPU são equivalentes.

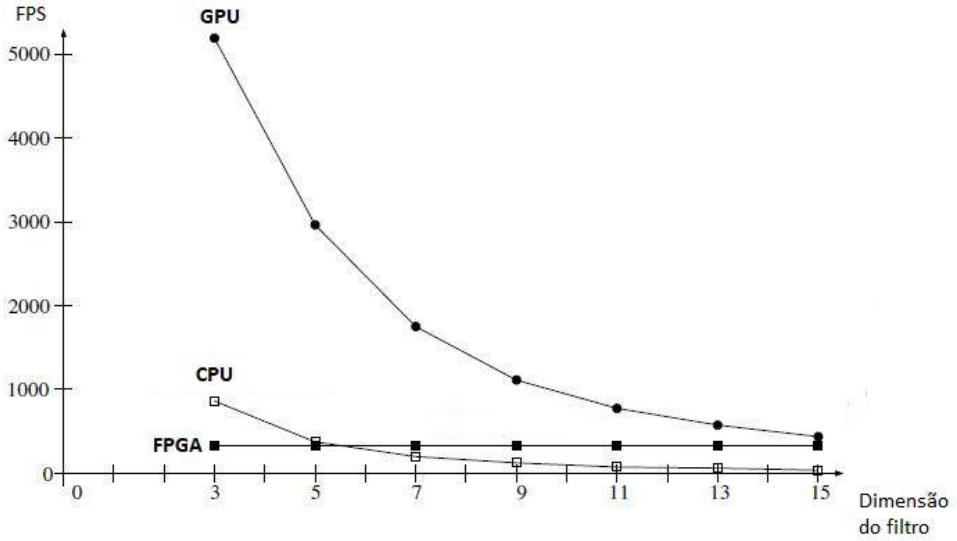


Figura 2.5: Comparação do comportamento da aplicação de um filtro de duas dimensões a uma imagem composta por tons de cinzento utilizando diferentes plataformas (Adaptado de: [27])

A aplicação de filtros em FPGA, GPU e CPU pode ser executada através de *OpenCL* (apresentado na secção 2.2.6). A configuração de *hardware* em algoritmos complexos, tem um elevado grau de dificuldade e não é amigável para o projetista. Desvantagem contornável pela utilização de *OpenCL*. Em [28], foi realizado um estudo sobre o comportamento desta técnica, nas três plataformas já referidas.

Neste artigo foi testado o algoritmo de *fractal compression* que pode ser descrito em quatro fases:

1. Armazenamento de uma imagem ou vídeo num ficheiro e processamento em *frame buffers*
2. Tratamento dos dados de uma *frame* (imagem)
3. Comparação de um código gerado aos já presentes numa base de dados (através do algoritmo SAD)
4. Recriação da imagem codificada através da geração de novos códigos

Tabela 2.4: Comparação do comportamento do algoritmo *fractal compression* nos três tipos de plataformas utilizando OpenCL (Adaptado de: [28])

Tipo	**Plataforma**	**Tempo de Execução do Kernel (ms)**	**Tempo de Transferência (ms)**	**FPS**
CPU	*Intel Xeon W3690*	196.1	0	4.6
GPU	*NVIDIA Fermi C2075*	5.17	3.1	53.1
FPGA	*Altera Stratix IV 530*	2.0	2.2	70.9
FPGA	*Altera Stratix V 5SGXA7*	1.72	1,9	74.4

Depois de implementado o algoritmo em quatro plataformas, foram obtidos os resultados para o tempo de execução do *kernel*, o tempo de transferência entre o *kernel* e *host* e a média de FPS, apresentados na tabela 2.4. A sua análise permite verificar que o algoritmo aplicado às **FPGA** tem um desempenho bastante mais elevado que nas outras duas tipologias de sistemas, pois apresenta os tempos de execução mais baixos e a média de FPS mais elevada. O **CPU** apresenta os piores resultados comparado com as outras duas plataformas. O valor nulo do tempo de transferência entre o *kernel* e o *host* era esperado, visto que, a entidade que executa o *OpenCL* é a mesma que contém o *host*. O tempo de execução do *kernel* e a média de FPS são bastantes negativos, pois os valores estão distantes das outras plataformas em estudo. O algoritmo implementado através de **GPU**, foi executado através de sessenta e quatro *threads*. O tempo de execução mais baixo e a média de FPS mais elevada permitem aferir a superioridade relativamente ao algoritmo implementado por CPU, contudo, não é capaz de rivalizar com os resultados obtidos em FPGA. Este artigo permitiu assim verificar que a implementação através de *OpenCL* nas três plataformas tem desempenhos diferentes, tendo a FPGA, o melhor desempenho.

Um estudo do comportamento entre a velocidade de execução de um CPU e FPGA foi realizado em [29], com o intuito de serem obtidos resultados relativamente à influência do paralelismo no processamento de dados. O algoritmo de teste, destinado ao reconhecimento de íris, foi implementado sequencialmente em *C++* num CPU e em FPGA a partir de VHSIC Hardware Description Language (VHDL). Os resultados obtidos através da plataforma *Cyclone-II EP2C35*, foram utilizados para estimar o rendimento do algoritmo caso este fosse implementado noutra FPGA (*Stratix IV*).

Tabela 2.5: Comparação do desempenho do algoritmo de reconhecimento de íris implementado em FPGA e CPU (Adaptado de: [29])

	***Intel Xeon X5355* (CPU)**	***Cyclone-II* EP2C35 (FPGA)**	***Stratix IV* (FPGA)**
Tempo de reconhecimento (ns)	383	20	2
Velocidade em relação ao CPU	-	19.15	191.5
Memória usada (%)	-	73	7.3

Na tabela 2.5 estão representados os resultados referentes ao desempenho do algoritmo de reconhecimento de íris. Os resultados da FPGA são bastante melhores comparativamente aos obtidos a partir da utilização de CPU. A plataforma *Cyclone-II EP2C35*, apresenta uma velocidade aproximadamente 19 vezes superior à do CPU. Caso se considere a FPGA *Stratix IV*, foi estimado que esta teria um rendimento aproximado de 10 vezes superior em relação à solução anterior, e desta forma, o valor da sua velocidade em relação ao processador *Xeon* seria de aproximadamente 191 vezes. A partir dos resultados obtidos, facilmente se conclui que o processamento em paralelo utilizado pela FPGA atingiu resultados bastante melhores comparativamente ao processamento sequencial do CPU.

2.2.2.2 Considerações

Com o aumento da complexidade dos sistemas industriais, aumenta o volume de dados e a velocidade a que tem de ser processados, o que levou à introdução das plataformas FPGA e GPU na sua composição. Alguns dos fatores favoráveis à sua escolha são: velocidade de processamento de dados, tempo de desenvolvimento do projeto, potência dissipada, energia consumida e preço.

Na secção 2.2.2.1 foram analisados estudos que comparam FPGA, GPU e CPU. A FPGA apresenta um nível elevado de paralelismo, portabilidade e um baixo consumo energético. Apesar das vantagens, o projeto de sistemas que utilizam esta plataforma é complexo e o seu preço é elevado. Considerando os GPU, estes fornecem uma programação mais amigável, que por consequente, pode diminuir o tempo de desenvolvimento do projeto. O seu preço é potencialmente mais baixo mas, os valores de consumo de energia e potência dissipada são frequentemente superiores em comparação com a FPGA. Em relação aos CPU, são caracterizados pelo paralelismo apenas quando possuem na sua composição vários *cores*.

Analisando alguns resultados dos artigos apresentados nesta secção, é possível encontrar resultados distintos. Em [28] é referido que a FPGA têm um desempenho superior ao GPU, no entanto, em [27] é referido o contrário. Pode ser então considerado que, a velocidade de processamento e desempenho destas plataformas, dependem da aplicação ou sistema onde serão introduzidas.

2.2.3 *System on Chip*

Na subsecção 2.2.2, foi apresentado o desempenho de FPGA, GPU e CPU quando aplicados ao processamento de imagem. No seguimento dessa análise, nesta secção irá ser abordado um tipo de sistema que integra FPGA e CPU, denominado *System on Chip (SoC)*.

Com a evolução tecnológica e o aumento do número de portas lógicas nos circuitos digitais, foi criado um tipo de sistema que integra no mesmo *chip* vários blocos (*hardware* e *software*). Segundo os autores de [30], esta solução apresenta vantagens como: uso

eficiente de recursos, redução do custo do produto e aumento da velocidade de processamento. Um exemplo deste tipo de sistema é a plataforma *Zynq-7000* [31], que combina um processador *ARM* com uma FPGA. Através desta integração, é possível dispor de aceleração de *hardware* e ter acesso a um processador num único *chip*, concedendo ao projetista a hipótese de escolher em qual dos blocos (*software* ou *hardware*) irá ser executado o código. Nos parágrafos seguintes, serão apresentados alguns sistemas baseados em SoC, utilizados em aplicações de processamento de imagem.

Em [32], foi realizado um estudo sobre o reconhecimento de semáforos de trânsito em tempo real, na realização deste projeto, foi utilizado um sistema baseado em SoC que integra um processador *ARM* com *Xilinx Zynq ZC-702* (FPGA). A detecção dos mesmos é possível através de um algoritmo de duas fases (deteção e classificação), onde todos os seus blocos foram configurados através da parte de *hardware* do *chip* (FPGA). O processador é utilizado para aceder à memória DDR que contém a posição do semáforo. Foi obtida uma precisão acima dos 90% e alcançado um processamento com cerca de 60 FPS.

Em [33], foi implementado um algoritmo identificador de linhas em estradas. Para esse efeito, foi utilizada a plataforma *Xilinx Zynq FPGA* (SoC). O *hardware* (FPGA) começa por aplicar uma sequência de filtros à imagem (extração da região de interesse, deteção de contornos, binarização e transformada de *Hough*), de seguida, o processador (*software*) tem a função de executar a identificação de linhas retas. Os resultados obtidos, apresentados na tabela 2.6, evidenciam que o sistema desenvolvido demonstra um bom desempenho quando comparado a outros algoritmos de deteção de linhas [34, 35], alcançando valores de FPS bastante mais elevados relativamente às duas outras soluções apresentadas.

Tabela 2.6: Comparação do valor de FPS obtidos em [33] com soluções existentes [34, 35] (Adaptado de: [33])

Sistemas	**Plataforma Utilizada**	**FPS**
[33]	*Xilinx Zynq ZC706*	69.4
[34]	ARM+FPGA	25
[35]	*NVIDIA 7600* GPU	10

A vídeo vigilância é outra área onde os SoC representam uma mais valia. Em [36], foi implementado um sistema de deteção de contornos através da plataforma *Zynq* da empresa *Xilinx*. O *chip* em questão combina um CPU e uma FPGA. Esta é responsável pela modificação da imagem através do filtro *Sobel*, assim como, pelo armazenamento dos dados resultantes em memória (*DDR* e *SD*). Relativamente ao CPU, tem como função lidar com as transferências de informação entre a parte de *hardware* e as memórias. Relativamente aos resultados obtidos, na tabela 2.7 é apresentada a quantidade de recursos utilizados deste sistema em comparação com uma solução existente [37]. Pela análise da mesma, é possível concluir que o sistema em questão é mais eficiente que o apresentado em [37] pois, a quantidade de *hardware* utilizado é menor.

Tabela 2.7: Comparação dos recursos utilizados em [36] relativamente a [37] (Adaptado de: [36])

Componentes da FPGA	Total Disponível	Componentes Utilizados (Unidades) [36]	Componentes Utilizados (%) [36]	Componentes Utilizados (%) [37]
lookup table	53200	5677	10.6	41.48
flip-flop	106400	7919	7.4	34.97
BRAM	140	7	5	20
IOB	200	52	26	32

2.2.4 HDL

Nesta secção, vão ser apresentadas duas linguagens utilizadas na descrição de circuitos digitais, *VHDL* (2.2.4.1) e *Verilog* (2.2.4.2), assim como, o seu contributo para o processamento de imagem através de FPGA.

2.2.4.1 VHDL

As *HDL* estão orientadas para descrever estruturas de *hardware* e comportamentos. Uma das grandes diferenças entre estas e as outras linguagens de programação é o facto de representarem operações em paralelo em vez de operações sequenciais [38].

Uma das linguagens de descrição de *hardware* é o VHDL. É utilizada na descrição, concepção e sintetização de circuitos digitais. Esta possui bibliotecas e permite a implementação de sistemas hierárquicos, ou seja, um bloco principal mais complexo é composto por vários blocos secundários mais simples. Também permite a simulação de circuitos, útil antes da síntese dos mesmos, pois permite verificar o seu correto funcionamento e proceder a ajustes de modo a obter os resultados desejados [38]. Algumas das suas características são a capacidade de descrever vários circuitos digitais (flexibilidade), a possibilidade de síntese de sistemas para uso posterior em FPGA, assim como, baixo custo [38].

A linguagem de descrição aqui apresentada é bastante diferente de outras como *C* ou *Java* pois, os sistemas desenvolvidos, têm de respeitar um comportamento específico e não uma sequência de operações. Para este efeito são usados sinais, componentes e comportamentos. Em [38], os autores realizam uma introdução a esta linguagem e explicam como são configurados circuitos digitais através de VHDL.

O processamento de imagem em FPGA pode ser efetuado por implementação de filtros em VHDL. Em [39], os autores realizaram um estudo sobre um filtro de deteção de contornos (*Sobel*). O algoritmo em questão consiste na combinação de dois métodos (convolução e deteção de contornos), ambos desenvolvidos em VHDL através de uma *Spartan 3E* (FPGA). Os resultados obtidos mostram que o algoritmo foi testado e verificado com sucesso. Outros filtros podem ser implementados recorrendo a esta linguagem de programação. Em [40], os autores realizaram um estudo no âmbito do processamento de imagem

através de FPGA (*Xilinx FPGA Spartan-6*), sendo que, foram implementados filtros de erosão, dilatação, remoção de ruído e deteção de gradiente. Neste sistema a informação a processar é recebida por um computador, através de uma porta *UART* (*Universal Asynchronous Receiver-Transmitter*). Os autores referem que o uso de VHDL aumenta o tempo de implementação do sistema mas, proporciona um maior controlo no desenvolvimento do circuito, assim como, maior flexibilidade nas alterações no mesmo.

2.2.4.2 Verilog

Outro exemplo de HDL é o *Verilog*. Tal como o VHDL, esta linguagem descreve circuitos digitais e possibilita a sua síntese, simulação e utilização em FPGA. Em [41], são apresentados vários exemplos de métodos como: lógica combinatória, operações aritméticas (adição, subtração e multiplicação), lógica sequencial e máquinas de estados síncronas.

Um das suas áreas de aplicação é o processamento de imagem. Em [42], foram desenvolvidos quatro filtros (modificação de contraste, binarização, negativo e modificação de brilho) através da plataforma *Altera Cyclone IV*. Os resultados apresentados pelos autores demonstram que o desenvolvimento destes algoritmos é possível através de *Verilog*. Outro exemplo é a implementação de um filtro de deteção de contornos na plataforma *Xilinx Virtex-5* [43]. Os resultados obtidos evidenciam que o algoritmo desenvolvido é mais eficaz na deteção de contornos que o filtro *Canny* tradicional.

2.2.5 *High Level Synthesis*

Nesta secção será introduzido um método de configuração de FPGA, o *High-Level Synthesis (HLS)*. Em 2.2.5.1 é apresentada uma breve introdução a esta técnica, em 2.2.5.2 são expostas algumas ferramentas disponíveis para este tipo de síntese, e por último, é apresentado o seu uso no processamento de imagem (2.2.5.3).

2.2.5.1 Introdução ao HLS

Quando surgiram as FPGA, a sua configuração tinha como foco a implementação de circuitos digitais através de HDL (VHDL e *Verilog*). Com a evolução dos circuitos digitais, a sua complexidade e o número de portas lógicas aumentaram significativamente. A consequência deste aumento de sofisticação deu origem ao HLS durante a década de 90. Este método gera uma arquitetura *Register Transfer Level (RTL)* a partir da sua implementação numa linguagem de nível mais alto (*C/C++/MATLAB*) [44].

O HLS tem duas fases sequenciais: síntese do algoritmo e síntese de interface. A primeira fase, tem como objetivo extrair o comportamento de um sistema descrito numa linguagem de alto nível, e de seguida, configurá-lo para um número de ciclos de *clock*. Esta fase pode ser explicada através de um exemplo prático, um circuito digital que realiza quatro multiplicações pode ser implementado de três formas: um somador e um multiplicador (quatro ciclos de *clock*); dois somadores e dois multiplicadores (dois ciclos

de *clock*); três somadores e quatro multiplicadores (um ciclo de *clock*). As três implementações são possíveis, contudo, fatores como o número de ciclos de *clock* e a quantidade de elementos em questão têm influência na sua escolha. Na segunda fase, é feita uma análise dos parâmetros da arquitetura principal, a partir da qual, são definidos os pinos de entrada e saída do sistema [45].

2.2.5.2 Ferramentas de HLS

Com o intuito de gerar um circuito digital através de HLS podem ser usadas várias ferramentas. Em [46], são apresentadas e analisadas ferramentas que geram arquiteturas RTL através de circuitos descritos em linguagens de nível mais elevado, assim como, as áreas de interesse do HLS na comunidade cientifica. Uma dessas ferramentas referida pelos autores é o *Vivado HLS*.

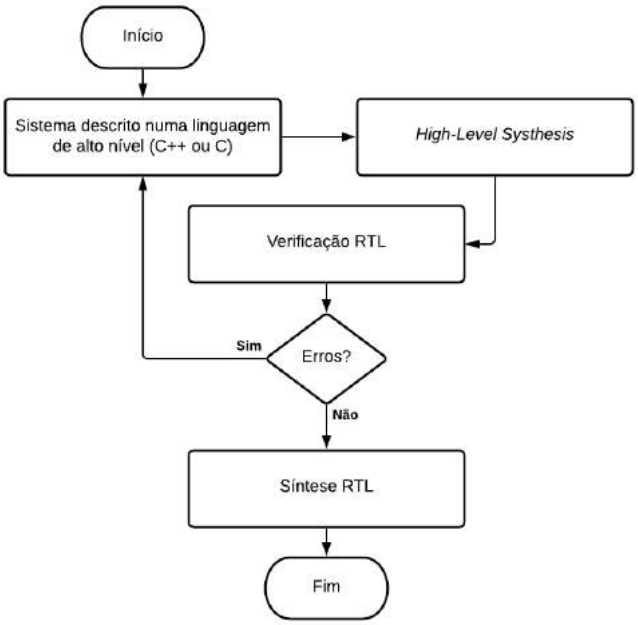


Figura 2.6: Diagrama de funcionamento do *High-Level Syntesis* através do *Vivado HLS* (Adaptado de: [45])

Na figura 2.6, está representado o diagrama de funcionamento do HLS através do *Vivado HLS*. Inicialmente, um circuito digital é descrito numa linguagem de alto nível (*C/C++*) tendo em conta especificações prévias, sendo de seguida, gerado um ficheiro de teste para ser garantida a funcionalidade do mesmo. Após a síntese, é gerado um modelo RTL que é verificado através de várias simulações. Caso não tenha o comportamento desejado, será necessário uma modificação do ficheiro *C/C++* por parte dos projetistas. Caso contrário, é gerado o modelo com a arquitetura RTL desejada [47].

O HDL *coder* é uma ferramenta que realiza a síntese de código em linguagem *MATLAB* para *Verilog* ou VHDL. Esta ferramenta presente no *software MATLAB*, permite a

configuração de FPGA [48]. Outro exemplo de ferramentas de HLS é a *Intel HLS Compiler* que realiza a conversão de código em *C++* para arquiteturas RTL, podendo ser utilizada para a configuração de FPGA da empresa *Intel* [49].

2.2.5.3 HLS Aplicado ao Processamento de Imagem

São vários os exemplos da aplicação do HLS no processamento de imagem através de FPGA. Em [45], foi implementado um algoritmo de deteção de faces utilizando a síntese de código de linguagem *C++*. Os resultados obtidos mostram que o algoritmo implementado através desta técnica possibilita a deteção de faces com sucesso. Outro exemplo, é a implementação de dois filtros: *Sobel* (deteção de contornos) e deteção de linhas [47]. Neste artigo, é referido que o uso desta técnica possibilita uma implementação mais rápida do sistema mas, tem como limitação a elevada quantidade de *hardware* utilizado pela FPGA.

Em [50], os autores apresentam diversas vantagens e desvantagens do HLS no âmbito do processamento de imagem. Uma das vantagens referidas, é o facto de muitos dos compiladores atuais terem a capacidade de converter código de linguagem *C* para HDL, facilitando o uso desta técnica.

Um circuito digital é composto por uma parte de dados e uma parte de controlo. Ao ser efetuada a síntese de uma linguagem de alto nível, a parte de controlo do sistema é automaticamente gerada, devido a estar implícita no código. Este facto pode ser considerado uma vantagem do uso de HLS pois, caso o sistema fosse implementado através de HDL seria necessário projetar as duas partes do sistema (controlo e dados). Esta técnica de configuração também apresenta algumas limitações que podem influenciar negativamente a correta síntese do circuito, tais como: uso de apontadores (endereço de uma variável em memória) e recursividade do código (operações cíclicas). Outra desvantagem a destacar é a maior dificuldade na descrição de circuitos digitais síncronos (dependentes da frequência de *clock*) pois, a noção de tempo está implícita quando o sistema é implementado através de uma sequência de instruções [50].

2.2.6 OpenCL

Open Computing Language (OpenCL) é uma estrutura de programação de sistemas do tipo dispositivo/*host*. Uma aplicação baseada em OpenCL pode ser separada em duas partes distintas: o programa principal (*host*) escrito em qualquer linguagem de programação, desde que seja compatível com o compilador a ser utilizado, e um dispositivo programado em linguagem *OpenCL C*. O funcionamento de uma aplicação baseada em OpenCL pode ser descrito da seguinte forma: inicialmente todos os dados estão alocados na memória do *host*, de seguida, esta é transferida para a memória do dispositivo através das respetivas funções da estrutura OpenCL. O código é executado no dispositivo, sendo que, os sinais de entrada são lidos e os sinais da saída escritos na memória do mesmo. Por fim, os dados resultantes são transferidos da memória do dispositivo para o *host* [51].

O OpenCL pode ser aplicado em FPGA. Em [52], foi feita uma análise sobre a utilização desta estrutura para a implementação de circuitos digitais em plataformas reconfiguráveis. Neste artigo, é referido o compilador OpenCL da empresa *Altera* que traduz o código do dispositivo (*OpenCL C*) para *hardware* (HDL). Os autores, referem que o uso de OpenCL pode ser visto como uma vantagem quando aplicado a sistemas baseados neste tipo de plataformas pois, pode oferecer melhor desempenho e menor consumo energético quando aplicado em FPGA. Também é referido que, a utilização do compilador da empresa *Altera* apresenta melhor desempenho quando aplicado em modelos de FPGA mais recentes.

Em [53], foi realizada uma comparação do desempenho entre a configuração de FPGA através de HDL e OpenCL, aplicada ao processamento de imagem. Foram implementados três algoritmos: *Sobel*, deteção de contornos (*canny*) e extração de características. Cada um destes foi implementado em VHDL e OpenCL, de modo a ser possível uma comparação entre as duas formas de configuração. Foram utilizadas três FPGA do tipo *Altera Stratix-V (28nm)*, sendo que, os filtros baseados em OpenCL foram implementados nas três plataformas e os algoritmos em VHDL apenas numa. Em relação aos resultados obtidos pelos autores, a implementação foi cerca de seis vezes mais rápida quando utilizada a estrutura OpenCL porém, a descrição através de VHDL providenciou um menor número de recursos utilizados para as mesmas frequências de operação.

2.3 Sistemas de Visão Industrial

Nesta secção são introduzidos os sistemas de visão industrial. Em 2.3.1 são expostos os seus princípios de funcionamento e as câmaras utilizadas na sua constituição. De seguida, são apresentados exemplos da visão industrial aplicada ao controlo de qualidade (em 2.3.2) e à deteção de objetos (em 2.3.3). Por fim, é feita uma análise do impacto e contributo das FPGA quando inseridas na sua composição (em 2.3.4).

2.3.1 Introdução à Visão Industrial

Esta subsecção é de carácter introdutório aos sistemas de visão industrial. Em 2.3.1.1 são apresentados os seus princípios de funcionamento e em 2.3.1.2 são analisadas várias câmaras que podem fazer parte da sua composição.

2.3.1.1 Princípios de Funcionamento

A garantia de produtos com elevada qualidade é essencial na atualidade. Devido a este facto, a necessidade do uso de sistemas de visão foi crescendo com o aumento da industrialização.

O seu funcionamento processa-se em várias fases da seguinte forma: em primeiro lugar, uma imagem do produto que irá ser analisado é captada através de uma câmara, que de seguida, é processada via CPU/FPGA/GPU. O processamento da imagem captada

tem como finalidade a extração de informação da mesma, que irá ser avaliada posteriormente. Por último, é tomada uma decisão dependendo do resultado da avaliação prévia [4]. Na figura 2.7 está representado graficamente um diagrama descritivo do processo de funcionamento genérico de um sistema de visão industrial.

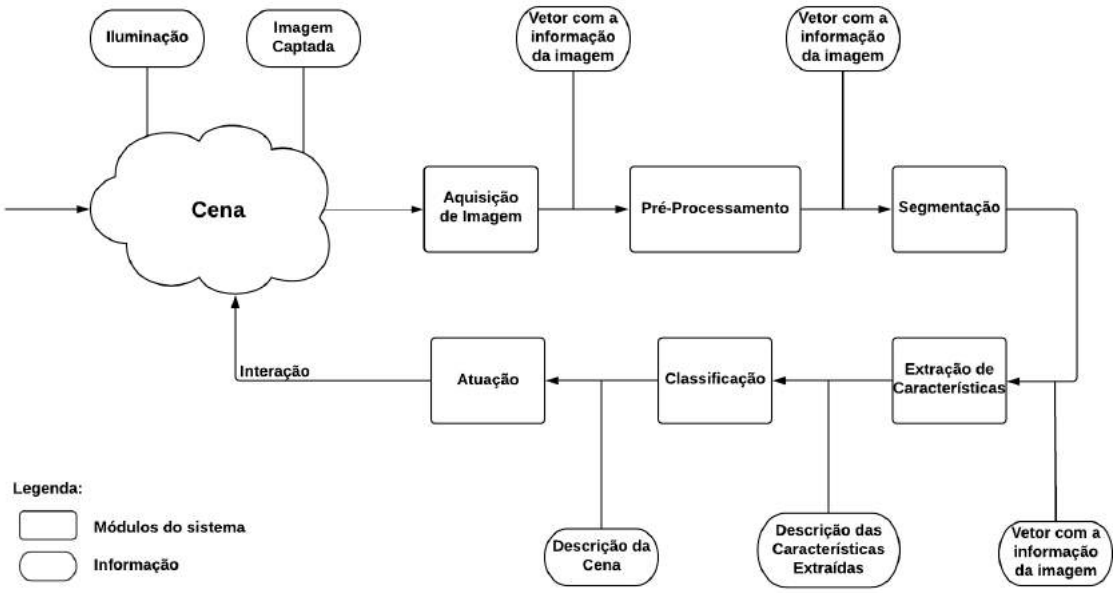


Figura 2.7: Diagrama generalizado do funcionamento de um sistema de visão industrial (Adaptado de: [54])

Com o intuito do funcionamento deste tipo de sistemas ser mais eficiente existem fatores importantes a ter em conta. Tem de ser adaptado ao seu ambiente envolvente pois, terá um rendimento mais elevado em condições benéficas. Fatores como a iluminação do meio onde está inserido, variação do ângulo do produto na linha de montagem e ruído reduzido, são bastante significativos para um melhor funcionamento. Duas das características que definem um sistema de visão industrial são a sua flexibilidade e tempo de execução [55]. Em relação à flexibilidade, esta capacidade pode ser descrita como a capacidade de realizar vários tipos de tarefas. O período do processo de visão é inversamente proporcional ao maior rendimento do sistema. Atualmente, uma das principais metas na implementação deste tipo de sistemas é a otimização destas duas características.

2.3.1.2 Câmaras

A imagem captada via câmara é um dos elementos principais no funcionamento dos sistemas de visão industrial. A escolha deste tipo de sensor pode influenciar a facilidade de extração da informação pretendida pois, cada dispositivo de captação de imagem tem características próprias que devem ser consideradas dependendo da finalidade do projeto. De seguida, vão ser analisadas algumas câmaras industriais para ser percetível o papel das mesmas neste tipo de sistemas.

As câmaras *2D* e *3D* podem ser utilizadas na visão industrial. Em [56], foi realizado um projeto baseado na deteção de objetos através destes dois tipos de sensores. A câmara *3D* foi utilizada na deteção do objeto, devido à sua capacidade de adicionar o efeito de profundidade à imagem captada. A câmara *2D* (montada num braço robô permitindo a sua movimentação) tem a função de obter a orientação e posição do mesmo depois de identificado. Foram realizadas três experiências: precisão da deteção *3D*, precisão da posição/orientação *2D* e precisão do sistema no seu todo. Os resultados não foram positivos relativamente ao primeiro teste porém, em relação às outras duas experiências, estes mostram que a deteção baseada em duas câmaras (*2D* e *3D*) é viável. Existem vários modelos de câmaras *3D* que podem ser utilizados, alguns estão presentes em [57], assim como, as suas características. Neste artigo, são referidos alguns exemplos do uso destas câmaras como: deteção de distâncias entre produtos, aplicação em sistemas de orientação para invisuais, deteção de objetos para evitar colisões (setor automóvel) e medição de volumes de produtos.

Outro exemplo de um dispositivo utilizado na indústria são as câmaras térmicas. Podem ser definidas como um sensor que captura a radiação infravermelha emitida por todos os objetos com temperatura acima de zero graus *celsius*. Inicialmente, este tipo de câmara foi desenvolvido para aplicações de vídeo vigilância, contudo, também pode ser utilizado em sistemas de visão, devido a eliminar os problemas de iluminação característicos das câmaras com imagens resultantes em RGB ou tons de cinzento. Na indústria, podem ser utilizadas para aplicações de diagnóstico de certas propriedades de materiais através da sua temperatura (controlo de qualidade) [58].

Outro tipo de câmaras que têm uma relação direta com os sistemas de visão industrial são as *smart cameras*. Este tipo de plataforma, tem como funções providenciar um elevado nível de informação sobre a imagem em questão, assim como, gerar dados específicos para serem utilizados em sistemas autónomos e inteligentes. Esta câmara não tem como finalidade principal fornecer uma imagem de elevada qualidade para a visualização humana, mas sim, conseguir descrever os acontecimentos presentes na mesma para proporcionar uma melhor tomada de decisão por parte do sistema. Em relação às outras câmaras destacam-se duas diferenças: integração de um ou mais microprocessadores de elevado desempenho na constituição do bloco de processamento de imagem; descrição da cena que está a ser analisada ou características da mesma. A extração de características de uma imagem e o reconhecimento de padrões são dos algoritmos mais utilizados neste tipo de câmaras. Vídeo vigilância, transporte inteligente e setor automóvel são exemplos de áreas de aplicação deste tipo plataforma [59].

As características dos sistema de visão são bastante favoráveis às *smart cameras*, oferecendo vantagens como: condições consistentes de luminosidade quando utilizadas no interior de fábricas; ausência de muitos problemas de oclusão (por exemplo, um objeto encontrar-se atrás de outro o que impede a sua deteção); padrões de reconhecimento limitados, facilitando a tarefa de identificação de produtos [59].

Em [60], foi realizado um estudo sobre o desempenho de três arquiteturas de *smart*

cameras, quando aplicadas a um sistema em tempo real que calcula a distância e ângulo de uma câmara, a partir de pontos de referência. Os três parâmetros utilizados para calcular o desempenho de cada uma foram: latência, velocidade de *frames* e potência consumida. A primeira arquitetura consiste numa FPGA composta por blocos de visão (modelados em RTL) e um processador. A segunda é uma câmara existente no mercado (*Matrox Iris GT*). Por fim, a terceira é composta por uma FPGA (blocos de visão) e um microprocessador externo. De notar que, foi utilizado o modelo *Spartan 6* da empresa *Xilinx* (FPGA). Os resultados obtidos mostram que a câmara existente no mercado apresenta consumo de potência e latência maiores, assim como, velocidade de *frames* menor comparado com outras arquiteturas. Contudo, esta é uma solução mais atrativa, beneficiando de uma programação de nível mais elevado e de uma biblioteca de processamento de imagem. Tendo em conta as outras plataformas, a menor latência e potência consumida foram obtidas quando utilizada a terceira arquitetura.

2.3.2 Controlo de Qualidade

Através do controlo de qualidade nos sistemas industriais, desde a primeira até à última fase do processo de manufatura, é possível obter um melhor resultado. Análises recorrentes ao longo do processo de manufatura são efetuadas aos requisitos principais do objeto para ser garantida a sua máxima qualidade, tais como: deteção de defeitos, supervisão da superfície e avaliação de dimensões. Existem dois tipos de processos de controlo, um tem como finalidade manter o sistema funcional e estável, enquanto que o outro, tem como único foco o produto [61].

A inspeção visual automática é uma das técnicas utilizadas na indústria com a finalidade de garantir a maior qualidade do produto [62]. O seu foco está diretamente relacionado com o resultado, não tendo um papel preponderante na flexibilidade da linha de montagem, mas tendo sim, bastante importância na eficiência da mesma. Esta técnica é caracterizada pelo uso de vários componentes (câmara, iluminação e um componente que realiza o processamento de imagem). Todos estes formam um sistema com diversas funcionalidades de análise de características de um produto, tais como: verificação das dimensões, integralidade e deteção de defeitos [54].

Devido a dispor de características como uma maior velocidade de processamento, elevada precisão de resultados e baixo custo, a inspeção automática eleva a produção de alta qualidade nas indústrias em que é aplicada. Na indústria automóvel, através do uso de um sistema baseado em inspeção, composto por cinco câmaras industriais e quatro fontes de iluminação, é possível a deteção de defeitos na superfície dos veículos em produção. Cinco ângulos diferentes são analisados, sendo que, o processo de aquisição de dados é feito através do uso de uma plataforma móvel, auxiliada por um sistema de iluminação específico para as condições do ambiente envolvente. Quando concluída a fase de captação dos diferentes ângulos, através de processamento de imagem realizado por um CPU, são detetados os defeitos existentes [63].

A deteção de deficiências é uma das áreas na qual a inspeção automática industrial tem maior destaque. Em [64], através do uso de uma câmara para a aquisição de uma imagem, na qual, é visível um conetor de um motor elétrico, são analisadas possíveis anomalias no mesmo através de três processos: identificação do número de circunferências presentes, medição dos parâmetros das mesmas e determinação do diâmetro dos cabos do conetor. As três análises enunciadas anteriormente são realizadas através de processamento de imagem, sendo de seguida, feita uma tomada de decisão acerca da existência ou não de defeitos.

A iluminação tem um impacto elevado na eficiência dos sistemas de inspeção automática. Em [65], de forma a detetar falhas em objetos transparentes que não possuem superfícies planas, foi necessário adaptar a iluminação para serem obtidos os resultados desejados. A transparência dos objetos tem um impacto significativo na identificação de falhas pois, quando estes são expostos a fluxos de luz refletem-nos na sua superfície, mas também, absorvem uma quantidade do feixe incidente (fenómeno de refração). Devido a este facto, foi implementado um algoritmo auxiliado por um sistema de iluminação binário, para ser possível a deteção de anomalias neste tipo de objetos.

Em [66], através da análise de peças industriais com formato de anel, foi possível a identificação de possíveis falhas. O processamento de três imagens do produto em diversas posições, recorrendo a um CPU, possibilitou a deteção de defeitos (presença de arranhões ou deformações na superfície da peça industrial), através de um sistema de inspeção automática.

2.3.3 Deteção e Classificação de Objetos

Vários tipos de indústrias utilizam técnicas de deteção de objetos. Este tipo de sistemas permitem diferenciar, classificar, ou até caracterizar os resultados quanto às suas propriedades [54]. A identificação e posterior classificação de produtos do mesmo tipo, em fases intermédias do processo de manufatura, providencia um maior rendimento ao sistema através da diminuição do tempo de produção.

Na indústria alimentar, existem vários exemplos do uso de sistemas de visão para a deteção de produtos. Na verificação da fertilização de ovos [67], ao utilizar um tipo de iluminação apropriada, é possível captar uma imagem (via câmara), com informação referente à qualidade do ovo analisado. Em [68], de modo a aumentar a eficiência na produção, é detetado um objeto (rebuçado) a partir da sua forma e cor, assim como, possíveis anomalias.

Outro exemplo deste tipo de sistemas é a solução apresentada em [69]. A implementação de uma garra equipada com uma câmara e um CPU, com a capacidade de reconhecer peças com diferentes formas e pesos, é um exemplo de um sistema que escolhe e recolhe objetos. A contagem de peças nas linhas de montagem [70] ou de pequenos metais como pregos e *clips* [71], são duas das soluções implementadas com a finalidade de analisar o número de objetos presentes durante a manufatura.

A capacidade de *pick and place* de um robô, pode ser realizada com a intervenção do processamento de imagem. Em [72], através do uso de uma câmara, foi possível calcular as coordenadas (x e y) de uma peça industrial, de seguida, estas são enviadas para um robô que irá recolher o produto consoante essa mesma informação. Outra aplicação tendo em conta o *pick and place*, é o sistema que através de dois braços robóticos, deteta e recolhe objetos aleatórios que estão rodeados por peças do mesmo tipo, localizados num contentor industrial [73]. Tendo em conta o cenário da solução anterior, mas apenas com um braço robótico, foi implementado um sistema robusto, que consegue detetar e recolher qualquer objeto de superfície plana rodeado de peças do mesmo tipo, através de localização *3D* [74].

2.3.4 Uso e Contributo de FPGA

Com os recorrentes avanços tecnológicos vividos ao longo do tempo, as FPGA, caracterizadas por altas velocidades de processamento e flexibilidade, boa relação custo/performance e disponibilidade de vários recursos na sua constituição, tornaram-se numa das soluções utilizadas na indústria.

A complexidade de projetar o sistema necessário e o desenvolvimento reduzido deste tipo de plataformas, foram dois dos fatores que condicionaram a adesão desta tecnologia na indústria. Atualmente, com a integração de processadores e FPGA no mesmo sistema, este tipo de tecnologia é visto não só como acelerador de *hardware*, mas também, como uma plataforma de alto rendimento (SoC) com aplicação em vários sistemas [75].

A visão industrial foi afetada positivamente com o desenvolvimento das FPGA. A aquisição de imagem de elevada qualidade só é possível a partir de sensores com alta resolução e *frame rate*. Devido a este facto, as FPGA caracterizadas por uma elevada portabilidade, podem ser utilizadas como interface para o sensor. Este tipo de tecnologia, é também aplicado na fase de processamento de imagem do sistema, no qual, é necessário uma elevada velocidade de tratamento de dados e paralelismo, de forma a ser garantida a correta análise de objetos [76]. Na tabela 2.8 são apresentadas quatro FPGA da empresa *Xilinx*, assim como, as suas aplicações na indústria.

Tabela 2.8: Características de várias FPGA da empresa *Xilinx* (Adaptado de [76])

Dispositivo	**Características**	**Aplicações Industriais**
Spartan-6	Baixo custo Elevada portabilidade	Controlo de Motores
Artix-7	Baixo custo	Controlo de Motores Módulos de I/O
Kintex-7	Melhor rácio preço/ desempenho	Transmissão e processamento de dados Aplicações de vídeo
Zynq-7000	Processador e FPGA	Sistemas de visão Vídeo vigilância

O contributo das FPGA nos sistemas de visão industrial estende-se ao controlo de qualidade, em [77] foi desenvolvido um sistema industrial onde A FPGA desempenha um papel importante no processamento de imagem através de duas análises ao objeto: deteção do tipo e orientação. De notar que, as operações aplicadas à imagem, através do *software Matlab* (*Simulink*), foram: binarização, extração de características e classificação do objeto. Os dados resultantes da aplicação destes algoritmos são usados para dirigir os produtos para as suas respetivas estações. Caso seja detetada uma orientação deficiente na peça industrial, um dispositivo mecânico é ativado, corrigindo a falha. Os resultados obtidos pelos autores mostram que o sistema apresenta valores de precisão de reconhecimento de produtos na ordem dos 99% .

Em [78], com a finalidade de classificar telhas cerâmicas quanto ao seu tipo e identificar os seus possíveis defeitos, foi implementado um sistema baseado em FPGA. O processamento da imagem da telha (imagem a preto e branco, onde o objeto se encontra representado a branco num fundo preto) foi realizado através de VHDL. Foi implementada uma máquina de estados que controla o sistema da seguinte forma: iterando a imagem através das suas linhas, irão ser detetados os píxeis brancos referentes à telha (através de um valor de referência calculado previamente). No final deste varrimento, a telha estará devidamente identificada. Caso existam píxeis escuros localizados dentro da área do objeto, ou nos seus contornos, os módulos desenvolvidos em VHDL detetam as anomalias. Um algoritmo da mesma natureza, implementado em *C++* executado num computador (processador *T7300*), foi utilizado para serem comparados resultados. Os autores referem que o desempenho através da linguagem de descrição de *hardware* é cerca de 6 vezes mais rápido quando comparado a um desenvolvimento do sistema baseado em *C++*.

A deteção de objetos realizada em ambientes industriais, foi uma das áreas que obteve vantagens através do uso de FPGA na composição dos seus sistemas. Em [79], foi utilizada esta plataforma para realizar o pré-processamento de uma imagem com a finalidade de serem identificadas peças industriais. O procedimento inicia-se com a aquisição de imagem de um tapete rolante que será pré-processada via FPGA. Este tratamento de dados é efetuado da seguinte forma: dois módulos são utilizados, o primeiro deteta os contornos da imagem e o segundo deteta os buracos característicos dos produtos analisados. Para a deteção de contornos, foi utilizado um filtro gaussiano de forma a diminuir o ruído da imagem original, de seguida, através de um valor de referencia previamente definido, foram identificados os contornos do objeto. A esta imagem pode ser efetuada uma operação morfológica (dilatação), dependendo do valor de um registo controlado por um DSP. O segundo módulo efetua a binarização da imagem, dando assim, informação acerca dos buracos característicos nas peças industriais, de modo a possibilitar a identificação do tipo de produto. Depois de concluída esta fase, o segundo processamento irá ser efetuado por um processador (DSP), no qual, é localizada e classificada a peça em questão. A informação obtida é recebida por um robô que irá recolher o objeto identificado. Depois da análise de resultados, foi concluído pelos autores que o pré-processamento da imagem

via FPGA, acelera o funcionamento do sistema.

Em [80], utilizou-se uma solução SoC baseada em FPGA (processador *Nios-II* integrado), que deteta imagens do efeito de um laser num metal. A imagem capturada por uma câmara sofre um processamento através de filtros de binarização e erosão (implementados em *Verilog* em FPGA). Este tratamento de dados, é usado para analisar as características da zona de efeito do laser (dimensões). Devido à extração de características ser de difícil implementação e de pouca eficiência por parte do *hardware*, foi realizada através do microprocessador *Nios-II*. Uma das considerações dos autores foi o facto da elevada flexibilidade da FPGA ter sido um fator bastante relevante durante o desenvolvimento do sistema. É também referido que os resultados obtidos demonstram que o uso de FPGA aumentou a eficiência do sistema de visão industrial em questão.

2.4 Considerações Finais

Nesta secção, são apresentadas algumas considerações acerca das soluções existentes de configuração de FPGA e o seu uso nos sistemas de visão industrial (2.4.1), assim como, a apresentação de bibliotecas de processamento de imagem para FPGA já existentes (2.4.2).

2.4.1 Considerações

Na secção 2.2, foi realizada uma análise ao processamento de imagem baseado em FPGA. Primeiramente, foram apresentados vários estudos acerca do desempenho de três plataformas (FPGA, GPU e CPU) quando aplicadas ao processamento de imagem, de forma a serem identificadas as suas características, vantagens e desvantagens na área em estudo. Na secção 2.2.2.2, foram apresentadas algumas considerações acerca desta comparação, destacando-se o facto das três possibilitarem o processamento de imagem mas, devido às características de cada uma, o seu desempenho depende do sistema em que estão inseridas. De seguida, foi efetuado um estudo acerca de uma estrutura (SoC) que integra duas das plataformas estudadas anteriormente (CPU e FPGA), ou seja, é constituída por blocos de *hardware* e *software*. A partir dos artigos analisados em 2.2.3, é percetível que uma das vantagens da sua utilização é fornecer ao projetista a escolha do bloco (*hardware* ou *software*) que irá executar cada parte do código, com a finalidade de um melhor desempenho na execução mesmo.

No seguimento desta análise, foram apresentados métodos de configuração de FPGA (HDL e HLS), assim como, a estrutura de programação OpenCL. Em relação às HDL, foram analisados exemplos de aplicações de processamento de imagem baseados em VHDL e *Verilog*, nos quais, pode ser considerado que a implementação de sistemas através destas linguagens é mais demorada, devido a apresentarem alguma complexidade de configuração, mas, apresentam vantagens como a descrição de vários circuitos digitais (flexibilidade), assim como, um maior controlo no desenvolvimento dos mesmos. O método HLS, gera uma arquitetura RTL a partir de um sistema digital descrito em linguagem de

alto nível. A partir dos estudos apresentados em 2.2.5, pode ser considerado que uma das vantagens da sua utilização é o facto da arquitetura gerada ter incluída a parte de dados e a parte de controlo do sistema digital, ao contrário de uma implementação em HDL, na qual o projetista tem de configurar as duas partes do circuito. Este método também apresenta algumas limitações como: técnicas de programação a evitar (recursividade do código e uso de apontadores), assim como, a noção de tempo em *software* estar implícita, o que dificulta a tarefa de implementação de circuitos digitais síncronos (dependentes dos ciclos de *clock*). Por último, foi analisada a estrutura de programação de sistemas do tipo dispositivo/*host* (OpenCL). A partir dos estudos apresentados em 2.2.6, pode ser considerado que este método apresenta vantagens como uma maior rapidez na implementação porém, também apresenta limitações como a utilização de uma quantidade maior recursos por parte da FPGA quando comparada com soluções configuradas em HDL.

Tendo em conta as vantagens e limitações de cada um dos métodos de configuração de FPGA apresentados, a linguagem escolhida para a implementação da biblioteca genérica de filtros foi o VHDL. Os argumentos principais desta escolha foram o facto do método em questão providenciar um maior controlo no desenvolvimento de sistemas ao projetista, assim como, a menor utilização de recursos quando comparada com outras formas de configuração de FPGA.

Em 2.3, foram analisados sistemas de visão industrial e o contributo do processamento de imagem através da plataforma FPGA nos mesmos. Vários exemplos foram apresentados, nos quais, a configuração de FPGA foi efetuada através dos métodos analisados em 2.2.4 e 2.2.5. De notar que, na maioria dos projetos analisados a FPGA tem um papel de pré-processamento de imagem no funcionamento do sistema de visão. Tendo em conta os estudos analisados, os filtros mais utilizados são: conversão de RGB para tons de cinzento, remoção de ruído, binarização, deteção de contornos, dilatação, erosão, filtro de definição de contraste e negativo.

2.4.2 Bibliotecas de Processamento de Imagem para FPGA

Sendo o objetivo da presente dissertação a implementação de uma biblioteca de filtros de pré-processamento de imagem em FPGA, de seguida, serão apresentadas algumas bibliotecas já existentes.

A *HiFlipVX* é uma biblioteca de processamento de imagem com aplicação em FPGA-SoC. Esta foi implementada através de HLS e alguns dos filtros da sua composição são: erosão, dilatação, filtro gaussiano, *Sobel*, mediana, entre outros. Os resultados obtidos pelos autores em comparação com a plataforma *xfOpenCV* foram positivos pois, o consumo de recursos por parte da solução descrita é menor [81].

Outro exemplo é a biblioteca descrita em [82]. Foi desenvolvida através de HLS (linguagem *C*). Dois dos filtros que fazem parte da composição desta biblioteca (filtro gaussiano e *Sobel*) foram comparados com os algoritmos presentes na plataforma OpenCV. Os resultados obtidos pelos autores foram positivos pois, a solução proposta é mais eficiente.

Capítulo 3

Pré-Processamento de Imagem

Este capítulo é de carácter descritivo, no qual, irá ser relatado o desenvolvimento da biblioteca de métodos de pré-processamento de imagem. Em primeiro lugar, de forma a contextualizar o leitor, é feita uma descrição de alto nível do projeto (3.1). Em segundo lugar, é apresentado o material utilizado (3.2), e de seguida, é explicitada a arquitetura base do projeto (3.3), onde são descritos os diversos módulos/Intellectual Property (IP) e o protocolo de comunicação presente na mesma. Posteriormente, são apresentados os métodos desenvolvidos através de VHDL (3.4). Por último, é relatado o processo de transmissão (3.5) e receção de dados (3.6) provenientes da FPGA para CPU via Ethernet, viabilizando assim, o processo de validação dos filtros e integração do projeto em aplicações reais.

3.1 Introdução

Com o intuito de desenvolver um sistema de pré-processamento de imagem em tempo real em FPGA, foi escolhida uma plataforma com a capacidade de captar vídeo, possibilitar o seu processamento e enviá-lo para CPU. Tendo como base um projeto existente (*Zybo Z7 Pcam 5C demo*), foram feitas adições e modificações aos módulos do mesmo, de forma a processar o vídeo recebido. Os resultados obtidos foram enviados através do protocolo de comunicação *Ethernet* (UDP), sendo recebidos por um CPU. Na figura 3.1, é possível visualizar um diagrama genérico do sistema implementado.

Nas secções seguintes, são descritas detalhadamente as várias partes do projeto desenvolvido.

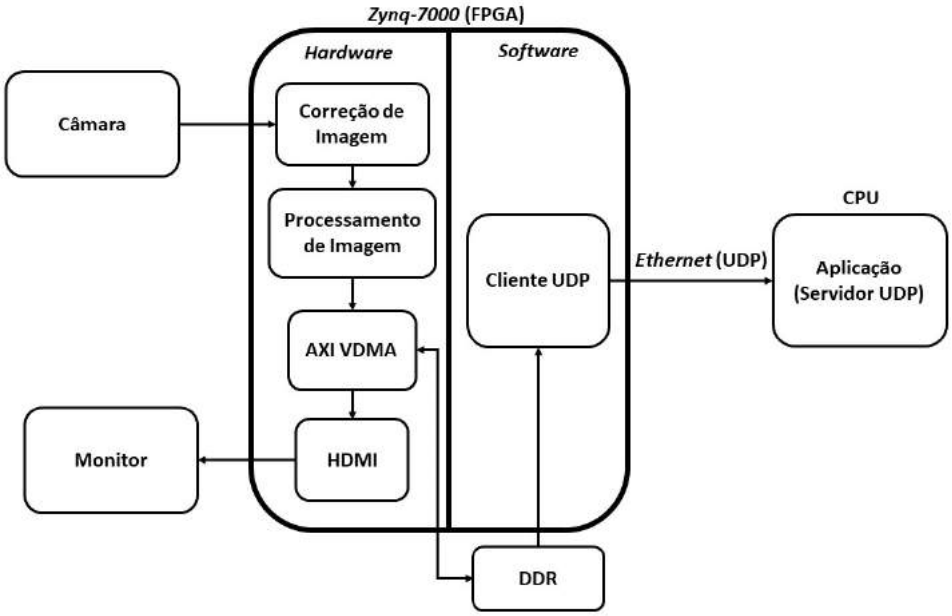


Figura 3.1: Diagrama genérico do sistema de pré-processamento de imagem em tempo real desenvolvido

3.2 Material Utilizado

De forma a implementar módulos de pré-processamento de imagem, foi utilizada a plataforma *Zybo Z7-20*. Esta oferece um meio de desenvolvimento de projetos baseados nos dispositivos da família *Zynq-7000*. O dispositivo presente é um SoC que integra uma FPGA *Xilinx 7-series* e um processador *ARM Cortex-A9*, permitindo uma maior diversidade de projetos, assim como, a possibilidade de executar o código em *hardware* ou *software*. A plataforma de desenvolvimento pode ser alimentada através de Universal Serial Bus (USB), uma bateria ou uma fonte externa (5 *Volt*) e apresenta vários elementos periféricos como: conetor de câmara *Pcam* (receção de vídeo através do protocolo *MIPI CSI-2*), conetor High-Definition Multimedia Interface (HDMI) de entrada e saída, uma memória DDR com elevada largura de banda e um conetor *Ethernet*, viabilizando o envio e receção de dados. Note-se que, existem mais periféricos na plataforma *Zybo Z7-20* mas, apenas são apresentados aqueles que foram utilizados no desenvolvimento do projeto. Na figura 3.2, é possível observar a disposição dos componentes mencionados. Como referido anteriormente, a constituição do SoC utilizado (*z-7020*) baseia-se em duas partes distintas: lógica programável e sistema de processamento (CPU). As suas características principais estão presentes na tabela 3.1.

Para permitir validar os métodos desenvolvidos em FPGA, foi utilizado um computador (*LAPTOP-C08RVMIO*) com as seguintes características: processador *Intel(R) Core(TM) i7-8550U* CPU @ 1.80 GHz 2.00 GHz (4 núcleos), 16 GB de memória RAM e 256 GB de memória SSD. Este computador recebe imagens da FPGA, compara-as e analisa se as mesmas foram processadas corretamente.

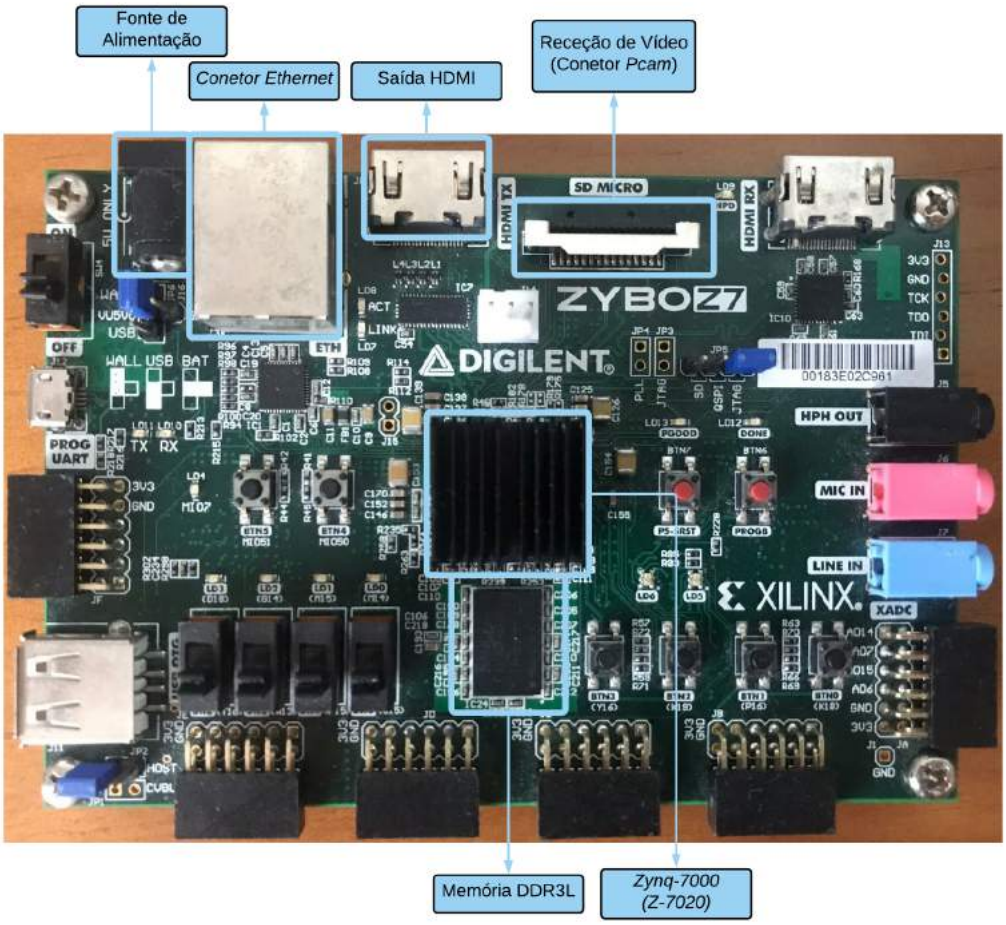


Figura 3.2: Disposição dos elementos da plataforma *Zybo Z7-20* utilizados no desenvolvimento do projeto

Tabela 3.1: Características do SoC presente na plataforma de desenvolvimento *Zybo Z7-20*

Zynq-7000 (Z-7020)

Lógica Programável		**CPU**	
Células Programáveis	85.000	Processador	*Dual Core ARM Cortex-A9*
lookup table	53.200	Memória Externa	DDR3L
LUTRAM	17.400	Canais de Acesso Direto à Memória	8
flip-flop	106.400	Periféricos	2 x UART 2 x I2C 4 x 32 *bits* GPIO
BRAM	140		
IOB	125		

3.3 Arquitetura Base (*Demo*)

O desenvolvimento dos métodos presentes na biblioteca foi realizado através do *software Vivado 2016.4* e teve como base o *Zybo Z7 Pcam 5C demo* [83]. Este apresenta como sinal de entrada vídeo captado por uma câmara (*Pcam 5C*), conectada a uma plataforma *Zybo Z7-20*. O sinal recebido (pixel a pixel) é convertido para RGB, sendo de seguida, armazenadas três *frames* ciclicamente numa memória DDR. Por último, o vídeo recebido é visualizado num monitor via HDMI (sinal de saída). Um dos atributos adicionais deste *demo* é a utilização da Universal Asynchronous Receiver-Rransmitter (UART) através do processador *Zynq-7000*, com a finalidade de modificar as características do vídeo por parte do utilizador (resolução, formato de imagem e a correção de gama). De notar que, este sistema tem como base o protocolo de comunicação *AXI4-Stream* que apresenta uma frequência de *clock* de 150 MHz. Nesta secção, é apresentada uma explicação mais detalhada dos módulos do projeto base utilizado e do protocolo de comunicação (3.3.1), assim como, uma descrição da configuração dos vários IP do sistema, através da ferramenta de *software Vivado SDK* (3.3.2).

3.3.1 *Hardware*

Nesta subsecção, em 3.3.1.1 é descrito o protocolo de comunicação utilizado no *demo* (*AXI4-Stream*), assim como, uma explicitação dos módulos da arquitetura base dividida em três secções: captação e correção de imagem (3.3.1.2), VDMA (3.3.1.3) e HDMI (3.3.1.4).

3.3.1.1 *AXI4-Stream*

O *interface* Advanced Extensible Interface 4 (AXI4) apresenta uma comunicação do tipo *master/slave*, permitindo troca de dados entre vários AXI4 *masters* e *slaves* [84]. Este é suportado por vários IP e apresenta três protocolos: AXI4, *AXI4-Lite* e *AXI4-Stream*.

No projeto base, o protocolo de comunicação *AXI4-Stream* é utilizado na transmissão de cada pixel de cada *frame* do vídeo recebido para a saída HDMI. Este apresenta vários sinais que garantem a correta comunicação entre dois módulos (tabela 3.2). De notar que, existem mais sinais na sua constituição mas, apenas são apresentados os utilizados no desenvolvimento do presente projeto.

Tabela 3.2: Sinais do protocolo *AXI4-Stream* utilizados no desenvolvimento do projeto

Sinal	**Nº de Bits**	**Descrição**
TDATA	24	Representa o valor do pixel em questão
TVALID	1	Indica se o *master* tem informação para enviar
TREADY	1	Indica se o *slave* está preparado para receber dados
TUSER	1	Assinala o primeiro pixel de cada *frame*
TLAST	1	Assinala o último pixel de cada linha de cada *frame*

O sinal TDATA apresenta 24 *bits* nos quais é representado o valor de cada pixel RGB, sendo que, cada componente apresenta 8 *bits*. Os sinais TVALID e TREADY têm como

função garantir a correta comunicação entre módulos. O primeiro indica se o *master* tem dados para enviar ao *slave*, enquanto que, o segundo indica se o *slave* está preparado para receber informação proveniente do *master*. Note-se que, apenas quando ambos apresentam o valor lógico '1' existe transmissão de dados. O sinal TUSER assinala se o pixel em questão está na primeira posição de cada *frame*, ou seja, o pixel correspondente ao canto superior esquerdo. Por último, o sinal TLAST assinala se o pixel corrente está no final de cada linha de cada *frame*. Na figura 3.3, é possível visualizar como é feita a comunicação entre dois módulos através dos sinais descritos anteriormente.

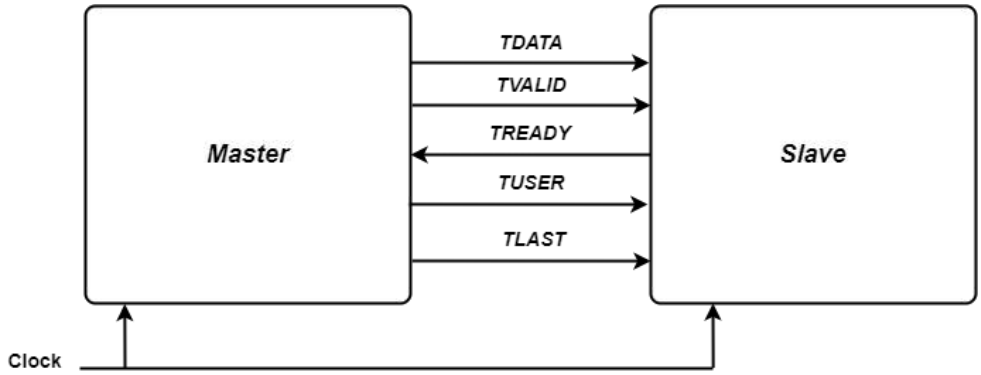


Figura 3.3: Representação da comunicação *master*/*slave* através do protocolo *AXI4-Stream*

3.3.1.2 Captação e Correção de Imagem

O *demo* descrito anteriormente apresenta como sinal de entrada vídeo captado por uma câmara (*Pcam 5C*), sendo esta, conectada à plataforma de desenvolvimento *Zybo Z7-20* através do protocolo *MIPI CSI-2*. De seguida, é efetuada uma correção de imagem de modo a possibilitar acesso ao valor RGB de cada pixel de cada *frame* do vídeo captado.

A captação de imagem é efetuada através do protocolo *MIPI CSI-2*, caracterizado pela elevada velocidade de transferência de dados entre dispositivos (ligação ponto a ponto). Este apresenta como principais aplicações a transmissão de vídeo ou imagem entre uma câmara e um *host*. De modo a receber a informação proveniente da câmara, foram utilizados os IP *MIPI_D_PHY_RX* [85] e *MIPI_CSI_2_RX* [86].

O *MIPI_D_PHY_RX* implementa a camada física do *interface MIPI D-PHY*. São recebidas três linhas de dados (uma referente ao sinal de *clock* e duas de informação captada pela câmara), sendo que, é efetuada uma desserialização dos dados recebidos, e de seguida, uma compressão dos mesmos numa única linha de dados.

O *MIPI_CSI_2_RX* tem como função interpretar a informação proveniente do IP anterior em píxeis, linhas e *frames*. Os dados na entrada deste módulo são formatados em RAW RGB e compactados no protocolo *AXI4-Stream*, possibilitando assim, o acesso ao valor de cada pixel do vídeo captado pela câmara. Note-se que, na saída deste módulo o vetor correspondente ao valor do pixel em questão apresenta 40 *bits*.

De seguida, de modo a aceder ao valor RGB de cada pixel, é utilizada uma secção de correção de imagem composta pelos módulos *AXI_BayerToRGB* e *AXI_GammaCorrection*.

O *AXI_BayerToRGB*, aplica um filtro *Bayer* ao valor do pixel na sua entrada, ou seja, é efetuada a sua conversão para RGB. O vetor de saída correspondente é composto por 32 *bits*, 10 *bits* por componente RGB e 2 *bits* de *padding*.

O *AXI_GammaCorrection* tem como função corrigir a gama do vídeo captado pela câmara e diminuir a dimensão do valor de cada pixel. Relativamente ao fator de correção de gama, pode tomar diferentes valores (1; 1/1,2; 1/1,5; 1/1,8; 1/2,2), sendo que, as regiões escuras do vídeo original apresentam tons mais claros quando aplicado um fator de menor valor, e vice-versa. Note-se que o fator utilizado no *demo* é 1/1,8. Por último, a dimensão do vetor correspondente a cada pixel é limitada a 24 *bits*, possibilitando assim, cada componente apresentar valores entre 0 e 255 (8 *bits*) [87]. Na figura 3.4, é possível visualizar o diagrama de blocos das zonas de captação e correção de imagem do *demo*.

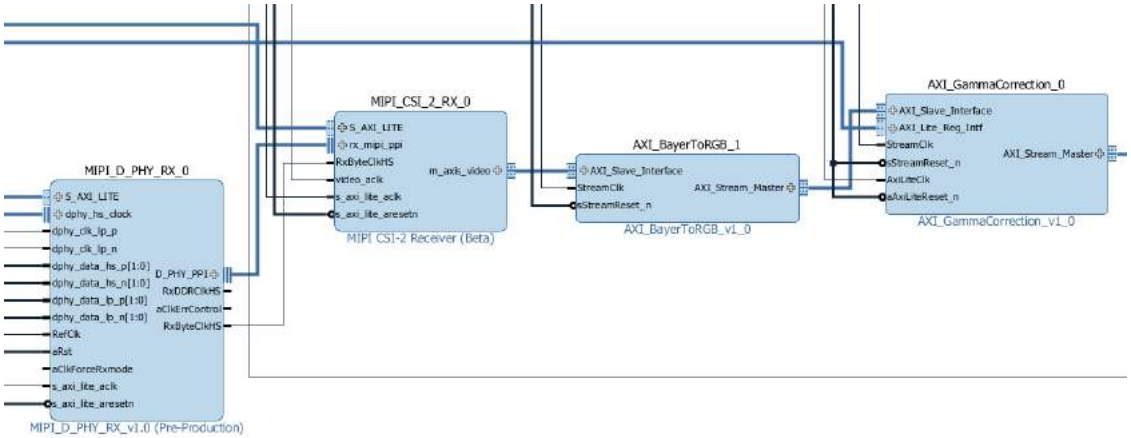


Figura 3.4: Zona de captação e correção de imagem no *demo*

3.3.1.3 VDMA

Um dos módulos mais importantes na constituição do *demo* é o VDMA (*Video Direct Acess Memory*) [88] responsável pelo armazenamento direto da *frame* atual numa memória DDR, sendo possível aceder à mesma através do processador *Zynq-7000*. Este apresenta 3 vetores referentes a 3 *frames* do vídeo captado, sendo estas armazenadas ciclicamente, ou seja, a quarta *frame* irá ser guardada no endereço de memória correspondente à primeira e assim sucessivamente. O protocolo *AXI4-Stream* está presente neste IP como sinal de entrada e como sinal de saída, sendo este o portador de informação a armazenar, assim como, o meio de transmissão do valor do pixel atual para os módulos seguintes. A sua configuração pode ser efetuada através da ferramenta de *software Vivado SDK*. Na figura 3.5, é possível visualizar o módulo VDMA e o processador *Zynq-7000* presentes no *demo*.

O seu funcionamento pode ser descrito em cinco fases:

1. Receção do valor do pixel atual (*AXI4-Stream*)

2. Conversão da *stream* recebida para *memory mapped*

3. Armazenamento do valor convertido num determinado endereço da memória DDR (previamente escolhido pelo processador)

4. Leitura do valor armazenado e conversão para *streaming*

5. Transferência da *stream* convertida para *AXI4-Stream* que irá possibilitar a transmissão do valor do pixel para os módulos seguintes

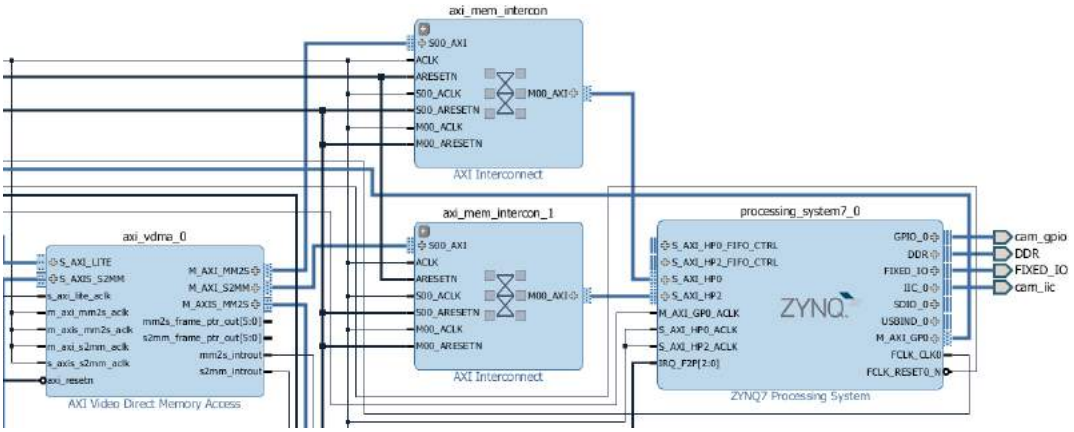


Figura 3.5: Módulo VDMA e processador *Zynq-7000* presentes no *demo*

3.3.1.4 HDMI

O sinal de saída do sistema é a visualização do vídeo captado pela câmara *Pcam 5C* via HDMI. De modo a ser possível ter esta característica no *demo*, são utilizados quatro módulos (*Dynamic clock Generator*, *Video Timing Controller*, *AXI4-Stream to Video Out* e *RGB to DVI*), dos quais, os dois primeiros têm como função gerar os elementos principais do HDMI (sinal de *clock* e sinais de controlo) e os dois últimos, destinam-se a converter os dados provenientes do VDMA para a saída HDMI.

O *Dynamic clock Generator* gera o sinal de *clock* a ser utilizado pelo HDMI, sendo que, a frequência deste sinal depende da resolução previamente escolhida pelo processador *Zynq-7000* (*software*). Relativamente ao *Video Timing Controller* [89], apresenta como função gerar os sinais de controlo do HDMI (*VSync*, *HSync*, *VBlank*, *HBlank* e *Active Video*). De notar que, o sinal de *clock* gerado pelo módulo anterior é utilizado neste IP de modo a ser garantida a correta sincronização.

O *AXI4-Stream to Video Out* [90], usa os sinais dos dois módulos anteriormente explicitados e converte a *stream* relativa ao pixel atual (proveniente do VDMA) num *interface* de vídeo. De seguida, o módulo *RGB to DVI* [91] converte os dados para *Digital Visual Interface* (DVI) de modo a ser possível transmitir os mesmos pelo conector HDMI. Na figura 3.6, é visível a secção do *demo* referente à saída HDMI.

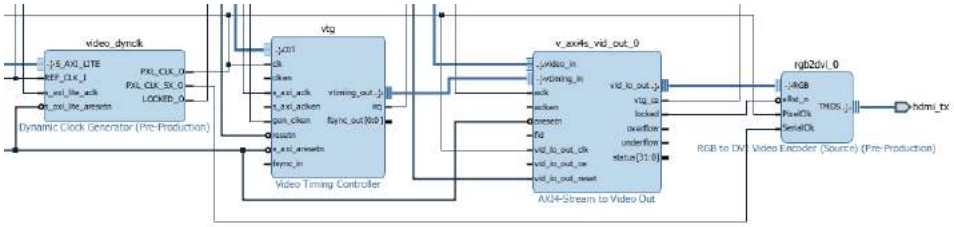


Figura 3.6: Módulos constituintes da saída HDMI do *demo*

3.3.2 *Software*

Tendo em conta que a plataforma utilizada neste projeto é um SoC, o *demo* referido utiliza o processador *Zynq-7000* com o intuito de inicializar alguns módulos descritos anteriormente, assim como, possibilitar a modificação de características do vídeo por parte do utilizador, via UART. A configuração do processador *Zynq-7000* é efetuada através da ferramenta de *software Vivado SDK* (linguagem *C/C++*). Nesta secção é feita uma descrição da função principal e do método responsável pela modificação da resolução e início da transmissão de vídeo.

O método principal tem como objetivo inicializar os IP referentes à parte de *hardware* do projeto, iniciar a transmissão de vídeo, e por último, oferecer ao utilizador a capacidade de modificação de características do vídeo.

O seu funcionamento pode ser descrito da seguinte forma: em primeiro lugar é inicializada a parte de *hardware* e o sinal de ativação da câmara (*General Purpose Input/Output*, GPIO). Em segundo, é inicializada a câmara através do GPIO, sendo de seguida, efetuada a configuração do primeiro módulo presente na parte de *hardware* (VDMA). Em terceiro lugar, são configurados dois registos que estão ligados à saída HDMI (*Dynamic clock Generator* e *Video Timing Controller*). De seguida, é chamada a função *pipeline_mode_change* que modifica a resolução e inicia a transmissão de video através da configuração da câmara e dos seguintes IP: *MIPI_CSI_2_RX*, *MIPI_D_PHY_RX*, VDMA, *AXI_GammaCorrection*, *Dynamic clock Generator* e *Video Timing Controller* [87].

A função *pipeline_mode_change* tem o seguinte funcionamento:

1. O bloco de processamento é inicializado
2. É definido o fator de correção de gama e a câmara é reinicializada
3. A resolução do vídeo é definida (resolução predefinida é 1920x1080 píxeis)
4. O módulo *Video Timing Controller* é reinicializado, configurado, e por fim, ativado
5. Início da transmissão de vídeo

Por último, esta aplicação oferece ao utilizador a capacidade de alterar as características do vídeo através da consola, tais como: resolução, formato de imagem e correção de gama.

3.4 Métodos Desenvolvidos

Depois de analisada a arquitetura base (*demo*), foram desenvolvidos e adicionados módulos de pré-processamento de imagem de forma a ser possível modificar o vídeo recebido. A sua implementação foi efetuada através de linguagem VHDL, sendo que, cada filtro recebe o valor de um pixel como sinal de entrada (RGB, tons de cinza ou preto e branco) e apresenta como saída, o valor processado correspondente. Para esse efeito, foram desenvolvidos módulos baseados no protocolo *AXI4-Stream* de modo a permitir a sua integração no *demo*. Na figura 3.7 é visível o diagrama de blocos correspondente à estrutura dos módulos de pré-processamento de imagem implementados.

Figura 3.7: Representação gráfica da estrutura dos módulos de pré-processamento de imagem

Com o intuito de filtrar o vídeo recebido pela câmara, foi necessário perceber onde colocar os módulos desenvolvidos no *demo*. Estes foram inseridos entre o bloco de correção de gama e o VDMA, possibilitando assim, a modificação de píxeis no formato RGB (cada componente apresenta 8 *bits*) e o armazenamento de cada *frame* processada em memória (DDR).

Por fim, foram escolhidos os métodos de pré-processamento de imagem a desenvolver:

1. Conversão de RGB para tons de cinzento (3.4.2)
2. Conversão de RGB para YCbCr (3.4.3)
3. Negativo (3.4.4)
4. Modificação de Brilho (3.4.5)
5. Binarização (3.4.6)
6. *Sobel* (3.4.7)
7. Média Uniforme (3.4.8)
8. Gaussiano (3.4.9)
9. Erosão (3.4.10)
10. Dilatação (3.4.11)

Ao analisar a lista anterior é possível identificar dois tipos de filtros, aqueles onde o pixel a processar depende única e exclusivamente do seu valor (cinco primeiros métodos), e os métodos onde o pixel a processar depende de si próprio, dos valores presentes na sua vizinhança e de uma máscara de convolução (últimos cinco).

Nesta secção, são explicitados os algoritmos base (3.4.1) e as equações utilizadas no desenvolvimento de cada módulo de pré-processamento de imagem, presentes na lista acima.

3.4.1 Algoritmo Base

Nesta subsecção são introduzidos os algoritmos utilizados no desenvolvimento dos métodos de pré-processamento de imagem constituintes da biblioteca. Em primeiro lugar, é descrito processo de implementação dos filtros cujo pixel a processar depende apenas de si próprio (3.4.1.1), e de seguida, é apresentado o algoritmo correspondente aos métodos que necessitam do auxílio de uma máscara de convolução, devido ao pixel a processar depender de si próprio e dos valores da sua vizinhança (3.4.1.2).

3.4.1.1 Algoritmo sem Máscara de Convolução

Nos métodos conversão de RGB para tons de cinzento, conversão de RGB para YCbCr, negativo, modificação de brilho e binarização, o processamento de qualquer pixel recebido depende única e exclusivamente de si próprio. Tendo em conta que o valor de entrada é recebido através do protocolo *AXI4-Stream* e como a sua localização na *frame* pode ser desprezada, foram apenas utilizados os sinais TDATA, TVALID e TREADY no algoritmo desenvolvido. O sinal TDATA oferece acesso ao valor das três componentes do pixel em questão. Os sinais TVALID e TREADY foram usados para garantir a correta comunicação entre módulos. Na figura 3.8, é possível visualizar o fluxograma representativo do algoritmo utilizado neste tipo de filtros.

Por análise da figura 3.8, é visível que a primeira condição a ser verificada é o sinal de *reset*. Caso este apresente o valor lógico '0', o pixel de saída equivale a zero, ou seja, apresentará cor preta. Caso contrário, será verificado se o sinal de *clock* está ativo (flanco ascendente). Se a condição for falsa, o pixel de saída apresentará o valor anterior, se for verdadeira, é averiguada a correta transmissão de dados através dos sinais do protocolo *AXI4-Stream*. Se os sinais TVALID e TREADY estiverem ambos ativos, o pixel de entrada é processado, sendo que, o resultado desta operação é apresentado na saída, caso contrário, o pixel de saída toma o valor anterior.

3.4.1.2 Máscara de Convolução

A implementação em VHDL dos métodos *Sobel*, remoção de ruído (média e gaussiano), erosão e dilatação é mais complexa relativamente aos filtros anteriores pois, o pixel a processar depende de si próprio, dos valores da sua vizinhança e de uma máscara de

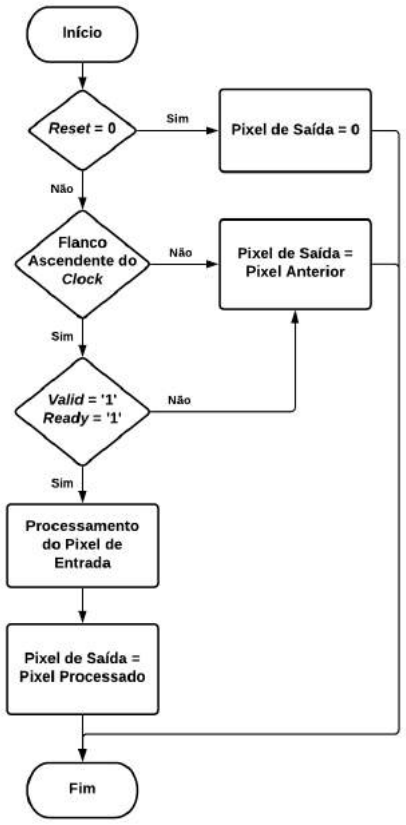


Figura 3.8: Fluxograma representativo do funcionamento dos métodos cujo pixel a processar depende apenas de si próprio

convolução. Foi escolhida uma máscara 3x3 para implementar os filtros referidos. Devido à elevada resolução de imagem (1920x1080 píxeis), optou-se por não aplicar o método nas margens de modo a não dificultar a implementação do algoritmo e aumentar a sua velocidade. Na figura 3.9, é visível que às duas primeiras linhas, assim como, às duas primeiras e duas últimas colunas não foi aplicado nenhum algoritmo, ou seja, os seus valores correspondem aos valores dos pixeis captados pela câmara.

De modo a aplicar estes métodos, é necessário ter o conhecimento da localização do pixel a processar na *frame*. Como referido anteriormente, o processamento é realizado pixel a pixel e cada *frame* é gerada da esquerda para a direita e de cima para baixo, o que não permite a aplicação imediata do filtro após receber o pixel central pois, esta operação depende de vizinhos que ainda não foram recebidos. Na figura 3.10 é visível que não é possível processar o pixel com o valor 35 pois, a sua vizinhança não está completa.

O problema apresentado, motivou a alteração do procedimento anterior de modo a ser possível ter acesso a todos os píxeis contidos na vizinhança, e por conseguinte, possibilitar o processamento do valor presente da posição central da máscara.

O algoritmo utilizado tem o seguinte funcionamento: quando o pixel corrente está localizado na região a processar, a máscara de convolução desloca-se de modo a este estar

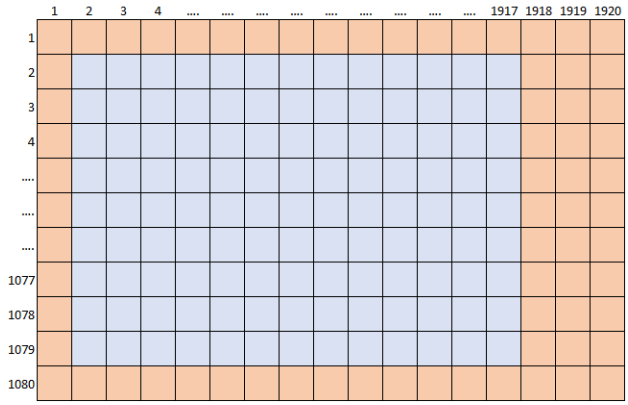


Figura 3.9: Representação gráfica das zonas de processamento de uma *frame* (a azul estão representados os píxeis que são processados e a laranja os que não são)

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	**24**	**25**	**26**	27	28	29	30
31	32	33	**34**	**35**	**36**	37	38	39	40
41	42	43	**44**	**45**	**46**	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

Figura 3.10: Representação gráfica de uma *frame* do vídeo e da região processamento incompleta quando aplicado o filtro à posição central da máscara de convolução (a bege estão representados os píxeis recebidos e a azul os não recebidos)

sempre presente no canto inferior direito da mesma, possibilitando assim, o acesso aos 9 valores necessários para a aplicação do método. Nestas condições, o pixel central é processado e colocado na imagem resultante com um desfasamento de uma linha e uma coluna em relação à posição original. Tendo isto em conta, a imagem resultante perderá as margens e será apresentada desfasada. Na figura 3.11, está representada a abordagem utilizada para o desenvolvimento dos métodos que necessitam do auxílio de uma máscara de convolução, onde é possível observar a imagem original do lado esquerdo, na qual, é visível que o processamento do pixel com o valor 35 já é possível, e a imagem resultante do lado direito.

Após ter sido desenvolvido o algoritmo de utilização da máscara de convolução 3x3, foram implementados módulos VHDL baseados no procedimento referido (filtros mencionados nesta subsecção). Para esse efeito, foram utilizados os sinais de controlo presentes na tabela 3.3.

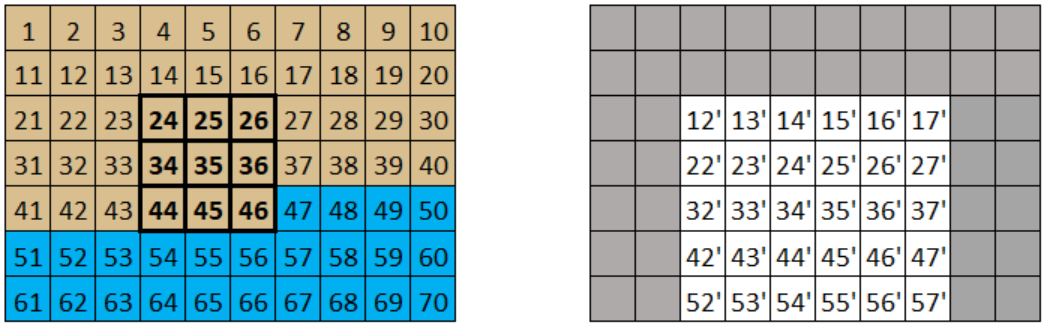


Figura 3.11: Representação gráfica da *frame* original (à esquerda) e da *frame* resultante (à direita) através da abordagem utilizada no desenvolvimento dos métodos que utilizam a máscara de convolução 3x3

Tabela 3.3: Sinais de controlo utilizados no desenvolvimento dos módulos que utilizam a máscara de convolução

Sinal	Valor	Dimensão	Descrição
Resolução_x	1920	-	Número de píxeis de uma linha de uma *frame*
Resolução_y	1080	-	Número de píxeis de uma coluna de uma *frame*
Contador_x	-	-	Indica a posição horizontal do pixel corrente
Contador_y	-	-	Indica a posição vertical do pixel corrente
Buffer_1	-	1918	Contém os valores dos píxeis da linha y-2 (y representa a linha atual)
Buffer_2	-	1918	Contém os valores dos píxeis da linha y-1
Buffer_3	-	2	Contém os valores dos dois píxeis anteriores ao corrente

Tendo em conta que a aplicação destes métodos depende da posição do pixel a processar, foi necessário utilizar dois contadores que identificam a localização do pixel corrente nas *frames* do vídeo ("Contador_x" e "Contador_y"). Note-se que, estas duas variáveis apresentam valores entre 0 e a resolução da *frame* correspondente (horizontal ou vertical).

Outra característica importante no desenvolvimento dos módulos é a ordem de receção dos píxeis (da esquerda para a direita e de cima para baixo) pois, com o intuito de utilizar a máscara de convolução 3x3, foi necessário armazenar duas linhas da *frame* em questão ("Buffer_1" e "Buffer_2") e dois píxeis anteriores à posição atual ("Buffer_3"). Note-se que, à medida que são recebidas novas linhas, a posição vertical do "Buffer_1" e "Buffer_2" é incrementada, enquanto que, o "Buffer_3" sofre alterações na sua posição sempre que um pixel é recebido. De modo a facilitar a implementação, as últimas duas colunas foram desprezadas, sendo assim, o "Buffer_1" e "Buffer_2" apresentam dimensão 1918 e o "Buffer_3" dimensão 2. Para uma melhor compreensão por parte do leitor, na figura 3.12 estão representadas três situações do uso da máscara de convolução através dos vetores descritos.

Na figura 3.12 à esquerda, está representada a situação de processamento do primeiro

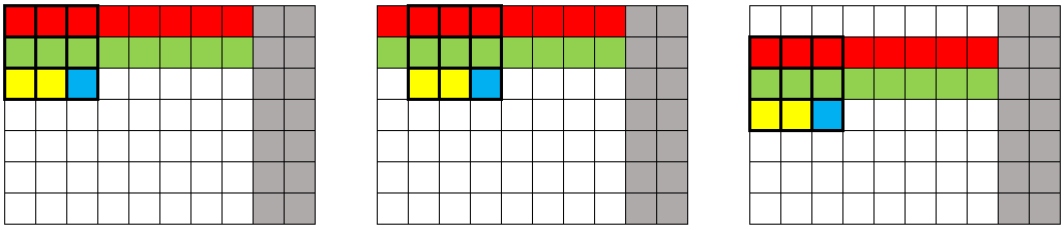


Figura 3.12: Representação gráfica de três situações da aplicação da máscara de convolução (a vermelho está representado o "Buffer_1", a verde o "Buffer_2", a amarelo o "Buffer_3", a azul o pixel corrente e a cinzento as duas colunas que não são processadas)

pixel, onde se pode visualizar as posições iniciais de cada vetor. A imagem central representa a aplicação da máscara de convolução à segunda posição a processar. Nesta situação, é visível que a localização horizontal do "Buffer_3" foi incrementada de modo a acompanhar a máscara. À direita, está representado o processamento do primeiro pixel da segunda linha, onde se pode observar o incremento vertical da posição dos três vetores. Em conclusão, à medida que os valores vão sendo recebidos, a localização dos vários *buffers* na *frame* é alterada de forma a ser garantida a correta aplicação do método.

Outra característica do algoritmo desenvolvido são os sinais do protocolo *AXI4-Stream*, sendo que, devido à posição do pixel a processar não poder ser desprezável foram utilizados os sinais TDATA, TVALID, TREADY, TLAST e TUSER. Na figura 3.13, está representado o fluxograma do funcionamento do algoritmo desenvolvido.

Por análise da figura 3.13, tal como nos métodos que não utilizam a máscara de convolução, as primeiras condições a serem verificadas são o *reset*, o sinal de *clock* (flanco ascendente) e os sinais TVALID e TREADY.

Após ser garantida a correta transmissão de dados, são verificadas condições referentes à localização do pixel corrente na *frame* em questão. Em primeiro lugar, é averiguado se este está localizado na primeira posição da *frame* (canto superior esquerdo), caso se confirme, a saída toma o valor de entrada, visto que, não são processadas as margens. De seguida, é preenchida a primeira posição do "Buffer_1"com o valor do pixel atual, e por último, o "Contador_x" é incrementado. Note-se que, esta parte do algoritmo é efetuada através de um dos sinais do protocolo *AXI4-Stream* (TUSER). Caso a condição anterior não se verifique, é averiguado se o pixel corrente se encontra na primeira ou segunda linha da *frame*. Caso o resultado seja verdadeiro, os vetores "Buffer_1" e "Buffer_2" são preenchidos (dependendo da localização atual), sendo que, o pixel resultante é sempre igual ao valor de entrada. Por último, é verificado se a posição corrente corresponde à última linha da *frame*, de modo a ser possível incrementar os contadores (horizontal e vertical) de forma correta. É utilizado o sinal TLAST do protocolo *AXI4-Stream*, sendo que, se este estiver ativo, o "Contador_x" toma o valor 0 e o "Contador_y" é incrementado (mudança de linha), caso contrário, apenas o contador "Contador_x" é incrementado.

Depois de verificadas as duas primeiras linhas, é averiguada a posição do pixel corrente relativamente às colunas que não são processadas (duas primeiras e duas últimas).

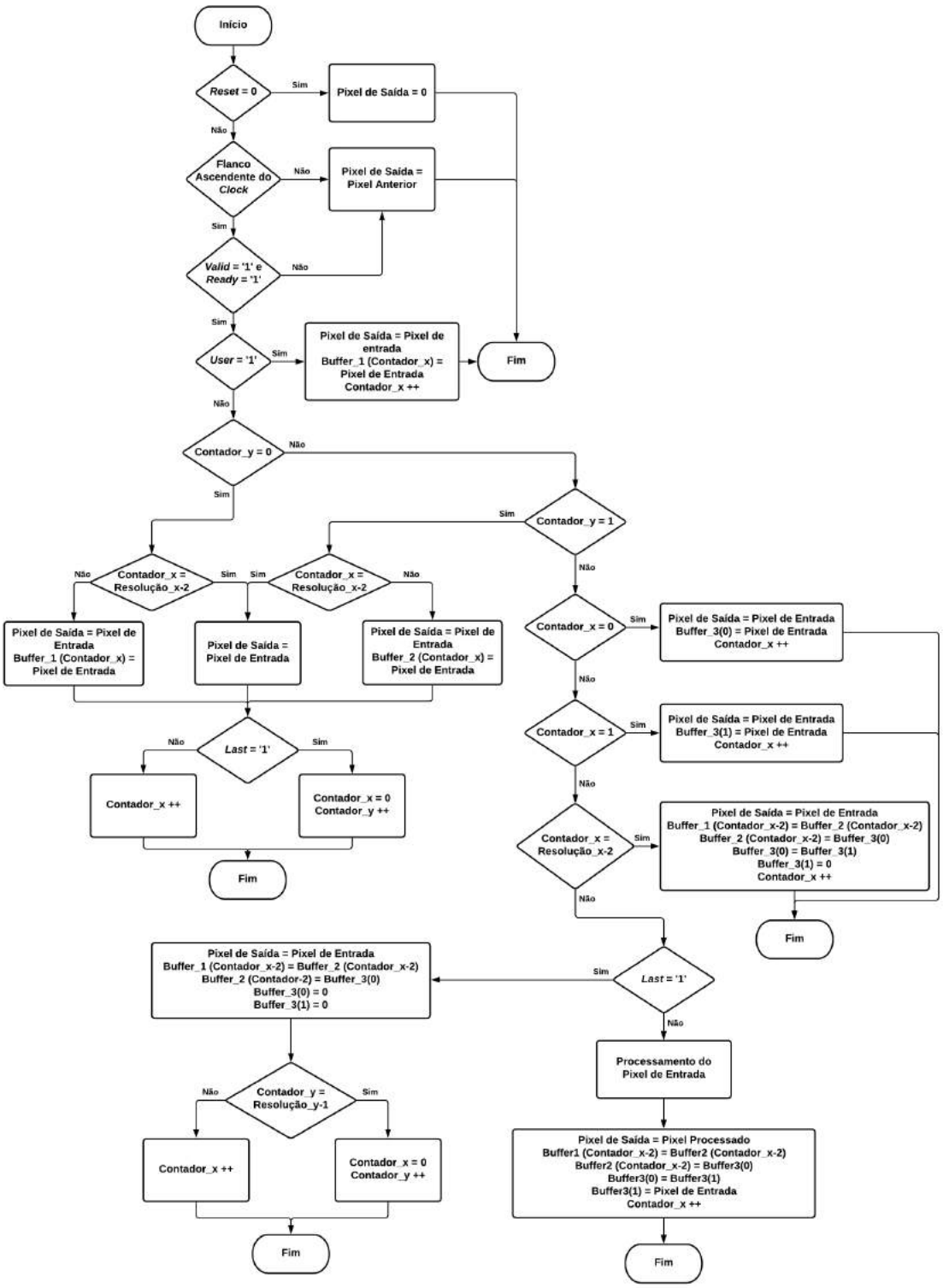


Figura 3.13: Fluxograma representativo do funcionamento dos métodos cujo pixel a processar depende de si próprio e dos valores presentes na sua vizinhança

Nestas situações, a saída toma sempre o valor de entrada. Caso se encontre na primeira ou segunda coluna, é preenchida a posição correspondente do "Buffer_3", de modo a ser possível ter acesso aos valores dos 8 vizinhos do pixel a processar, por último, o "Contador_x" é incrementado. Caso se verifique que o pixel corrente está localizado nas duas últimas colunas, todos os *buffers* são atualizados de forma a ser possível aceder a todos os valores da vizinhança dos píxeis das próximas linhas, visto que, as duas colunas em questão não sofrem processamento. De notar que, os contadores sofrem modificações distintas dependendo da localização na *frame*, na penúltima coluna o "Contador_x" é incrementado, enquanto que, na última coluna existem duas situações possíveis, se o pixel corrente estiver localizado numa das últimas linhas da *frame*, o "Contador_x" irá apresentar o valor 0 e o "Contador_y" será incrementado mas, se a sua posição for a última da *frame* (última coluna e última linha), ambos os contadores tomam o valor 0, ou seja, o próximo pixel será a primeira posição de uma nova *frame*.

Após serem averiguadas todas condições anteriores, ou seja, quando o pixel corrente está localizado na região a processar, é aplicado o filtro em questão. Os *buffers* são atualizados de forma a ser garantida a correta aplicação do método aos píxeis seguintes, e por último, o "Contador_x" é incrementado.

Nas subsecções seguintes, são explicitadas as equações utilizadas no desenvolvimento dos métodos constituintes da biblioteca, ou seja, é descrita a implementação do bloco "Processamento do Pixel de Entrada" nos fluxogramas apresentados (figuras 3.8 e 3.13) .

3.4.2 Conversão RGB/Cinza

O método de conversão RGB/cinza, tem como objetivo retirar informação de cada *frame* do vídeo captado através da modificação de um pixel RGB num tom de cinzento. O pixel recebido apresenta 3 componentes às quais é aplicada uma média aritmética, sendo que, o resultado desta operação é colocado em todas as componentes do pixel de saída, sendo obtido assim, o tom de cinza correspondente. Devido a ter sido utilizada a linguagem VHDL no desenvolvimento dos filtros, só é possível dividir valores por potências de 2. Tendo em conta esta especificidade, na equação 3.1 é apresentada a fórmula utilizada para realizar uma divisão por 3 aproximada.

$$Tom\ de\ Cinza = \frac{(R+G+B)\times 171}{512} \tag{3.1}$$

3.4.3 Conversão RGB/YCbCr

Com o intuito de modificar as *frames* RGB do vídeo captado para YCbCr, foi desenvolvido um módulo responsável por essa conversão. O método mais utilizado para o desenvolvimento deste filtro tem como base as equações 3.2, 3.3 e 3.4.

$$Y = 0.257\times R + 0.504\times G + 0.098\times B + 16 \tag{3.2}$$

$$Cb = -0.148 \times R - 0.291 \times G + 0.439 \times B + 128 \tag{3.3}$$

$$Cr = 0.439 \times R - 0.368 \times G - 0.071 \times B + 128 \tag{3.4}$$

Devido às dificuldades referentes à linguagem utilizada, foi feita uma aproximação das equações anteriores para ser possível o cálculo das componentes Y, Cb e Cr em VHDL (equações 3.5, 3.6, 3.7).

$$Y' = \frac{4 \times R + 8 \times G + 2 \times B + 256}{16} \tag{3.5}$$

$$Cb' = \frac{-2 \times R - 5 \times G + 7 \times B + 2048}{16} \tag{3.6}$$

$$Cr' = \frac{7 \times R - 6 \times G - B + 2048}{16} \tag{3.7}$$

3.4.4 Negativo

O método negativo tem como objetivo alterar o valor de cada pixel para o seu complemento, ou seja, os píxeis claros da imagem original irão apresentar cores escuras na imagem resultante, e vice versa. A sua implementação em VHDL, foi realizada através da modificação do valor das componentes RGB de cada pixel do vídeo captado através da equação 3.8.

$$Negativo(R,G,B) = 255 - Componente(R,G,B) \tag{3.8}$$

3.4.5 Modificação de Brilho

O filtro de modificação de brilho permite o processamento de vídeo num ambiente onde a iluminação não é perfeita, podendo ser ajustada pelo método. A sua implementação foi realizada através da adição de um valor de referência predefinido (positivo ou negativo), a todas as componentes (RGB) de cada pixel do vídeo (equação 3.9). Caso o valor de referência seja positivo, o resultado filtrado terá maior luminosidade, caso contrário, a imagem resultante apresentará menos brilho. De notar que, foi feita uma condição com o intuito de garantir que o resultado da soma esteja compreendido entre 0 e 255, visto que, cada uma das componentes possui 8 *bits* de resolução.

$$\begin{aligned} Brilho(R,G,B)' &= Componente(R,G,B) + Valor\ de\ Referência \\ Brilho(R,G,B) &= \begin{cases} 0 & , Brilho(R,G,B)' < 0 \\ 255 & , Brilho(R,G,B)' > 255 \\ Brilho(R,G,B) & , Brilho(R,G,B)' \leq 255 \wedge Brilho(R,G,B)' \geq 0 \end{cases} \end{aligned} \tag{3.9}$$

3.4.6 Binarização

O método de binarização desenvolvido tem como objetivo modificar o valor de um pixel em tons de cinza para preto ou branco. A sua implementação depende de uma referência predefinida, à qual, é feita uma comparação com o pixel em questão (equação 3.10). Caso a referência seja maior ou igual relativamente ao valor do pixel atual, o pixel resultante apresentará cor preta (zero), caso contrário, apresentará cor branca (255).

$$Pixel\ Binário = \begin{cases} 0 & , Valor\ do\ Pixel \leq Valor\ de\ Referência \\ 255 & , Valor\ do\ Pixel > Valor\ de\ Referência \end{cases} \tag{3.10}$$

Ao contrário dos filtros anteriores, o módulo de binarização recebe como entrada o tom de cinza de um pixel e apresenta como saída, o seu valor correspondente depois de filtrado (preto ou branco).

3.4.7 Sobel

O filtro *Sobel* desenvolvido deteta e realça contornos através de duas componentes (horizontal e vertical), auxiliado de uma máscara de convolução 3x3 e das equações respetivas (3.11, 3.12, 3.13). Na figura 3.14, estão representadas as posições da máscara utilizadas nas equações do método onde, "a" representa a intensidade do pixel presente no canto superior esquerdo.

$$Sx(R,G,B) = (a + 2 \times d + g) - (c + 2 \times f + i) \tag{3.11}$$

$$Sy(R,G,B) = (a + 2 \times b + c) - (g + 2 \times h + i) \tag{3.12}$$

$$S(R,G,B) = |Sx| + \left|Sy\right| \tag{3.13}$$

a	b	c
d	e	f
g	h	i

Figura 3.14: Representação gráfica das posições da máscara de convolução utilizadas nas equações do método *Sobel* ("e" representa o pixel a processar e "i" a posição que o projetista tem acesso durante o processamento)

3.4.8 Média Uniforme

Um dos filtros de remoção de ruído implementados foi a média uniforme. Neste método, cada pixel foi substituído pela média aritmética do seu valor e dos seus vizinhos, tendo

sido utilizada uma máscara unitária 3x3. Considerando a especificidade da linguagem descrita anteriormente, foi utilizada a equação 3.14 de modo a ser possível obter uma divisão por 9, na qual, M(i,j) representa o valor do pixel contido em cada posição da máscara de convolução.

$$Média(R,G,B) = \frac{\sum M(i,j) * 57}{512} \tag{3.14}$$

3.4.9 Gaussiano

O filtro gaussiano foi desenvolvido com o intuito de remover o ruído existente nas *frames* do vídeo captado. Para esse efeito, foi utilizada uma máscara de convolução com diferentes pesos por posição, representada na figura 3.15.

1	2	1
2	4	2
1	2	1

Figura 3.15: Representação gráfica dos pesos das posições da máscara de convolução utilizados na implementação do filtro gaussiano

De modo a implementar o método, foram multiplicados os pesos de cada posição da máscara pelos valores dos píxeis correspondentes (por componente), sendo de seguida, efetuada uma soma dos resultados, e por fim, uma divisão por 16.

3.4.10 Erosão

O filtro de erosão tem como objetivo diminuir o ruído em *frames* a preto e branco. Este método foi desenvolvido com o auxílio de uma máscara composta por zeros e de uma operação lógica *AND* entre os valores da máscara e os valores dos pixeis contidos nas suas posições. O algoritmo tem o seguinte funcionamento: caso o pixel a processar e os seus vizinhos tenham exatamente os mesmos valores que a máscara, o pixel resultante terá cor preta (zero), caso contrário, terá cor branca (255). Para uma melhor compreensão por parte do leitor, na figura 3.16 é apresentado um exemplo da aplicação deste filtro.

3.4.11 Dilatação

O último método a ser desenvolvido foi a operação morfológica dilatação que, tal como o filtro de erosão, é aplicada a *frames* a preto e branco. Também foi utilizada uma máscara composta por zeros de modo a ser possível implementar o método porém, foi utilizada a operação lógica *OR* em substituição da operação *AND* usada na implementação do filtro anterior. O seu funcionamento pode ser descrito da seguinte forma: se o valor do pixel a processar ou um dos seus vizinhos for igual ao valor correspondente na máscara, o pixel

resultante terá cor preta (zero), caso contrário, apresentará cor branca (255). Na figura 3.17 é apresentado um exemplo da aplicação deste filtro.

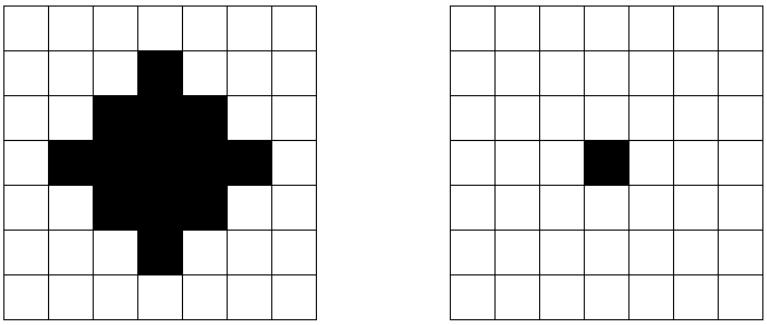


Figura 3.16: Exemplo da aplicação do filtro de erosão (à esquerda está representada a imagem original e à direita a imagem processada)

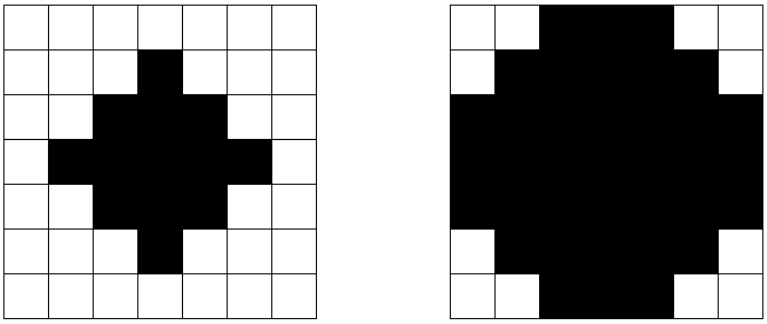


Figura 3.17: Exemplo da aplicação do filtro de dilatação (à esquerda está representada a imagem original e à direita a imagem processada)

3.5 Transmissão de Imagem

A partir do *demo* utilizado e dos métodos desenvolvidos, foi possível verificar visualmente os resultados do vídeo filtrado (HDMI) mas, com o intuito de os validar corretamente, assim como, efetuar a integração em aplicações reais, foi necessária a extração de *frames* de FPGA para CPU.

Foram analisadas duas abordagens: a utilização da UART (devido ao *demo* já apresentar esse componente integrado) e o envio de pacotes via *Ethernet* pois, a plataforma de desenvolvimento utilizada (*Zybo Z7-20*) apresenta um conector deste protocolo. Desde logo, foi escolhida a segunda solução devido à sua elevada rapidez de transmissão de dados.

Nesta secção, em 3.5.1 é descrito o protocolo utilizado na transmissão de informação (LwIP UDP) e em 3.5.2, o processo de envio de pacotes referentes aos valores dos píxeis de cada *frame* do vídeo.

3.5.1 LwIP UDP

O LwIP (Lightweight IP) [92] está associado aos protocolos de transmissão de dados *Transmission Control Protocol/ Internet Protocol* (TCP/IP) e User Datagram Protocol (UDP), sendo que uma das suas aplicações são os sistemas embutidos. É um produto de acesso aberto à comunidade científica (*open-source*), desenvolvido em linguagem C e apresenta como principal benefício o acesso aos protocolos de comunicação TCP/IP e UDP através da utilização de poucos recursos. De notar que, o LwIP é bastante utilizado por fabricantes de sistemas embutidos como: *Altera*, *Analog Devices*, *Xilinx* e *Honeywell*.

Relativamente ao UDP [93], é um protocolo de comunicação que tem como base o envio e receção de pacotes, sendo que, é caracterizado pela sua simplicidade, rapidez e por não apresentar sobrecarga de informação, devido a não possuir um sistema de recuperação de dados perdidos. Em contrapartida, não garante proteção da informação enviada, nem que a ordem de envio da mesma seja a ordem de receção. Um pacote UDP é composto por dois campos: cabeçalho e área de dados. O cabeçalho é composto por quatro sub-campos, o primeiro guarda o endereço da fonte de transmissão, o segundo contém o endereço do destino, o terceiro indica o comprimento máximo da mensagem a enviar e o quarto corresponde ao *checksum*. A área de dados é um vetor composto pela mensagem a transmitir. Este protocolo tem como principal aplicação a transmissão de vídeo e áudio, sendo assim, a opção escolhida para a implementação da transmissão de imagem via *Ethernet*.

3.5.2 Transmissão de Dados

De modo a ser possível enviar pacotes com a informação relativa às *frames* do vídeo filtrado, foi necessário realizar alterações à parte de *software* do projeto base utilizado. Como explicitado na secção 3.3.1.3, através do módulo VDMA, são armazenadas três *frames* na memória DDR, logo, foi necessário desenvolver uma aplicação que tenha acesso a essas posições de memória e as envie via *Ethernet*. Note-se que, foram previamente definidos um endereço de IP e um porto com o intuito de efetuar a ligação entre FPGA e CPU.

Foi utilizado o LwIP UDP, sendo que, a FPGA comporta-se como cliente e o CPU como servidor, ou seja, a informação proveniente da FPGA é recebida pelo CPU. Em primeiro lugar, foi utilizado um exemplo [94] deste protocolo que envia mensagens iguais da FPGA para CPU, sendo que, este foi modificado e integrado no *demo* base garantindo o envio de informação relativa às *frames* do vídeo filtrado. Na tabela 3.4, são apresentadas algumas funções do LwIP UDP utilizadas no desenvolvimento da transmissão de imagem.

Uma das limitações do protocolo UDP é o número de *bytes* que podem ser enviados por pacote, sendo que, no projeto utilizado são enviados 1440 *bytes*. Como cada *frame* apresenta uma resolução de 1920x1080 píxeis (6.220.800 *bytes*), a primeira abordagem teve como base a divisão de cada linha em quatro pacotes (1440 *bytes* cada um), ou seja, são enviados 7680 pacotes de forma a ter a informação relativa a uma *frame* RGB. Tendo

Tabela 3.4: Funções utilizadas no desenvolvimento da transmissão de imagem através de LwIP UDP

Função	**Descrição**
udp_new()	Inicialização de um bloco de controlo (PCB) do protocolo UDP
udp_connect()	Conexão do PCB UDP ao endereço remoto
udp_send()	Envio de dados
udp_remove()	Remoção do PCB UDP

em conta que os métodos desenvolvidos são de pré-processamento de imagem, ou seja, têm como objetivo retirar informação e realçar a parte pertinente da mesma, foi escolhido enviar *frames* em tons de cinza. Esta nova abordagem reduz o número de *bytes* total a enviar para 2.073.600, sendo que, cada linha foi dividida em 3 pacotes com 640 *bytes* cada (a informação de uma *frame* é representada por 3240 pacotes). Na figura 3.18, está representada a divisão efetuada de uma *frame* em tons de cinza, sendo possível visualizar a ordem dos pacotes e os píxeis de cada linha que representam. O primeiro pacote tem informação dos primeiros 640 píxeis da primeira linha, o segundo apresenta a informação dos 640 píxeis seguintes, e assim sucessivamente.

	0 639	640 1279	1280 1919
0	0	1	2
1	3	4	5
2	6	7	8
...	...	...	...
...	...	...	...
...	...	...	...
1077	3.231	3.232	3.233
1078	3.234	3.235	3.236
1079	3.237	3.238	3.239

Figura 3.18: Representação gráfica da divisão de uma *frame* em vários pacotes (3 pacotes de 640 *bytes* por cada linha)

Após este raciocínio, foi desenvolvido um método de acesso à memória DDR de modo a serem enviados os seus valores via *Ethernet*. No *demo*, a primeira *frame* está localizada entre as posições de memória 0x0A000000 e 0x0A5EEC00, logo, foi utilizado um apontador de modo a ter acesso aos valores dos píxeis presentes nas mesmas. De seguida, foi necessário configurar o vetor a enviar para CPU. Este apresenta 1440 *bytes* de resolução, sendo que, as duas primeiras posições indicam qual o número do pacote em questão (0 a 3239), as 640 posições seguintes são preenchidas com os valores dos píxeis a enviar e as seguintes não são preenchidas. De forma a serem transmitidos os valores corretos, a cada envio de um pacote o vetor é preenchido com nova informação referente aos próximos píxeis da *frame*, controlada pelo apontador da memória DDR. Na figura 3.19, está representado o vetor enviado sendo possível visualizar o seu conteúdo por posição.

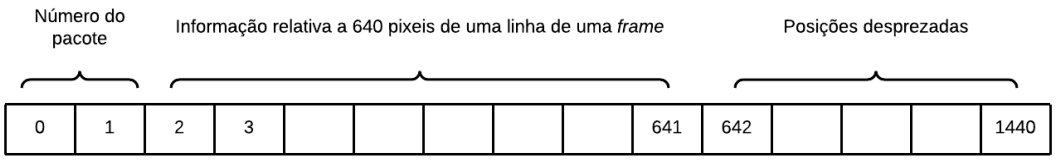


Figura 3.19: Representação gráfica do vetor enviado para CPU através do protocolo UDP

Por último, foi feita uma condição para não existir sobreposição de *frames*, ou seja, prevenir a leitura de informação de uma imagem armazenada enquanto o módulo VDMA está a escrever novos dados na mesma. Para esse efeito, as propriedades do VDMA foram modificadas de forma a este possuir 10 *frame buffers* em vez de 3, sendo assim possível, ler as posições de memória a enviar enquanto o VDMA escreve nas seguintes. Com este método foi garantido o envio correto de *frames* do vídeo filtrado para CPU.

3.6 Receção de Imagem

Depois de implementada uma aplicação que envia informação para CPU via *Ethernet*, foi desenvolvido um servidor UDP que recebe esses mesmos dados e armazena-os imediatamente em memória (imagem). Na figura 3.20, pode-se observar o raciocínio utilizado para o desenvolvimento do servidor.

Por análise da figura 3.20, é possível enumerar os passos necessários para a correta receção de uma *frame*: em primeiro lugar é criada uma imagem do tipo *MIplImage*, assim como, um apontador que é inicializado na primeira posição da mesma (canto superior esquerdo). Em segundo lugar, é inicializado o servidor com o mesmo endereço de IP e porto da aplicação implementada no processador *Zynq-7000*, permitindo assim, a ligação entre as duas plataformas. De seguida, o servidor inicia a receção de pacotes. Note-se que, como é utilizado o protocolo UDP estes podem não estar ordenados. De forma a resolver este problema, foi feita uma condição que ignora *frames* desordenadas. Quando o pacote correto é recebido, os seus valores são armazenados na imagem criada até a *frame* estar completa, sendo que, entre dois pacotes diferentes o apontador da imagem é sempre incrementado para a posição seguinte. No fim do processo, o apontador é reinicializado e inicia-se a receção de uma nova *frame*.

Para além da receção de *frames* em tempo real, também é possível a extração de apenas uma imagem. Foi utilizado o algoritmo apresentado na figura 3.20 mas, ao contrário da receção em tempo real, quando a primeira *frame* é recebida o processo é finalizado.

A receção de dados em CPU foi desenvolvida através de uma aplicação C#, na qual, foram utilizadas as bibliotecas *System.Net* e *Emgu*, de modo a efetuar o procedimento do fluxograma apresentado (figura 3.20). De forma a completar a aplicação, foi desenvolvida uma GUI (*Graphical User Interface*) com o objetivo de facilitar a receção de *frames* por

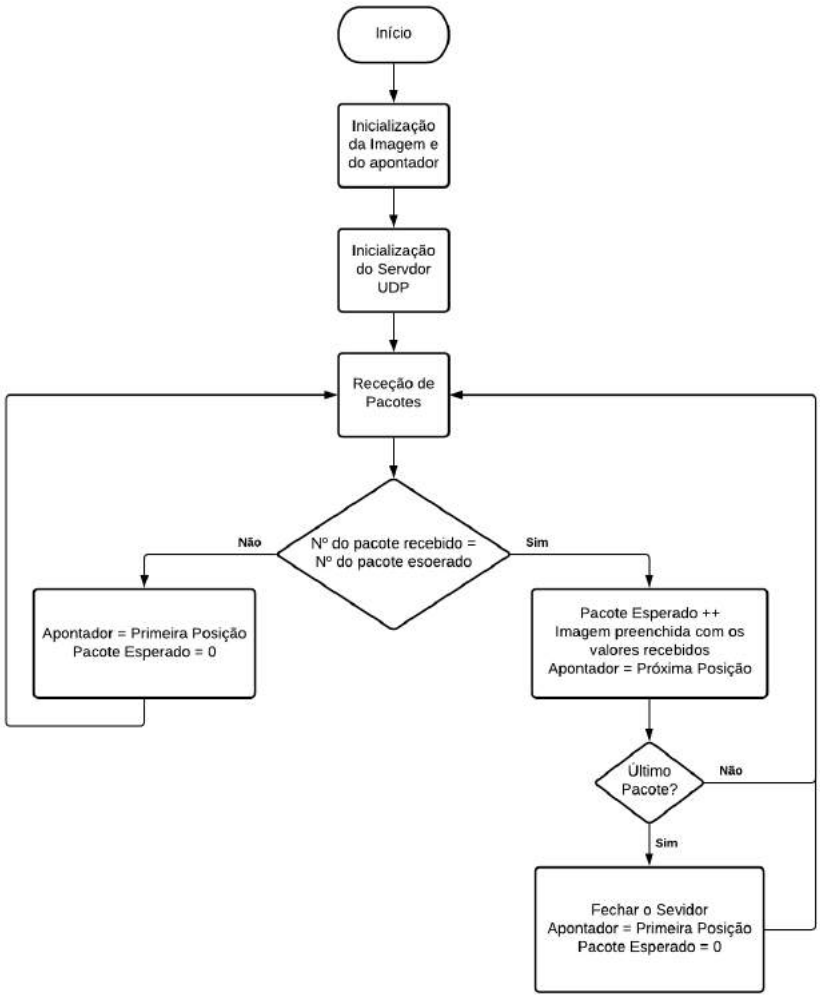


Figura 3.20: Fluxograma representativo do processo de receção de *frames* por parte do CPU

parte do utilizador, assim como, o processo de validação dos métodos. Na figura 3.21, é possível observar a GUI desenvolvida. Esta está dividida em duas partes: receção de *frames* provenientes da FPGA e o processo de validaçao.

Relativamente à de receção de *frames*, o primeiro campo é referente à conexão com o cliente UDP onde o utilizador pode inserir o endereço e a porta de comunicação correspondentes. De seguida, estão representados os campos de receção de *frames*, o primeiro onde o utilizador tem a possibilidade de iniciar e finalizar a receção de *frames* em tempo real e o segundo onde é recebida apenas uma *frame*. Note-se que é possível inserir o nome da imagem que será armazenada em memória e visualizar o estado da receção através do terminal.

A secção de validação tem na sua constituição um campo onde o utilizador pode comparar duas imagens e um campo de aplicação e medição do tempo de execução de um algoritmo. De forma a comparar duas imagens, o utilizador escolhe dois ficheiros

armazenados em memória sendo possível desprezar colunas e linhas. A aplicação dos filtros e medição de tempo é possível através da escolha do algoritmo e de um ficheiro e memória (imagem original). Note-se que os resultados são visíveis na subsecção "*Results*".

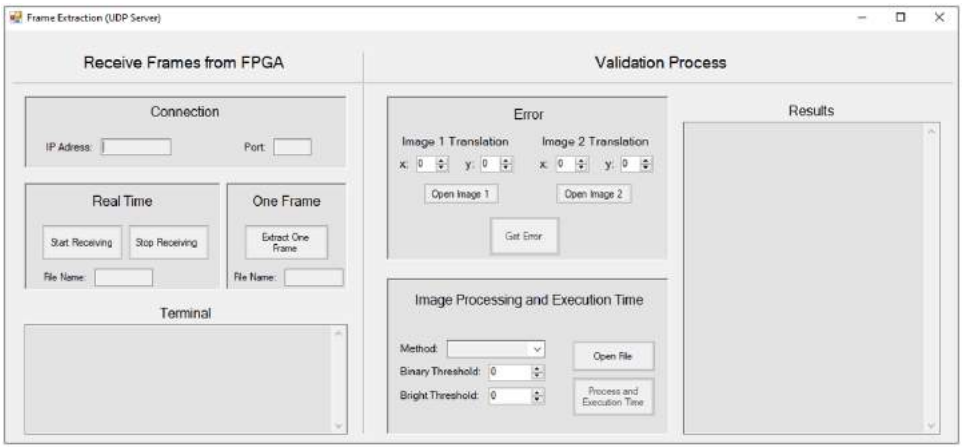


Figura 3.21: Aplicação C# desenvolvida (GUI)

Capítulo 4

Validação

Neste capítulo são apresentados os resultados da aplicação dos métodos desenvolvidos. Em primeiro lugar, é descrito o processo de validação utilizado (4.1), de seguida, em 4.2-4.11, são expostos os resultados obtidos para cada método (recursos utilizados, tempos de execução e comparação com o processamento em CPU). Posteriormente, é descrito o processo de integração em aplicações reais. Em 4.12 é apresentada a sequência de filtros utilizada, e em 4.13, são apresentados os resultados da aplicação do projeto num sistema de visão industrial. Por último, é apresentada uma secção de reflexão dos resultados obtidos (4.14).

4.1 Processo de Validação

Depois de desenvolvido um sistema com a capacidade de pré-processamento de imagem em FPGA, foi necessário averiguar a correta aplicação dos métodos implementados. Como o projeto base utilizado (*demo*) apresenta transmissão de vídeo via HDMI, o primeiro procedimento de validação foi a visualização do vídeo filtrado num monitor. Apesar de ser um método que permite a avaliação dos resultados (através da visualização das propriedades esperadas), não permite a confirmação de veracidade.

O problema anterior, motivou a modificação do procedimento de forma a garantir a correta validação dos filtros desenvolvidos. Este pode ser descrito da seguinte forma: em primeiro lugar, são extraídas e armazenadas duas *frames* em memória CPU (original e processada em FPGA). De seguida, é efetuada a aplicação do método em questão à imagem original, o armazenamento da *frame* resultante e a medição do tempo de processamento. Por último, são comparadas as duas imagens processadas, viabilizando a confirmação da veracidade dos filtros. O processo de medição do tempo de execução também pode ser realizado via GPU através do envio da imagem armazenada em memória CPU para GPU.

Na figura 4.1 é possível observar a arquitetura de validação dos filtros.

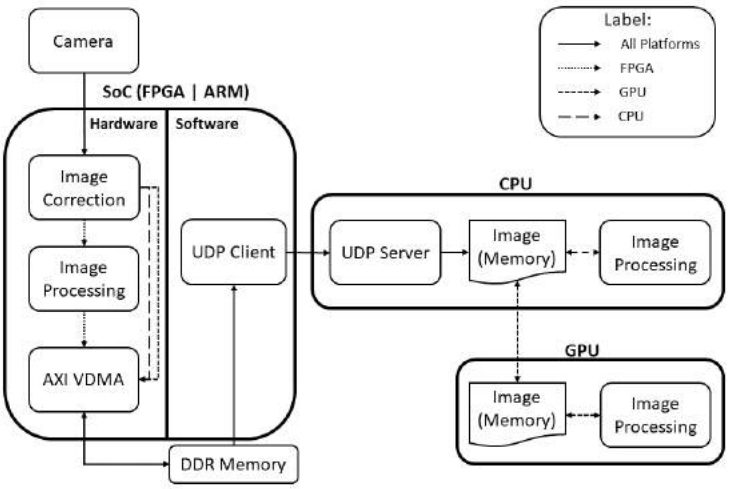


Figura 4.1: Diagrama representativo do processo de validação

Ao analisar a figura 4.1 é possível verificar que as três plataformas podem executar processamento de imagem. A FPGA recebe os dados captados pela câmara e envia-os para CPU (pixeis processados ou originais). Em CPU é feita a aplicação de filtros, medição de tempos de execução e comparação de imagens processadas, assim como, o envio da imagem para GPU de modo a ser possível executar algoritmos nesta plataforma e por conseguinte a medição dos tempos de processamento correspondentes.

Nesta secção é feita uma descrição pormenorizada do processo de validação, sendo que, em 4.1.1 é explicitado o envio e armazenamento de *frames* em memória, em 4.1.2 é descrito o algoritmo utilizado para o processamento da imagem original, assim como, a medição do tempo de processamento em CPU. Por último, é apresentado o algoritmo de comparação entre as imagens processadas em FPGA e CPU (4.1.3)

4.1.1 Transmissão de *Frames*

De modo a serem transmitidas *frames* para CPU com o intuito de validar os métodos desenvolvidos em FPGA, foram analisadas duas abordagens: a primeira baseia-se no envio de duas *frames* RGB, enquanto que, na segunda é feito o envio de uma *frame* com 3 canais em tons de cinzento composta por uma imagem original e uma imagem processada através dos métodos em FPGA .

O funcionamento da primeira abordagem é o seguinte: através do circuito presente no *demo* é feito o envio da *frame* original para CPU. Em segundo lugar, é modificado o circuito anterior através da ferramenta de *software Vivado 2016.4* com a finalidade de aplicar o método de pré-processamento de imagem a validar. Por último, é efetuado o

envio da *frame* processada, possibilitando assim, o acesso às duas imagens em memória CPU.

Relativamente à segunda abordagem, é criado um circuito onde o primeiro módulo de pré-processamento de imagem é a conversão RGB/Cinza e o segundo corresponde ao filtro a validar. Como o projeto base apresenta cada pixel representado por três componentes (RGB), é possível guardar e enviar informação relativa a três imagens com uma única componente (tons de cinza). Para este efeito, a imagem original (tons de cinzento) é guardada na componente vermelha de cada pixel e a imagem filtrada é armazenada na componente azul, permitindo assim, o envio de duas imagens relativas à mesma *frame* do vídeo para CPU (original e processada). Note-se que, a componente verde foi desprezada devido a não ser necessário aceder a uma terceira imagem. Na figura 4.2, é possível observar o fluxograma representativo das duas abordagens utilizadas.

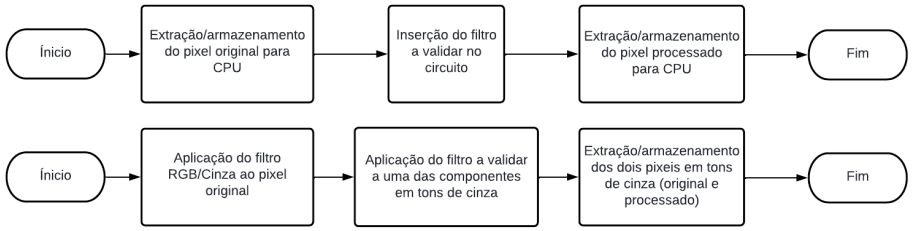


Figura 4.2: Representação das duas abordagens de envio de *frames* entre FPGA e CPU (em cima está representada a primeira abordagem e em baixo a segunda)

Por análise da figura 4.2, é possível concluir que a primeira abordagem tem como vantagem a validação de métodos que utilizam a imagem original em RGB (conversões RGB/Cinza e RGB/YCbCr) mas, apresenta uma desvantagem pois, ao ser efetuada a modificação do circuito, as condições de luminosidade ou a posição da câmara podem ser alteradas inesperadamente, logo, não é garantido que a imagem resultante tenha sido processada a partir da imagem original previamente enviada. Relativamente à segunda abordagem, apresenta como grande vantagem o facto de ser garantido que a aplicação do filtro é efetuada à *frame* original, devido às imagens serem enviadas em simultâneo. Por outro lado, apresenta a desvantagem de não ser possível validar métodos de pré-processamento em RGB. Em conclusão, as duas abordagens são possíveis de aplicar, sendo que, são esperados resultados mais precisos quando utilizada a segunda abordagem pois, as duas imagens são enviadas ao mesmo tempo. Os métodos conversão RGB/Cinza e conversão RGB/YCbCr foram validados a partir da primeira abordagem e os restantes oito pela segunda.

4.1.2 Aplicação e Medição do Tempo de Processamento do Método a Validar

Após recebidas as duas imagens (original e processada), através dos dois métodos explicitados na subsecção anterior, foi aplicado o filtro a validar à imagem original (implementado em CPU), assim como, a medição do tempo de processamento do mesmo, de forma a ser comparado com o tempo medido em FPGA.

Devido à aplicação do filtro em FPGA ser executada pixel a pixel, foi utilizada como primeira abordagem apenas a medição do tempo despendido durante a execução das equações referentes ao filtro. O algoritmo foi desenvolvido em C# sendo utilizada a classe *Stopwatch* de modo a serem executadas retomas e paragens da medição de tempo a cada posição da imagem. Como segunda abordagem foi medido o tempo total da *frame*, ou seja, foram utilizadas apenas uma vez cada função da classe *Stopwatch* (no início e no fim do método). Ao comparar as duas abordagens, era esperado que o tempo medido pela primeira fosse inferior ao tempo medido quando utilizada a segunda porém, os resultados demonstraram que o segundo método de medição era cerca de 20 vezes mais rápido. Devido a este facto, concluiu-se que o processador não é eficiente quando são invocadas enumeras vezes as funções de começo e paragem da classe *Stopwatch*, e por conseguinte, foi utilizada a segunda abordagem para a medição de tempos. Na figura 4.3 é visível o funcionamento dos dois procedimentos.

O algoritmo presente na figura 4.3 à esquerda (primeira abordagem) tem o seguinte funcionamento: em primeiro lugar, a imagem original é acedida e é criado um apontador inicializado na primeira posição da mesma (canto superior esquerdo). Em segundo lugar, é criado o contador de tempo, e de seguida, é averiguada a posição atual. Caso seja uma posição a processar, é iniciada a medição do tempo de processamento e é aplicado o filtro a partir das equações utilizadas na implementação VHDL, sendo de seguida, finalizada a medição do tempo e incrementada a posição da imagem. Caso contrário, a posição é incrementada. Por último, é verificada a condição que indica o final da *frame*, se esta for verdadeira, o procedimento acaba, se for falsa, é repetido o mesmo processo para o próximo pixel. Tendo em conta este raciocínio e o facto da ineficiência do processador quando utilizada a abordagem anterior, a solução utilizada (representada na figura 4.3 à direita), foi implementada através da medição do tempo no início e no final do método. Note-se que, quando a aplicação do filtro é finalizada, o tempo medido representa o período de processamento de uma *frame* mas, como o método é aplicado pixel a pixel, é pertinente ter a informação relativa ao período de processamento de cada posição. Para esse efeito, foi feita uma divisão do tempo resultante pelo número total de píxeis da imagem (equação 4.1). De notar que, foram efetuadas 30 medições por filtro de modo a serem obtidas conclusões mais precisas acerca do período de processamento.

$$Tempo\ de\ Processamento\ Médio\ por\ Pixel\ (CPU) = \frac{Tempo\ Total}{Número\ de\ Píxeis\ da\ Frame} \quad (4.1)$$

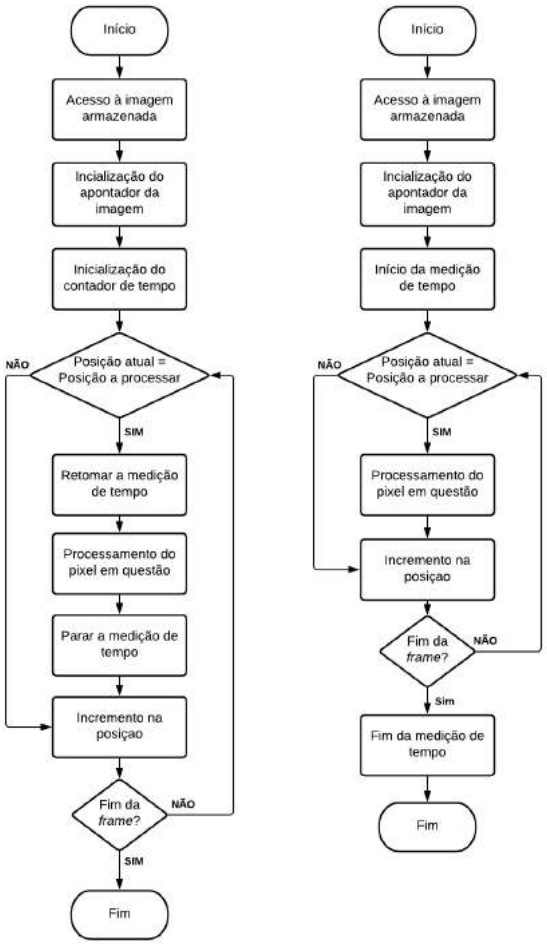


Figura 4.3: Fluxograma representativo da medição do tempo de processamento e aplicação de um filtro à *frame* original extraída para CPU (à esquerda está representada a primeira abordagem e à direita a segunda)

Por último, com o intuito de realizar uma comparação entre os dois tempos de processamento (CPU e FPGA), foi utilizada a equação 4.2 de forma a medir o período de processamento de um pixel em FPGA. De notar que, foi utilizada uma frequência de *clock* de 100 MHz.

$$Tempo\ de\ Processamento\ por\ Pixel\ (FPGA) = \frac{Número\ de\ Ciclos\ de\ Clock\ por\ Pixel}{Frequência\ de\ Clock} \tag{4.2}$$

4.1.3 Comparação de *Frames*

Uma das partes mais importantes na validação dos métodos desenvolvidos é a comparação entre os resultados obtidos através implementação dos métodos em FPGA e CPU. Para esse efeito, foi analisado o erro entre duas imagens pixel a pixel (uma processada em

FPGA e outra em CPU). Na figura 4.4, é apresentado o fluxograma representativo do funcionamento deste processo.

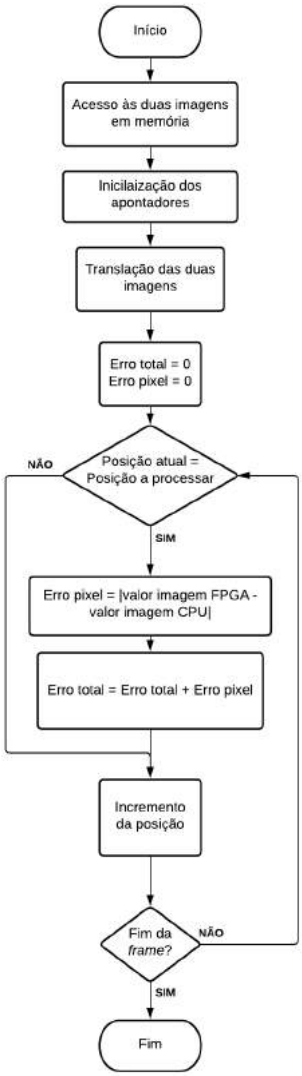


Figura 4.4: Fluxograma representativo do funcionamento do processo de comparação das duas *frames* processadas (CPU e FPGA)

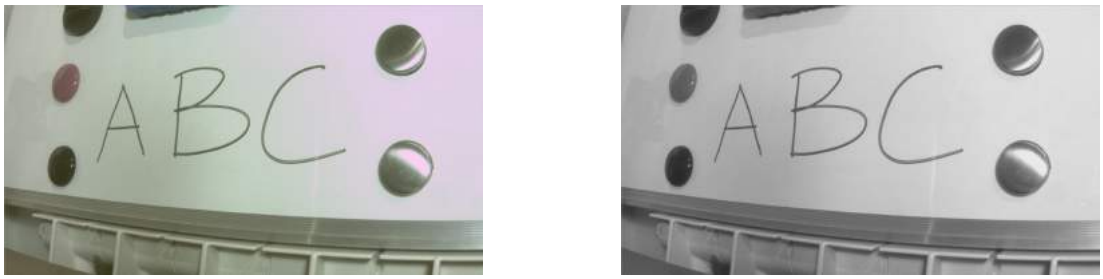
Por análise da figura 4.4, é possível verificar que os primeiros passos do procedimento de comparação de *frames* são o acesso às duas imagens armazenadas e a inicialização dos respetivos apontadores. De seguida, é aplicada a operação translação de forma a alinhar as mesmas, e por fim, são inicializadas as variáveis que quantificam o erro, uma que representa o erro de um determinado pixel e outra o erro total. De seguida é averiguada a posição atual das duas imagens, caso não corresponda a um pixel a processar, a posição é incrementada, caso contrário, é calculado o erro absoluto entre as duas *frames*, e por fim, este é adicionado à variável responsável pelo erro total. Posteriormente, a posição é incrementada e é verificada a condição que indica o fim da *frame*. Se a condição se verificar,

o processo chega ao fim, se não se verificar, o processo é efetuado para a posição seguinte. Como explicitado anteriormente, é pertinente ter a informação relativa à diferença entre cada pixel. Para esse efeito foi utilizada a equação 4.3, garantindo assim, o erro absoluto médio entre cada pixel das duas *frames* processadas.

$$Erro\ Absoluto\ Médio\ por\ Pixel = \frac{Erro\ Total}{Número\ de\ Píxeis\ da\ Frame} \tag{4.3}$$

4.2 Conversão RGB/Cinza

Com o intuito de validar o método de conversão RGB/Cinza, foi necessária a extração para CPU de uma *frame* RGB (imagem original) e uma *frame* em tons de cinza (imagem processada). Na figura 4.5 são apresentadas as imagens utilizadas para a validação do filtro.



(a) Imagem original (b) Imagem processada em FPGA

Figura 4.5: Resultados obtidos da aplicação do método conversão RGB/Cinza

Depois de extraídas, foi calculada a diferença entre o processamento em FPGA e CPU (comparação de *frames*). O erro resultante, isto é, a diferença média entre a intensidade dos píxeis processados em CPU e FPGA, foi de 3,102 unidades por pixel, ou seja, o valor de cada pixel processado em FPGA apresenta um desvio médio de 3,102 unidades relativamente ao valor processado em CPU, correspondente a 1,21% (3,102/256). De notar que, tal como em todas as secções seguintes referentes à validação de métodos, a imagem processada em CPU não é apresentada devido a exibir bastantes semelhanças com a *frame* processada em FPGA.

Relativamente aos recursos utilizados pela FPGA, na tabela 4.1 é visível a quantidade de elementos e a percentagem em relação à totalidade dos mesmos.

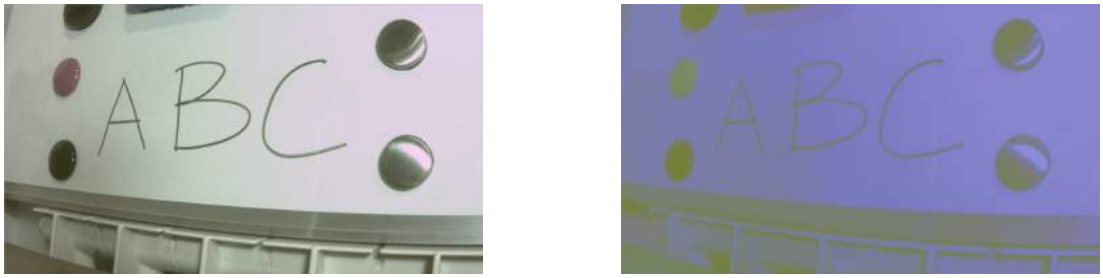
Por último, foram medidos os tempos da aplicação do filtro em FPGA e CPU. Em FPGA, cada pixel é processado num único ciclo de *clock* logo, o tempo calculado foi de 10,00 ns (equação 4.2). Relativamente ao CPU, foram medidas 30 amostras nas quais, o tempo médio por pixel equivale a 6,69 ns e o tempo mais rápido, 5,30 ns.

Tabela 4.1: Recursos utilizados pelo método conversão RGB/Cinza

	Utilização	
Recurso	**Unidades**	**Total (%)**
LUT	15	0,03
LUTRAM	0	0,00
FF	4	0,01
BRAM	0	0,00

4.3 Conversão RGB/YCbCr

O método de conversão RGB/YCbCr foi validado da mesma forma que o filtro de conversão RGB/Cinza, ou seja, através da extração da *frame* original RGB e da *frame* processada. As duas imagens são apresentadas na figura 4.6.



(a) Imagem original

(b) Imagem processada em FPGA

Figura 4.6: Resultados obtidos da aplicação do método conversão RGB/YCbCr

Após a extração, foi aplicado o mesmo filtro (desenvolvido em CPU) à imagem original, sendo de seguida, medida a diferença entre *frames*. O erro médio por pixel calculado foi de 6,709 unidades (2,62%). Em relação aos recursos utilizados pela FPGA, os resultados obtidos são apresentados na tabela 4.2.

Tabela 4.2: Recursos utilizados pelo método conversão RGB/YCbCr

	Utilização	
Recurso	**Unidades**	**Total (%)**
LUT	145	0,27
LUTRAM	0	0,00
FF	28	0,03
BRAM	0	0,00

De forma a processar um pixel em FPGA, este método necessita de um único ciclo de *clock* logo, o tempo de processamento é de 10,00 ns. Tendo em conta a implementação em

CPU, o tempo médio de processamento foi de 9,50 ns e o tempo mais rápido foi de 9,16 ns.

4.4 Negativo

De forma a validar o método negativo, tal como os métodos apresentados nas secções seguintes, foram extraídas para CPU duas imagens relativas à mesma *frame* em tons de cinza (original e processada). Na figura 4.7 estão representadas as duas imagens.

(a) Imagem original (b) Imagem processada em FPGA

Figura 4.7: Resultados obtidos da aplicação do método negativo

De seguida, foi aplicado o filtro desenvolvido em CPU à imagem original, e por conseguinte, a medição do erro entre *frames* processadas em FPGA e CPU. O erro médio por pixel calculado foi zero, ou seja, as *frames* comparadas são exatamente iguais. Outro parâmetro pertinente são os recursos utilizados pelo método (tabela 4.3).

Tabela 4.3: Recursos utilizados pelo método negativo

	Utilização	
Recurso	**Unidades**	**Total (%)**
LUT	26	0,05
LUTRAM	0	0,00
FF	27	0,03
BRAM	0	0,00

Relativamente ao período de processamento de um pixel, a implementação em FPGA necessita de um único ciclo de *clock*, ou seja, 10,00 ns. Em CPU, o tempo médio por pixel das amostras medidas apresenta o valor de 3,60 ns e o tempo mais rápido equivale a 3,38 ns.

4.5 Modificação de Brilho

Os resultados obtidos da aplicação do método de modificação de brilho a uma *frame* em tons de cinza podem ser observados na figura 4.8. Note-se que o valor de referência

utilizado equivale a -70.

(a) Imagem original (b) Imagem processada em FPGA

Figura 4.8: Resultados obtidos da aplicação do método de modificação de brilho

Após a extração das duas imagens, foi efetuado o processamento da *frame* original em CPU, e de seguida, foram comparadas as duas imagens filtradas. O erro calculado apresenta valor nulo, o que significa que as duas *frames* são idênticas. Relativamente aos recursos utilizados, estes são apresentados na tabela 4.4.

Tabela 4.4: Recursos utilizados pelo método de modificação de brilho

	Utilização	
Recurso	**Unidades**	**Total (%)**
LUT	29	0,058
LUTRAM	0	0,00
FF	28	0,03
BRAM	0	0,00

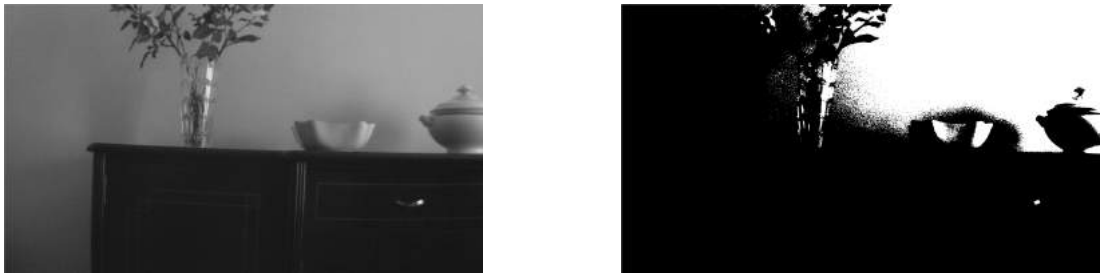
Por último, foram medidos os tempos de processamento nas duas plataformas (FPGA e CPU). A implementação em FPGA necessita de um ciclo de *clock* para o processamento de um pixel (10,00 ns), enquanto que, na implementação em CPU o tempo médio por pixel equivale a 8,31 ns e o mais rápido, 7,72 ns.

4.6 Binarização

Os resultados obtidos na validação do método de binarização com um valor de referência de 127 são apresentados na figura 4.9.

Depois de processada a *frame* original pelo método desenvolvido em CPU, o erro calculado foi nulo, ou seja, as imagens comparadas são iguais. Relativamente aos recursos utilizados, os resultados são apresentados na tabela 4.5.

Por último, foi medido o tempo de processamento de um pixel nas duas plataformas. A implementação em FPGA necessita de um ciclo de *clock* de modo a aplicar o filtro logo, o tempo de processamento equivale a 10,00 ns. Em CPU, o tempo médio de processamento medido equivale a 2,59 ns e o mais rápido corresponde a 2,41 ns.



(a) Imagem original (b) Imagem processada em FPGA

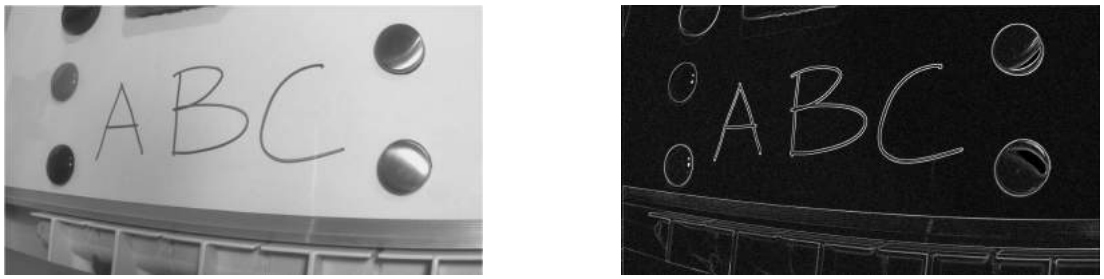
Figura 4.9: Resultados obtidos da aplicação do método de binarização

Tabela 4.5: Recursos utilizados pelo método de binarização

	Utilização	
Recurso	**Unidades**	**Total (%)**
LUT	2	0,01
LUTRAM	0	0,00
FF	27	0,03
BRAM	0	0,00

4.7 Sobel

Os resultados obtidos na validação do método *Sobel* são visíveis na figura 4.10.



(a) Imagem original (b) Imagem processada em FPGA

Figura 4.10: Resultados obtidos da aplicação do método *Sobel*

Após as duas imagens serem extraídas, foi aplicado o método *Sobel* desenvolvido em CPU à imagem original e as duas *frames* processadas foram comparadas (FPGA e CPU). De notar que, no desenvolvimento deste filtro não foram processadas as duas primeiras e duas últimas colunas, assim como, as duas primeiras linhas. Devido a este facto, e de modo a ser garantido o menor erro possível na comparação de *frames*, foram desprezadas as regiões não processadas, as seis colunas e as duas linhas seguintes. Tendo em conta a região a comparar, o erro médio foi de 0,021 unidades por pixel (0,0082%). Relativamente aos recursos utilizados pelo filtro, os resultados são visíveis na tabela 4.6.

Tabela 4.6: Recursos utilizados pelo método *Sobel*

	Utilização	
Recurso	**Unidades**	**Total (%)**
LUT	3052	5,74
LUTRAM	1920	11,03
FF	547	0,51
BRAM	4,5	3,21

Por último, foi medido o período de processamento de um pixel em FPGA e CPU. Em FPGA, de modo a ser processado um pixel, são necessários seis ciclos de *clock* logo, 60,00 ns por pixel. Tendo em conta as medições em CPU, o tempo médio por pixel equivale a 48,80 ns e o mais rápido corresponde a 38,10 ns. Note-se que, como é necessário mais que um ciclo de *clock* para processar cada posição da imagem, os píxeis processados são colocados na imagem resultante com um desfasamento horizontal correspondente ao número de *clocks* do algoritmo. Este efeito é visível neste método, no filtro de média uniforme (4.8) e gaussiano (4.9).

4.8 Média Uniforme

O primeiro método de remoção de ruído validado foi a média uniforme. O resultado da aplicação do mesmo é apresentado na figura 4.11.

(a) Imagem original (b) Imagem processada em FPGA

Figura 4.11: Resultados obtidos da aplicação do método média uniforme

O erro resultante entre a *frame* processada em FPGA e em CPU foi de 0,01 unidades por pixel (0,0039%) sendo que, não foram consideradas as regiões não processadas, a linha e as duas colunas seguintes. Em termos de recursos utilizados, na tabela 4.7 é possível observar a quantidade e a percentagem de utilização dos mesmos.

Relativamente ao tempo de processamento de um pixel através deste método, em FPGA, são necessários 4 ciclos de *clock* logo, 40,00 ns. Em CPU, as medições efetuadas mostram que o tempo médio por pixel é de 40,35 ns e o tempo mais rápido equivale a 39,06 ns.

Tabela 4.7: Recursos utilizados pelo método média uniforme

	Utilização	
Recurso	**Unidades**	**Total (%)**
LUT	4087	7,68
LUTRAM	2880	16,55
FF	336	0,32
BRAM	4,5	3,21

4.9 Gaussiano

O segundo método de remoção de ruído desenvolvido foi o filtro gaussiano e foi validado através das imagens presentes na figura 4.12.

(a) Imagem original

(b) Imagem processada em FPGA

Figura 4.12: Resultados obtidos da aplicação do método gaussiano

Depois de extraídas as *frames* em tons de cinza (original e processada em FPGA), foi aplicado o método desenvolvido em CPU à *frame* original, e por conseguinte, foram comparadas as duas imagens processadas. O erro calculado foi de 0,02 unidades por pixel (0,0078%), sendo que, não foram consideradas as regiões não processadas, a linha e as três colunas seguintes de forma a ser feita uma comparação mais precisa. Os recursos utilizados pelo método são apresentados na tabela 4.8.

Tabela 4.8: Recursos utilizados pelo método gaussiano

	Utilização	
Recurso	**Unidades**	**Total (%)**
LUT	4205	7,90
LUTRAM	2880	16,55
FF	367	0,34
BRAM	4,5	3,21

Em relação ao tempo de processamento, em FPGA, a implementação do filtro foi efetuada através de três ciclos de *clock*, ou seja, 30,00 ns por pixel. Em CPU, o tempo

médio por pixel corresponde a 26,43 ns e o mais rápido, 25,56 ns.

4.10 Erosão

De modo a validar o método de erosão foram extraídas duas imagens a preto e branco relativas à mesma *frame* (original e processada em FPGA). Na figura 4.13, é possível observar o resultado da aplicação do método.

(a) Imagem original (b) Imagem processada em FPGA

Figura 4.13: Resultados obtidos da aplicação do método erosão

Após a aplicação do filtro em CPU à imagem original, foi efetuada uma comparação das duas imagens processadas nas duas plataformas (FPGA e CPU). O erro resultante foi de 0,00049 unidades por pixel (0,00019%). Outro parâmetro importante são os recursos utilizados pelo método, sendo que, estes são apresentados na tabela 4.9.

Tabela 4.9: Recursos utilizados pelo método erosão

	Utilização	
Recurso	**Unidades**	**Total (%)**
LUT	2951	5,55
LUTRAM	2160	12,41
FF	168	0,16
BRAM	0	0,00

Por último, foi medido o tempo de processamento de um pixel através da aplicação do método. Em FPGA, foi desenvolvido um algoritmo que necessita de um único ciclo de *clock* de forma a aplicar o filtro, ou seja, 10,00 ns por pixel. Em CPU, foi medido o tempo médio de processamento (5,34 ns) e o tempo mais rápido (4,82 ns).

4.11 Dilatação

O último método a ser validado foi o algoritmo de dilatação, tal como o filtro de erosão, a sua validação teve como base a extração de duas imagens a preto e branco para CPU (figura 4.14).

(a) Imagem original (b) Imagem processada em FPGA

Figura 4.14: Resultados obtidos da aplicação do método dilatação

Após a extração, foi aplicado o filtro desenvolvido em CPU de modo a ser possível efetuar a comparação entre duas imagens (uma processada em FPGA e outra em CPU). O erro resultante médio por pixel foi de 0,0027 unidades (0,0011%). Relativamente aos recursos utilizados, os resultados obtidos são apresentados na tabela 4.10.

Tabela 4.10: Recursos utilizados pelo método dilatação

	Utilização	
Recurso	**Unidades**	**Total (%)**
LUT	2972	5,59
LUTRAM	2160	12,41
FF	154	0,16
BRAM	0	0,00

Por último, foram medidos os tempos de processamento em FPGA e CPU, sendo que, em FPGA o algoritmo precisa de um único de *clock* para processar um pixel, o que equivale a 10,00 ns. Em CPU, das amostras medidas, o tempo médio por pixel corresponde a 8,91 ns e o tempo mais rápido, 8,68 ns.

4.12 Sequência de Filtros

Depois de concluída a validação individual dos filtros implementados em FPGA, foi desenvolvida uma sequência de filtros com o intuito de integrar o projeto num sistema de visão industrial. O objetivo proposto foi a deteção e realce de contornos, logo, foi implementado um circuito que apresenta na sua constituição os filtros *Sobel*, RGB/Cinza, binarização e negativo de forma sequencial. O primeiro método tem a função de realçar os contornos das *frames* captadas pela câmara, os dois módulos seguintes retiram informação da mesma ao converter o formato RGB para preto e branco (contornos realçados a branco), e por fim, de modo aos contornos serem visíveis a preto, foi utilizado o filtro negativo. Note-se que, o filtro de binarização foi aplicado com um valor de referência de 127.

Após a implementação e execução do circuito, verificou-se não ser possível transmitir *frames* para CPU. Foi feita uma análise ao relatório dos tempos disponibilizado pela ferramenta, no qual, foi verificado que não eram assegurados os requisitos do sinal de *clock* do protocolo *AXI4-Stream*. Este problema motivou à redução da frequência de *clock* de 150 MHz para 100 MHz de forma a garantir o funcionamento do circuito. Ao utilizar uma frequência mais baixa, o tempo entre eventos de *clock* é mais elevado logo, o tempo para o processamento de um pixel aumenta. Nas figuras 4.15 e 4.16 estão representados os relatórios de tempos do circuito quando utilizadas frequências de 150 MHz e 100 MHz respetivamente.

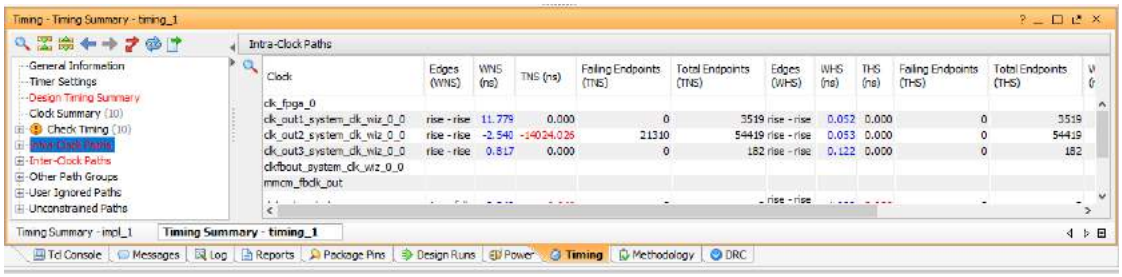


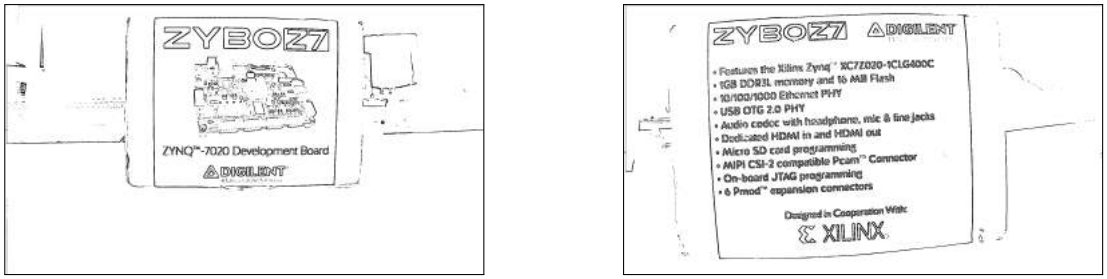
Figura 4.15: Relatório de tempos da sequência de filtros para uma frequência de *clock* de 150 MHz

Figura 4.16: Relatório de tempos da sequência de filtros para uma frequência de *clock* de 100 MHz

Na figura 4.15, é possível verificar que o valor negativo a vermelho na coluna correspondente ao *Total Negative Slack* (TNS) impossibilita o bom funcionamento do circuito pois, o sinal respetivo (*clk_out2_system_clk_wiz_0_0*) apresenta um atraso significativo (14.024,026 ns). Relativamente à frequência de 100 MHz (figura 4.16), apesar de também apresentar um atraso (0,254 ns), este é bastante mais baixo possibilitando o funcionamento do circuito. A situação ideal seria obtida caso não existissem atrasos nas variáveis relativas aos diferentes sinais de *clock*, porém, como o circuito funciona nestas condições e os filtros podem ser futuramente otimizados, foram efetuados os testes com a frequência de 100 MHz. Outro dado importante a considerar são os pontos de falha. Quando utilizada uma frequência de 150 MHz o seu valor equivale a 21.310 mas, ao utilizar 100 MHz existem apenas 6, num total de 54.419. De notar que, o sinal de *clock* que apresenta valores negativos é correspondente ao protocolo *AXI4-Stream*.

Outro problema encontrado foi a sincronização do sinal TREADY do protocolo *AXI4-Stream*. De forma a ser possível implementar a sequência de quatro métodos foi necessário configurar o sinal TREADY de cada módulo como assíncrono. Esta abordagem possibilita que todos os módulos da sequência sejam notificados ao mesmo tempo que o último *slave* está pronto para receber informação. De notar que, este sinal só necessita de configuração assíncrona quando a sequência de filtros apresenta mais que 2 *clocks* de atraso, caso contrário, pode ser síncrono.

Antes de integrar a referida sequência em aplicações reais foi necessário realizar testes de funcionamento. Pretendia-se o realce de contornos das faces da caixa da placa de desenvolvimento utilizada (*Zybo Z7-20*), sendo que, os resultados estão presentes na figura 4.17.



(a) Imagem extraída da face frontal da caixa (b) Imagem extraída da face posterior da caixa

Figura 4.17: Resultados obtidos da aplicação da sequência de filtros

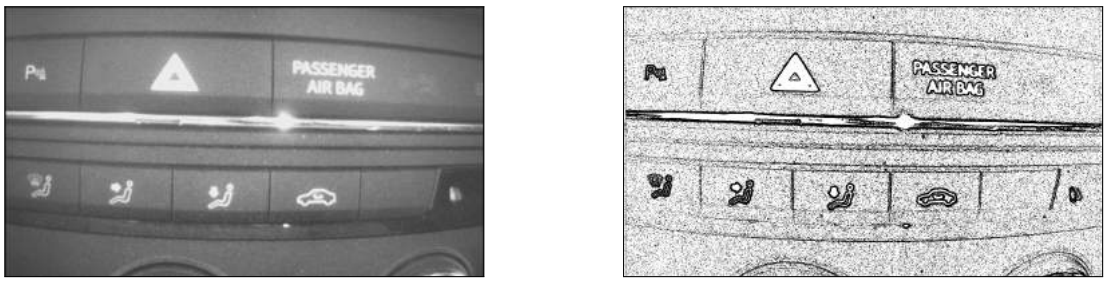
Ao analisar a figura 4.17 é possível concluir que os resultados obtidos foram os esperados, sendo visível em ambas as imagens os contornos referentes às letras e limites das faces da caixa.

4.13 Integração em Aplicações Reais

Após ser definida uma sequência de métodos de modo a ser possível identificar contornos, foi realizada a integração no projeto CheckMate da empresa Introsys. Este está inserido na área do controlo de qualidade na industria automóvel e tem como finalidade analisar características e realizar testes estruturais após a montagem final do produto, com recurso a uma estação móvel.

A integração do presente sistema no projeto CheckMate pretende auxiliar a identificação de defeitos entre os botões presentes no tabliê de um carro através de métodos de pré-processamento de imagem (realce de contornos). Posteriormente, é feita a extração de características com a respectiva detecção de anomalias. Uma vez que o pré-processamento é o foco desta dissertação, para ser verificada a correta integração no projeto CheckMate foram realizados dois testes: confirmação da identificação dos contornos relativos ao espaço entre botões através de pré-processamento em FPGA e comparação dos tempos de processamento entre implementações em CPU e FPGA.

O primeiro teste foi realizado através da captação da cena em questão seguida da extração das imagens para um computador (original e processada em FPGA). Note-se que, o valor de referência do filtro de binarização foi modificado de 127 para 40 de forma aos contornos serem visíveis. Na figura 4.18 estão representadas as imagens extraídas onde são visíveis os contornos entre botões do tabliê do carro (imagem à direita).



(a) Imagem original da cena em questão (b) Imagem processada em FPGA

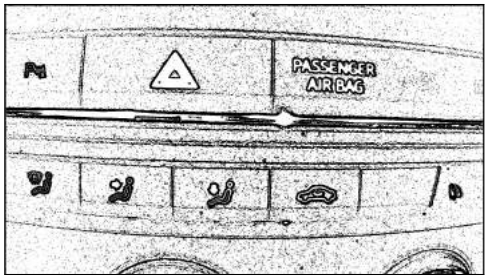
Figura 4.18: Resultados obtidos da integração do presente projeto em aplicações reais

Para o segundo teste foram comparados os tempos de execução de 4 soluções diferentes de pré-processamento:

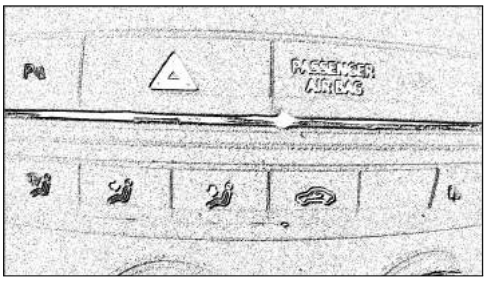
1. FPGA
2. Ferramenta *Halcon* [95] (CPU)
3. Biblioteca OpenCV (CPU)
4. Implementação direta em C# dos algoritmos implementados em VHDL (CPU)

Relativamente à ferramenta *Halcon*, foi efetuada a seguinte sequência de métodos: *invert_image*, *sobel_amp* e *threshold*. Através da biblioteca OpenCV, foram utilizadas sequencialmente as seguintes funções: *Sobel*, *ThresholdBinary* e *Not*. Por fim, foram utilizados os métodos desenvolvidos em C#. Na figura 4.19 são apresentados os resultados obtidos quando processada a imagem original através da ferramenta *Halcon*, assim como, através das funções OpenCV. Note-se que, o valor de referência utilizado na ferramenta *Halcon* equivale a 20 pois, com uma referência de 40 os contornos não são visíveis. A imagem resultante da aplicação da sequência utilizada em FPGA não é apresentada pois é idêntica à figura 4.18 (imagem à direita).

Por fim, foi medido o tempo de processamento das 4 implementações, sendo que, em FPGA o processamento foi efetuado em tempo real, ou seja, à medida que os píxeis são recebidos é aplicada a sequência de filtros, enquanto que em CPU, foi efetuado o processamento de uma imagem armazenada em memória. Os resultados obtidos são apresentados na tabela 4.11.



(a) Imagem processada através da ferramenta Halcon



(b) Imagem processada através da biblioteca OpenCV

Figura 4.19: Resultados obtidos do processamento da imagem original através da ferramenta Halcon e das funções da biblioteca OpenCV

Tabela 4.11: Tempos de processamento da sequência de filtros em FPGA e CPU

Plataforma	**Tempo de Processamento por Pixel (ns)**	**Tempo de Processamento por Frame (ms)**
FPGA	80	-
Halcon (CPU)	14,5828	30,2389
OpenCV (CPU)	17,9598	37,2415
Equações utilizadas em FPGA (CPU)	55,6109	115,3149

4.14 Discussão de Resultados

O processo de validação dos métodos desenvolvidos baseou-se em três parâmetros: erro absoluto entre implementação em FPGA e CPU, tempo de processamento e recursos utilizados. Na tabela 4.12, é possível observar os dados obtidos relativamente ao erro absoluto e tempos de processamento medidos.

Tabela 4.12: Resultados obtidos no processo de validação dos métodos (erro absoluto e tempos de processamento)

		FPGA		**CPU**			
		Tempo de Atraso (ns)	**Número de Ciclos de Clock (Unidades)**	**Tempo por Pixel (ns)**		**Tempo por Frame (ms)**	
Métodos	**Erro Absoluto (Unidades)**			**Média**	**Mais Rápido**	**Média**	**Mais Rápido**
RGB/cinza	3,102	10	1	6,69	5,30	13,87	10,99
RGB/YCbCr	6,709	10	1	9,50	9,16	19,70	18,99
Negativo	0	10	1	3,60	3,38	7,46	7,01
Modificação de Brilho	0	10	1	8,31	7,72	17,23	16,01
Binarização	0	10	1	2,59	2,41	5,37	5,00
Sobel	0,021	60	6	48,80	38,10	101,19	79,00
Média Uniforme	0,01	40	4	40,35	39,06	83,67	80,99
Gaussiano	0,02	30	3	26,43	25,56	54,81	53,00
Erosão	0,00049	10	1	5,34	4,82	11,07	9,99
Dilatação	0,0027	10	1	8,91	8,68	18,48	18,00

Por análise da tabela 4.12, é possível verificar que os métodos validados através da

primeira abordagem de extração de imagens para CPU (explicitada em 4.1.1) apresentam um **erro absoluto** bastante mais significativo comparativamente aos filtros validados pela segunda abordagem. Tal acontece porque as duas imagens não correspondem à mesma *frame*, assim, podem ocorrer alterações repentinas das condições de luminosidade ou posição da câmara, o que impossibilita a obtenção de resultados precisos. Seria espectável, nos métodos validados pela segunda abordagem, que a diferença entre píxeis fosse nula, uma vez que se utilizaram as mesmas equações em FPGA e CPU. Esta suposição não se sucedeu porém, os resultados obtidos apresentam valores de erro nulo ou bastante próximos de zero, sendo que, o valor mais elevado equivale a 0,021 unidades (método *Sobel*). O erro não nulo entre implementações pode vir a ser corrigido com a otimização dos métodos. Através dos valores apresentados, é possível confirmar veracidade de todos os algoritmos desenvolvidos em FPGA.

Relativamente ao **tempo de processamento**, em FPGA, os resultados são referentes à aplicação dos métodos em tempo real, ou seja, à medida que os píxeis são recebidos, é efetuado o seu processamento. Por outro lado, em CPU o processamento foi realizado através da aplicação do filtro em questão a uma imagem armazenada em memória. Na figura 4.20 estão representados os dois processos de aplicação dos métodos nas duas plataformas, sendo visível que em FPGA o bloco de pré-processamento recebe e retorna um pixel de cada vez, enquanto que, em CPU o processo é executado através da receção e processamento de uma imagem.

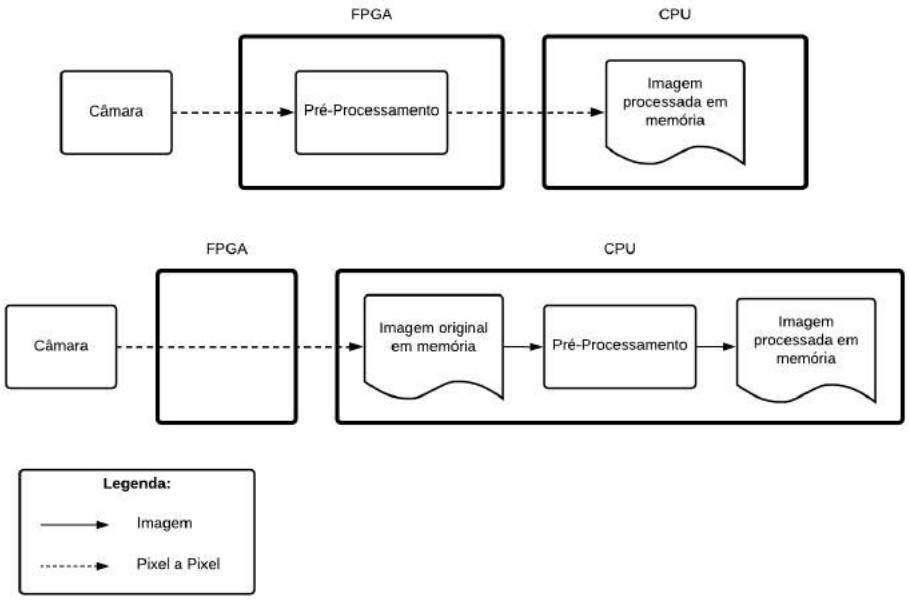


Figura 4.20: Método de aplicação dos filtros de pré-processamento em FPGA e CPU

Por análise dos resultados da tabela 4.12, ao comparar os tempos por pixel das duas plataformas, é possível concluir que em CPU foram obtidos melhores resultados excepto

no tempo médio de processamento do filtro média uniforme. Contudo, tendo em os tempos por *frame* e considerando que em FPGA a aplicação dos filtros é realizada em tempo real, é possível afirmar que esta solução atrasa o processamento de uma *frame* na mesma ordem que atrasa um pixel. Considerando o exemplo do método RGB/Cinza, de modo a ser efetuada a aplicação do filtro, cada pixel sofre um atraso de 1 ciclo de *clock* (10 ns), logo, a *frame* processada na sua totalidade também será atrasada apenas 1 ciclo de *clock*, devido ao processamento ser efetuado logo após a receção dos píxeis. Tendo isto em conta, ao comparar os resultados referentes ao processamento de uma *frame* nas duas plataformas, em FPGA, foram obtidos valores na ordem dos nanosegundos, sendo significativamente melhores comparativamente aos milissegundos obtidos pelo CPU. Assim, considerando o processamento em tempo real, em FPGA são atingidos melhores resultados visto que o período de atraso é menor comparativamente ao período de processamento de uma imagem em CPU.

Como último critério de validação dos métodos analisou-se a **quantidade de recursos utilizados** em FPGA (unidades e percentagem). Na tabela 4.13 são visíveis os resultados obtidos para todos os filtros.

Tabela 4.13: Resultados obtidos no processo de validação dos métodos relativos à quantidade de recursos utilizados

	Recursos Utilizados (Unidades)				**Recursos Utilizados (%)**			
Métodos	**LUT**	**LUTRAM**	**FF**	**BRAM**	**LUT**	**LUTRAM**	**FF**	**BRAM**
RGB/Cinza	15	0	4	0	0,03	0	0,01	0
RGB/YCbCr	145	0	28	0	0,27	0	0,03	0
Negativo	26	0	27	0	0,05	0	0,03	0
Modificação de Brilho	29	0	28	0	0,058	0	0,03	0
Binarização	2	0	27	0	0,01	0	0,03	0
Sobel	3052	1920	547	4,5	5,74	11,03	0,51	3,21
Média Uniforme	4087	2880	336	4,5	7,68	16,55	0,32	3,21
Gaussiano	4205	2880	367	4,5	7,90	16,55	0,34	3,21
Erosão	2951	2160	168	0	5,55	12,41	0,16	0
Dilatação	2972	2160	154	0	5,59	21,41	0,16	0

Por análise da tabela 4.13, é visível que os filtros que não utilizam a máscara de convolução apresentam valores de LUT e FF mais baixos, e não necessitam de LUTRAM ou BRAM. Os métodos que utilizam máscara de convolução, devido à sua elevada complexidade, apresentam maior utilização de recursos, sendo que, apenas os que necessitam de mais de um ciclo de *clock* para processar um pixel utilizam BRAM. O método que utiliza mais recursos é o filtro gaussiano, sendo que, apresenta 4205 LUT, 2880 LUTRAM e 4,5 BRAM. Destacar também o filtro *Sobel* que atinge o máximo de FF (547 unidades). Por outro lado, os métodos menos exigentes são RGB/Cinza (4 FF) e a binarização (2 LUT). Em termos percentuais, todos os métodos apresentam resultados baixos, sendo o

máximo atingido no componente LUTRAM nos filtros gaussiano e média uniforme (16,55 %). Através destes resultados, é possível concluir que a utilização de recursos por parte dos algoritmos desenvolvidos é diminuta.

Depois de analisados os resultados para cada método desenvolvido foi medido o tempo de receção de uma *frame*. Recorreu-se ao mesmo procedimento utilizado na medição dos tempos de processamento em CPU mas aplicado à receção de uma única *frame*. O menor tempo obtido foi de 240 ms, já o valor médio correspondeu a 256,5 ms.

Por último foi feita a integração do projeto em aplicações reais. Foi utilizada uma sequência que realça os contornos do vídeo captado viabilizando a deteção dos espaços entre botões do tabliê de um carro. Foram averiguados dois parâmetros: a confirmação da deteção de contornos e a medição de tempos em CPU de forma a ser possível obter conclusões acerca da rapidez do processo.

Relativamente à deteção de contornos, confirma-se que o pré-processamento efetuado em FPGA permite a sua visualização como é possível observar na figura 4.18 à direita. Em termos de tempo de processamento, foi medido o tempo em FPGA e em CPU (*Halcon*, OpenCV e através das equações desenvolvidas em VHDL). Em FPGA, cada *frame* é atrasada em 8 ciclos de *clock*, ou seja, 80 ns. Considerando que em CPU o processamento não foi efetuado em tempo real, através da ferramenta *Halcon* o tempo obtido por *frame* foi de 30,2389 ms, ao utilizar as funções da biblioteca OpenCV o resultado foi 37,2415 ms e através das equações utilizadas em FPGA o tempo medido foi de 115,3149 ms. Note-se que, os métodos da ferramenta *Halcon* e da biblioteca OpenCV realizam o processamento das margens, o que não acontece na implementação em FPGA. Os resultados obtidos demonstram que ao comparar o tempo de processamento por pixel, a implementação em CPU é mais eficaz, porém, caso se considere o processamento por *frame* (tempo real), em FPGA, obtêm-se melhores resultados. Neste cenário, cada imagem é apenas atrasada em 8 ciclos de *clock* para ser processada (80 ns), enquanto que, através da ferramenta *Halcon* (resultado mais rápido em CPU), são necessários 30,2389 ms. Nestas circunstâncias, é possível concluir que a utilização de pré-processamento de imagem em FPGA em tempo real apresenta vantagens claras ao nível do tempo despendido na aplicação dos filtros face a uma implementação em CPU.

4.15 Comparação com GPU

Nesta secção são apresentados os resultados relativos aos tempos de processamento dos métodos descritos em 3.4.2-3.4.11 através de GPU. De forma a contextualizar o leitor quanto aos procedimentos necessários para o processamento de uma imagem através de GPU, de seguida é apresentada a lista numerada de procedimentos necessários:

1. Envio da imagem original armazenada em memória CPU para memória GPU

2. Processamento da imagem em GPU

3. Envio da imagem resultante armazenada em memória GPU para memória CPU

Ao analisar os procedimentos anteriores, é possível concluir que o tempo total de processamento de uma *frame* em GPU corresponde à soma do tempo de envio das duas imagens entre plataformas com o período de aplicação do método. Relativamente ao tempo de envio da imagem original para GPU, equivale a 3,3 ms. O tempo do processo inverso (envio da imagem resultante para CPU) corresponde a 5,5 ms, logo, de forma a ser efetuada a aplicação de um filtro em GPU são necessários 8,8 ms para o envio de imagens entre plataformas. Em relação ao tempo de aplicação dos filtros, na tabela 4.14 são apresentados o tempo mais rápido e o tempo médio por pixel e por *frame* para todos os filtros.

Tabela 4.14: Comparação dos resultados obtidos em GPU, FPGA e CPU

	GPU				CPU				FPGA	
	Tempo por Pixel (ns)		Tempo por Frame (ms)		Tempo por Pixel (ns)		Tempo por Frame (ms)		Tempo de Atraso (ns)	Número de Ciclos de Clock (Unidades)
Métodos	**Média**	**Mais Rápido**	**Média**	**Mais Rápido**	**Média**	**Mais Rápido**	**Média**	**Mais Rápido**		
RGB/cinza	0,044	0,032	0,092	0,066	6,69	5,30	13,87	10,99	10	1
RGB/YCbCr	0,044	0,031	0,091	0,065	9,50	9,16	19,70	18,99	10	1
Negativo	0,046	0,031	0,094	0,064	3,60	3,38	7,46	7,01	10	1
Modificação de Brilho	0,044	0,032	0,090	0,067	8,31	7,72	17,23	16,01	10	1
Binarização	0,044	0,033	0,092	0,068	2,59	2,41	5,37	5,00	10	1
Sobel	0,042	0,032	0,088	0,066	48,80	38,10	101,19	79,00	60	6
Média Uniforme	0,041	0,031	0,086	0,064	40,35	39,06	83,67	80,99	40	4
Gaussiano	0,043	0,031	0,090	0,064	26,43	25,56	54,81	53,00	30	3
Erosão	0,042	0,031	0,087	0,065	5,34	4,82	11,07	9,99	10	1
Dilatação	0,041	0,030	0,085	0,063	8,91	8,68	18,48	18,00	10	1

Por análise da tabela 4.14 é possível verificar que os tempos por pixel em GPU são bastante mais baixos comparativamente com CPU, sendo aproximadamente 100 vezes mais rápidos. Contudo, tendo em conta o tempo por *frame*, a implementação em GPU necessita do envio de uma imagem em memória CPU para GPU e vice-versa, logo, cada *frame* é processada em 8,8 ms mais o tempo de aplicação do filtro. Ao comparar o tempo por *frame* das duas plataformas é possível concluir que nos métodos negativo e binarização é mais eficiente a utilização de CPU (tempo de processamento menor que aproximadamente 8,8 ms), enquanto que, nos restantes filtros a implementação em GPU é mais rápida.

Os resultados dos tempos em GPU mostram que estes não aumentam significativamente com a complexidade do algoritmo, ou seja, os tempos de execução são idênticos para todos os algoritmos. Na figura 4.21, está representado o processo utilizado caso a aplicação dos filtros fosse efetuada em GPU. Tendo em conta o processamento em tempo real, em FPGA cada *frame* original é apenas atrasada no máximo em 6 ciclos de *clock* (60 ns) para uma frequência de *clock* de 100 MHz (filtro *Sobel*). Por outro lado, a implementação em GPU necessita aproximadamente de 8,8 ms para processar uma imagem. A partir destes dados é possível concluir que nesta situação específica a execução do pré-processamento de imagem em FPGA é também mais eficaz comparativamente com GPU. Note-se que, as características do computador utilizado na medição dos tempos em GPU

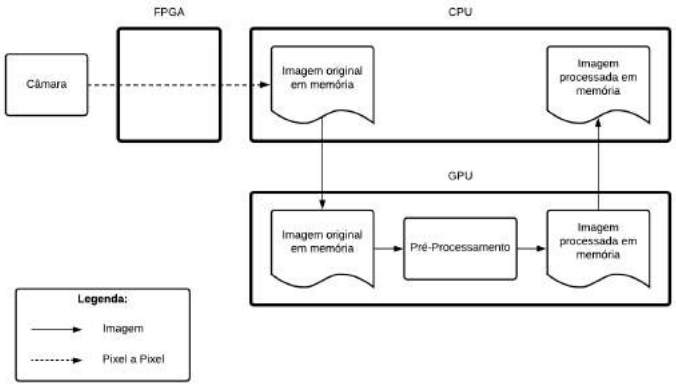


Figura 4.21: Método de aplicação dos filtros de pré-processamento em GPU

são: processador *Intel i7-8750H CPU @ 2.20GHz * 12* com 12 *cores*, 16 GB de memória RAM e placa gráfica *NVIDIA GeForce GXT 1060*.

Capítulo 5

Conclusão

De forma a desenvolver um sistema de pré-processamento em FPGA é necessário ter em consideração que a sua constituição tem por base três fases principais: captação de imagem, processamento de dados e envio dos resultados para extração de características. Durante o desenvolvimento do projeto foram realizadas várias tarefas como a compreensão do *Zybo Z7 Pcam 5C demo* (utilizado como arquitetura base), implementação dos módulos de pré-processamento de imagem e transmissão de *frames* filtradas para CPU via *Ethernet*, de modo a ser possível integrar o projeto em aplicações reais e averiguar a veracidade dos resultados obtidos (validação).

O *demo* utilizado apresenta como sinal de entrada vídeo captado via câmara, inclui a comunicação com o processador *Zynq-7000* e possibilita a visualização do resultado via HDMI. A sua utilização ofereceu bastantes benefícios no desenvolvimento do projeto pois, apresenta uma secção de captação de imagem completamente configurada, um meio de visualização dos dados, assim como, um projeto na ferramenta de *software Vivado SDK* com finalidade de configurar os módulos do circuito possibilitando a receção e transmissão de vídeo.

De modo a confirmar a veracidade dos métodos desenvolvidos foi efetuada a extração de *frames* do vídeo processado seguida de uma comparação com os mesmos algoritmos implementados em CPU, sendo que, os resultados obtidos demonstram o bom funcionamento dos filtros. Foi também realizada uma comparação de tempos de processamento, na qual, a FPGA sobressaiu relativamente ao CPU quando comparados os tempos por *frame*. Como se trata de processamento em tempo real, a implementação dos filtros em FPGA atrasa a imagem processada no máximo em 6 ciclos de *clock* (método *Sobel*). Por último, foi avaliada a quantidade de recursos utilizados por cada método em FPGA onde se concluiu que os algoritmos são eficientes, devido a necessitarem de baixas quantidades de recursos.

Na comparação entre imagens processadas em CPU e FPGA, o erro elevado das conversões RGB/cinza e RGB/YCbCr pode ser explicado pelo facto das duas imagens em questão (original e processada) não corresponderem à mesma *frame*.

Uma parte importante do presente projeto é a implementação de sequências de filtros através dos módulos desenvolvidos. Para esse efeito foi necessário ter em conta que nos métodos que necessitam de mais de um ciclo de *clock* para processar um pixel, a imagem resultante é apresentada desfasada horizontalmente em tantas posições quantos os ciclos de *clock* de atraso, ou seja, a utilização sequencial de vários filtros que utilizem máscara de convolução retira informação de algumas linhas e colunas à imagem resultante, o que não é significativo tendo em conta que a resolução da imagem é elevada (1920x1080 píxeis).

Foram efetuadas alterações para as sequências de filtros terem o comportamento desejado. O sinal TREADY do protocolo *AXI-Stream* tem de ser configurado como assíncrono em todos os módulos de modo a todos os métodos serem notificados quando é que o último *slave* está preparado para receber dados. Por fim, para assegurar o cumprimento dos tempos em FPGA, e por conseguinte, assegurar a correta transmissão de *frames* para CPU, foi diminuída a frequência de *clock* para 100 MHz. Note-se que estas alterações só são necessárias para as sequências que apresentam mais de 2 ciclos de *clock* de atraso.

No seguimento da implementação da sequência de filtros foi realizada a integração no projeto CheckMate da empresa Introsys, na qual, foi possível identificar os contornos referentes aos espaços entre os botões do tabliê de um carro, assim como, a comparação de tempos de processamento entre CPU e FPGA. Ao processar o vídeo captado em FPGA (tempo real), as *frames* são apenas atrasadas em 8 ciclos de *clock* (80 ns), enquanto que, o processamento em CPU demora no mínimo 30,2389 ms (Halcon). Estes resultados demonstram a vantagem da realização de pré-processamento em FPGA comparativamente a CPU.

Por último foram comparados os resultados obtidos para FPGA com medições em GPU. Considerando o tempo de aplicação de um filtro, em GPU são obtidos melhores resultados porém, há que ter em consideração o envio de imagens entre CPU/GPU para obter o tempo de processamento real (aproximadamente 8,8 ms). Tendo em conta que em FPGA o máximo atraso por filtro é de apenas 60 ns (6 eventos de *clock*) e que em GPU são necessários aproximadamente 8,8 ms para processar uma *frame*, na situação em estudo o pré-processamento em FPGA é mais eficiente para todos os métodos.

O desenvolvimento de um sistema de pré-processamento de imagem em tempo real em FPGA é uma tarefa de complexidade elevada. Foram encontrados desafios devido à falta de experiência de trabalho com a ferramenta *Vivado 2016.4*, assim como, o desconhecimento dos protocolos de comunicação presentes no sistema desenvolvido (*AXI4-Stream* e UDP). Existem algumas limitações no desenvolvimento deste tipo de sistemas em *hardware* como: a complexidade da linguagem utilizada (VHDL) e o elevado tempo de síntese do circuito que impossibilitam a rápida implementação do projeto.

Ao longo da realização da presente dissertação foi publicado um artigo científico, com o título "Sistemas de Visão Industrial: Biblioteca Genérica de Pré Processamento de

Imagem em FPGA", nas atas das XVII Jornadas sobre Sistemas Reconfiguráveis (REC2021) [96].

Como trabalho futuro, destacar uma possível melhoria na rapidez dos filtros, ou seja, a otimização do número de ciclos de *clock* por método. Transmissão de *frames* para CPU mais eficiente (visto que em cada vetor enviado são desprezadas algumas posições), assim como, a otimização da receção das imagens através da ordenação dos pacotes recebidos, com a finalidade de não serem perdidas *frames*. Por fim, destacar também a implementação de mais métodos de forma a aumentar a biblioteca.

Bibliografia

[1] R. L. Silva, M. Rudek, A. L. Szejka e O. Canciglieri Junior. "Machine vision systems for industrial quality control inspections". Em: *IFIP Advances in Information and Communication Technology* 540 (2018), pp. 631–641. ISSN: 18684238. DOI: 10.1007/978-3-030-01614-2_58 (ver p. 1).

[2] H. Würschinger, M. Mühlbauer, M. Winter, M. Engelbrecht e N. Hanenkamp. "Implementation and potentials of a machine vision system in a series production using deep learning and low-cost hardware". Em: *Procedia CIRP* 90 (2020), pp. 611–616. ISSN: 22128271. DOI: 10.1016/j.procir.2020.01.121 (ver p. 2).

[3] O. O. Vergara-Villegas, V. G. Cruz-Sánchez, H. de Jesús Ochoa-Domiénguez, M. de Jesús Nandayapa-Alfaro e Á. Flores-Abad. "Automatic product quality inspection using computer vision systems". Em: *Lean Manufacturing in the Developing World*. Springer, 2014, pp. 135–156 (ver p. 2).

[4] V. Alonso, A. Dacal-Nieto, L. Barreto, A. Amaral e E. Rivero. "Industry 4.0 implications in machine vision metrology: An overview". Em: *Procedia Manufacturing* 41 (2019), pp. 359–366. ISSN: 23519789. DOI: 10.1016/j.promfg.2019.09.020. URL: https://doi.org/10.1016/j.promfg.2019.09.020 (ver pp. 2, 24).

[5] *See Q*. http://seeq.introsys.eu. Acedido: 2021-11-11 (ver p. 2).

[6] *Checkmate CONTIGO*. https://www.introsys.eu. Acedido: 2021-11-11 (ver p. 2).

[7] D. G. Bailey. *Design for embedded image processing on FPGAs*. John Wiley & Sons, 2011 (ver p. 5).

[8] S. Koranne. "Handbook of Open Source Tools". Em: *Handbook of Open Source Tools* (2011). DOI: 10.1007/978-1-4419-7719-9 (ver p. 5).

[9] J. P. Balarini e S. Nesmachnow. "A C++ Implementation of Otsu's Image Segmentation Method". Em: *Image Processing On Line* 5 (2016), pp. 155–164. ISSN: 2105-1232. DOI: 10.5201/ipol.2016.158 (ver p. 6).

[10] A. Tourani e A. Shahbahrami. "Vehicle counting method based on digital image processing algorithms". Em: *2015 2nd International Conference on Pattern Recognition and Image Analysis, IPRIA 2015* Ipria (2015), pp. 10–15. DOI: 10.1109/PRIA.2015.7161621 (ver p. 6).

[11] G. B. Garcıéa, O. D. Suarez, J. L. E. Aranda, J. S. Tercero, I. S. Gracia e N. V. Enano. *Learning image processing with OpenCV*. Packt Publishing Ltd, 2015 (ver p. 7).

[12] C. E. Widodo, K. Adi e R. Gernowo. "Medical image processing using python and open cv". Em: *Journal of Physics: Conference Series* 1524.1 (2020). ISSN: 17426596. DOI: 10.1088/1742-6596/1524/1/012003 (ver p. 7).

[13] I. B. Mustaffa e S. F. B. M. Khairul. "Identification of fruit size and maturity through fruit images using OpenCV-Python and Rasberry Pi". Em: *Proceeding of 2017 International Conference on Robotics, Automation and Sciences, ICORAS 2017* 2018-March (2018), pp. 1–3. DOI: 10.1109/ICORAS.2017.8308068 (ver p. 8).

[14] A. S. Rathore. "Lane Detection for Autonomous Vehicles using OpenCV Library". Em: *International Research Journal of Engineering and Technolog* 6.1 (2019), pp. 1326–1332 (ver p. 8).

[15] M. Alasdair. "An Introduction to Digital Image Processing with Matlab, Notes for SCM2511 Image Processing 1". Em: *Jurnal Ilmiah Elite Elektro* 2.2 (2014), pp. 83–87 (ver pp. 9, 10).

[16] J. L. Epiphany. "On Teaching Digital Image Processing with MATLAB". Em: *American Journal of Signal Processing* January 2014 (2014). URL: https://www.researchgate.net/publication/275038286_On_Teaching_Digital_Image_Processing_with_MATLAB (ver p. 10).

[17] R. Kalaivani, S. Muruganand e A. Periasamy. "Identifying the quality of tomatoes in image processing using matlab". Em: *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering* 2.8 (2013), pp. 3525–3531 (ver p. 10).

[18] S. N. Ghaiwat e P. Arora. "Detection and Classification of Plant Leaf Diseases Using Image processing Techniques: A Review". Em: *International Journal of Recent Advances in Engineering & Technology (IJRAET) ISSN (Online)* 2 (2014), pp. 2347–2812. URL: https://www.researchgate.net/publication/321587357_Detection_and_Classification_of_Plant_Leaf_Diseases_in_Image_Processing_using_MATLAB (ver p. 10).

[19] R. Yusnita, N. Fariza e B. Norazwinawati. "Intelligent Parking Space Detection System Based on Image Processing". Em: *International Journal of Innovation, Management and Technology* 3.3 (2012), pp. 232–235. URL: http://www.ijimt.org/papers/228-G0038.pdf (ver p. 10).

[20] R. E. Haskell e D. M. Hanna. *Introduction to Digital Design Using Digilent FPGA Boards*. 2009, p. 175. ISBN: 9780980133790 (ver p. 11).

[21] Y. Chen e V. Dinavahi. "FPGA-based real-time EMTP". Em: *IEEE Transactions on Power Delivery* 24.2 (2008), pp. 892–902 (ver p. 11).

[22] X. Tang, E. Giacomin, G. D. Micheli e P. Gaillardon. "FPGA-SPICE: A Simulation-Based Architecture Evaluation Framework for FPGAs". Em: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.3 (2019), pp. 637–650. DOI: 10.1109/TVLSI.2018.2883923 (ver pp. 11, 12).

[23] *DSP Functions on FPGAs*. url: https://www.mathworks.com/company/newsletters/articles/dsp-functions-on-fpgas.html. Acedido: 2020-12-6 (ver p. 11).

[24] D. Bhowmik e K. Appiah. "Embedded vision systems: A review of the literature". Em: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10824 LNCS.May (2018), pp. 204–216. ISSN: 16113349. DOI: 10.1007/978-3-319-78890-6_17 (ver p. 12).

[25] A. HajiRassouliha, A. J. Taberner, M. P. Nash e P. M. Nielsen. "Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms". Em: *Signal Processing: Image Communication* 68.July (2018), pp. 101–119. ISSN: 09235965. DOI: 10.1016/j.image.2018.07.007. URL: https://doi.org/10.1016/j.image.2018.07.007 (ver p. 13).

[26] J. Fowers, G. Brown, P. Cooke e G. Stitt. "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications". Em: *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. 2012, pp. 47–56 (ver pp. 13, 14).

[27] S. Asano, T. Maruyama e Y. Yamaguchi. "Performance comparison of FPGA, GPU and CPU in image processing". Em: *FPL 09: 19th International Conference on Field Programmable Logic and Applications* (2009), pp. 126–131. DOI: 10.1109/FPL.2009.5272532 (ver pp. 14, 15, 17).

[28] D. Chen e D. Singh. "Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms". Em: *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC* (2013), pp. 297–304. DOI: 10.1109/ASPDAC.2013.6509612 (ver pp. 15–17).

[29] R. N. Rakvic, H. Ngo, R. P. Broussard e R. W. Ives. "Comparing an FPGA to a cell for an image processing application". Em: *Eurasip Journal on Advances in Signal Processing* 2010 (2010). ISSN: 16876172. DOI: 10.1155/2010/764838 (ver p. 16).

[30] E. Ormaetxea, J. Andreu, I. Kortabarria, U. Bidarte, I. M. De Alegría, E. Ibarra e E. Olaguenaga. "Matrix converter protection and computational capabilities based on a system on chip design with an FPGA". Em: *IEEE Transactions on Power Electronics* 26.1 (2011), pp. 272–287. ISSN: 08858993. DOI: 10.1109/TPEL.2010.2062539 (ver p. 17).

[31] *Zynq-7000 SoC*. url: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Acedido: 2020-12-10 (ver p. 18).

[32] Y. Zhou, Z. Chen e X. Huang. "A system-on-chip FPGA design for real-time traffic signal recognition system". Em: *Proceedings - IEEE International Symposium on Circuits and Systems* 2016-July (2016), pp. 1778–1781. ISSN: 02714310. DOI: 10.1109/ISCAS.2016.7538913 (ver p. 18).

[33] Z. Yuteng. "Computer Vision System-On-Chip Designs for Intelligent Vehicles". Tese de doutoramento. Faculty of the Worcester Polytechnic Institute, 2018 (ver p. 18).

[34] P.-Y. Hsiao, C.-W. Yeh, S.-S. Huang e L.-C. Fu. "A portable vision-based real-time lane departure warning system: day and night". Em: *IEEE Transactions on Vehicular Technology* 58.4 (2008), pp. 2089–2094 (ver p. 18).

[35] C. Lipski, B. Scholz, K. Berger, C. Linz, T. Stich e M. Magnor. "A fast and robust approach to lane marking detection and lane tracking". Em: *2008 ieee southwest symposium on image analysis and interpretation*. IEEE. 2008, pp. 57–60 (ver p. 18).

[36] S. Eetha, S. Agarwal e S. Neelam. "Zynq FPGA based system design for video surveillance with sobel edge detection". Em: *Proceedings - 2018 IEEE 4th International Symposium on Smart Electronic Systems, iSES 2018* (2018), pp. 76–79. DOI: 10.1109/iSES.2018.00025 (ver pp. 18, 19).

[37] A. B. Amara, E. Pissaloux e M. Atri. "Sobel edge detection system design and integration on an FPGA based HD video streaming architecture". Em: *2016 11th International Design & Test Symposium (IDT)*. IEEE. 2016, pp. 160–164 (ver pp. 18, 19).

[38] S. Areibi e A. T. Bld. *A First Look at VHDL For Digital Design*. Rel. téc. School of Engineering at the University of Guelph, 2019 (ver p. 19).

[39] D. J. Ayyed. "Image Steganography Based Sobel Edge Detection Using FPGA". Em: *Technium* 2.6 (2020), pp. 23–34 (ver p. 19).

[40] J. Carmona-Suárez, I. Santillan, S. Martıénez-Dıéaz, J. Gómez-Torres e J. Sandoval-Galarza. "Image Processing Implementation on FPGA Hardware with Mathematical Morphological Operators Centered in VHDL". Em: *Memorias del Congreso Nacional de Control Automático* (2018) (ver p. 19).

[41] R. B. Reese e M. A. Thornton. *Introduction to logic synthesis using Verilog HDL*. Vol. 6. 2006, pp. 1–84. ISBN: 1598291076. DOI: 10.2200/S00060ED1V01Y200610DCS006 (ver p. 20).

[42] N. Singla e M. S. Narula. "FPGA Implementation of Image Enhancement Using Verilog HDL". Em: *International Research Journal of Engineering and Technology (IRJET)* 05.05 (2018), pp. 1794–1794 (ver p. 20).

[43] S. Neethu Raj e Alex. V. "Parallel Block-Based Architecture for Improved Edge Detection in Verilog". Em: *International Journal of Engineering Research and* V4.07 (2015), pp. 995–1001. DOI: 10.17577/ijertv4is070778 (ver p. 20).

[44] P. Coussy, D. D. Gajski, M. Meredith e A. Takach. "An introduction to high-level synthesis". Em: *IEEE Design & Test of Computers* 26.4 (2009), pp. 8–17 (ver p. 20).

[45] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons e F. Morgan. "Xilinx vivado high level synthesis: Case studies". Em: *IET Conference Publications* 2014.CP639 (2014), pp. 352–356. DOI: 10.1049/cp.2014.0713 (ver pp. 21, 22).

[46] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson e K. Bertels. "A Survey and Evaluation of FPGA High-Level Synthesis Tools". Em: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604. ISSN: 02780070. DOI: 10.1109/TCAD.2015.2513673 (ver p. 21).

[47] A. Cortes, I. Velez e A. Irizar. "High level synthesis using Vivado HLS for Zynq SoC: Image processing case studies". Em: *2016 Conference on Design of Circuits and Integrated Systems, DCIS 2016 - Proceedings* (2017), pp. 183–188. DOI: 10.1109/DCIS.2016.7845376 (ver pp. 21, 22).

[48] *HDL Coder*. url:https://www.mathworks.com/products/hdl-coder.html, journal=MATLAB Simulink. Acedido: 2021-01-14 (ver p. 22).

[49] *High-Level Synthesis Compiler - IntelÂ® HLS Compiler*. url:https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html. Acedido: 2021-01-16 (ver p. 22).

[50] D. G. Bailey. "The advantages and limitations of high level synthesis for FPGA based image processing". Em: *ACM International Conference Proceeding Series* 08-11-Sep- (2015), pp. 134–139. DOI: 10.1145/2789116.2789145 (ver p. 22).

[51] H. R. Zohouri. "High Performance Computing with FPGAs and OpenCL". Em: August (2018). arXiv: 1810.09773. URL: http://arxiv.org/abs/1810.09773 (ver p. 22).

[52] Intel. "Implementing FPGA Design with the OpenCL Standard". Em: *White Paper* November (2013), pp. 1–8. URL: www.altera.com (ver p. 23).

[53] K. Hill, S. Craciun, A. George e H. Lam. "Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA". Em: *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors* 2015-Septe (2015), pp. 189–193. ISSN: 10636862. DOI: 10.1109/ASAP.2015.7245733 (ver p. 23).

[54] H. Golnabi e A. Asadpour. "Design and application of industrial machine vision systems". Em: *Robotics and Computer-Integrated Manufacturing* 23.6 (2007), pp. 630–637. ISSN: 07365845. DOI: 10.1016/j.rcim.2007.02.005 (ver pp. 24, 26, 27).

[55] O. Semeniuta, S. Dransfeld, K. Martinsen e P. Falkman. "Towards increased intelligence and automatic improvement in industrial vision systems". Em: *Procedia CIRP* 67 (2018), pp. 256–261. ISSN: 22128271. DOI: 10.1016/j.procir.2017.12.209. URL: http://dx.doi.org/10.1016/j.procir.2017.12.209 (ver p. 24).

[56] A. L. Kleppe, A. Bjørkedal, K. Larsen e O. Egeland. "Automated assembly using 3D and 2D cameras". Em: *Robotics* 6.3 (2017). ISSN: 22186581. DOI: 10.3390/robotics6030014 (ver p. 25).

[57] S. Hussmann, M. Gonschior, B. Buttgen, C. Peter, S. Schwope, C. Perwass, M. Johannesson e E. Hallstig. "A Review on commercial solid state 3D Cameras for Machine Vision Applications". Em: *Recent Advances in Topography Research* 1 (2013), pp. 303–351 (ver p. 25).

[58] R. Gade e T. B. Moeslund. "Thermal cameras and applications: A survey". Em: *Machine Vision and Applications* 25.1 (2014), pp. 245–262. ISSN: 09328092. DOI: 10.1007/s00138-013-0570-5 (ver p. 25).

[59] Y. Shi e S. Lichman. "Smart Cameras : A Review". Em: *Proceedings of 2005 Asia-Pacific Workshop on Visual Information Processing* 11-13 December 2005 (2005), pp. 95–100 (ver p. 25).

[60] A. W. Malik, B. Thörnberg e P. Kumar. "Comparison of three smart camera architectures for real-time machine vision system". Em: *International Journal of Advanced Robotic Systems* 10 (2013), pp. 1–12. ISSN: 17298806. DOI: 10.5772/57135 (ver p. 25).

[61] K. Tout. "Automatic vision system for surface inspection and monitoring: Application to wheel inspection". Tese de doutoramento. Université de Technologie de Troyes-UTT, 2018 (ver p. 26).

[62] Y. Xie, Y. Ye, J. Zhang, L. Liu e L. Liu. "A physics-based defects model and inspection algorithm for automatic visual inspection". Em: *Optics and Lasers in Engineering* 52.1 (2014), pp. 218–223. ISSN: 01438166. DOI: 10.1016/j.optlaseng.2013.06.006. URL: http://dx.doi.org/10.1016/j.optlaseng.2013.06.006 (ver p. 26).

[63] Q. Zhou, R. Chen, B. Huang, C. Liu, J. Yu e X. Yu. "An automatic surface defect inspection system for automobiles using machine vision methods". Em: *Sensors (Switzerland)* 19.3 (2019). ISSN: 14248220. DOI: 10.3390/s19030644 (ver p. 26).

[64] B. C. F. De Oliveira, A. L. S. Pacheco, R. C. C. Flesch e M. B. Demay. "Detection of defects in the manufacturing of electric motor stators using vision systems: Electrical connectors". Em: *2016 12th IEEE International Conference on Industry Applications, INDUSCON 2016* July 2018 (2017). DOI: 10.1109/INDUSCON.2016.7874551 (ver p. 27).

[65] S. Satorres Martínez, J. Gómez Ortega, J. Gámez García e A. Sánchez García. "A machine vision system for defect characterization on transparent parts with non-plane surfaces". Em: *Machine Vision and Applications* 23.1 (2012), pp. 1–13. ISSN: 09328092. DOI: 10.1007/s00138-010-0281-0 (ver p. 27).

[66] H. Shen, S. Li, D. Gu e H. Chang. "Bearing defect inspection based on machine vision". Em: *Measurement: Journal of the International Measurement Confederation* 45.4 (2012), pp. 719–733. ISSN: 02632241. DOI: 10.1016/j.measurement.2011.12.018. URL: http://dx.doi.org/10.1016/j.measurement.2011.12.018 (ver p. 27).

[67] M. Hashemzadeh e N. Farajzadeh. "A Machine Vision System for Detecting Fertile Eggs in the Incubation Industry". Em: *International Journal of Computational Intelligence Systems* 9.5 (2016), pp. 850–862. ISSN: 18756883. DOI: 10.1080/18756891.2016.1237185 (ver p. 27).

[68] K. Parkot e A. Sioma. "Development of an automated quality control system of confectionery using a vision system". Em: *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2018*. Vol. 10808. International Society for Optics e Photonics. 2018, p. 1080817 (ver p. 27).

[69] C. Canali, F. Cannella, F. Chen, G. Sofia, A. Eytan e D. G. Caldwell. "An automatic assembly parts detection and grasping system for industrial manufacturing". Em: *IEEE International Conference on Automation Science and Engineering* August (2014), pp. 215–220. ISSN: 21618089. DOI: 10.1109/CoASE.2014.6899329 (ver p. 27).

[70] Q. Zhao, I. Taniguchi, M. Nakamura e T. Onoye. "An efficient parts counting method based on intensity distribution analysis for industrial vision systems". Em: *The 21st Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 2018)*. 2018, pp. 118–123 (ver p. 27).

[71] R. Furferi, L. Governi, L. Puggelli, M. Servi e Y. Volpe. "Machine vision system for counting small metal parts in electro-deposition industry". Em: *Applied Sciences (Switzerland)* 9.12 (2019), pp. 1–14. ISSN: 20763417. DOI: 10.3390/app9122418 (ver p. 27).

[72] P. Andhare e S. Rawat. "Pick and place industrial robot controller with computer vision". Em: *Proceedings - 2nd International Conference on Computing, Communication, Control and Automation, ICCUBEA 2016* (2017). DOI: 10.1109/ICCUBEA.2016.7860048 (ver p. 28).

[73] S. Kang, K. Kim, J. Lee e J. Kim. "Robotic vision system for random bin picking with dual-arm robots". Em: *MATEC Web of Conferences* 75 (2016). ISSN: 2261236X. DOI: 10.1051/matecconf/20167507003 (ver p. 28).

[74] A. Pretto, S. Tonello e E. Menegatti. "Flexible 3D localization of planar objects for industrial bin-picking with monocamera vision system". Em: *IEEE International Conference on Automation Science and Engineering* (2013), pp. 168–175. ISSN: 21618070. DOI: 10.1109/CoASE.2013.6654067 (ver p. 28).

[75] J. J. Rodriguez-Andina, M. D. Valdes-Pena e M. J. Moure. "Advanced Features and Industrial Applications of FPGAS-A Review". Em: *IEEE Transactions on Industrial Informatics* 11.4 (2015), pp. 853–864. ISSN: 15513203. DOI: 10.1109/TII.2015.2431223 (ver p. 28).

[76] Y. Lin. "Using FPGAs to solve challenges in industrial applications". Em: *EE times* (2011) (ver p. 28).

[77] A. M. Ashir, A. A. Ata e M. S. Salman. "FPGA-based image processing system for Quality Control and Palletization applications". Em: *2014 IEEE International Conference on Autonomous Robot Systems and Competitions, ICARSC 2014* (2014), pp. 285–290. DOI: 10.1109/ICARSC.2014.6849800 (ver p. 29).

[78] Ž. Hocenski, I. Aleksi e R. Mijaković. "Ceramic tiles failure detection based on FPGA image processing". Em: *IEEE International Symposium on Industrial Electronics* 40.ISlE (2009), pp. 2169–2174. DOI: 10.1109/ISIE.2009.5219911 (ver p. 29).

[79] H. Guo, H. Xiao, S. Wang, W. He e K. Yuan. "Real-time detection and classification of machine parts with embedded system for industrial robot grasping". Em: *2015 IEEE International Conference on Mechatronics and Automation, ICMA 2015* (2015), pp. 1691–1696. DOI: 10.1109/ICMA.2015.7237740 (ver p. 29).

[80] J. Rodríguez-Araújo, J. J. Rodríguez-Andina, J. Farina, F. Vidal, J. Mato e M. Angeles Montealegre. "Industrial laser cladding systems: FPGA-based adaptive control". Em: *IEEE Industrial Electronics Magazine* 6.4 (2012), pp. 35–46. ISSN: 19324529. DOI: 10.1109/MIE.2012.2221356 (ver p. 30).

[81] L. Kalms, A. Podlubne e D. Göhringer. "HiFlipVX: An Open Source High-Level Synthesis FPGA Library for Image Processing". Em: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11444 LNCS (2019), pp. 149–164. ISSN: 16113349. DOI: 10.1007/978-3-030-17227-5_12 (ver p. 31).

[82] M. Schmid, N. Apelt, F. Hannig e J. Teich. "An image processing library for C-based high-level synthesis". Em: *Conference Digest - 24th International Conference on Field Programmable Logic and Applications, FPL 2014* (2014), pp. 22–25. DOI: 10.1109/FPL.2014.6927424 (ver p. 31).

[83] *Zybo Z7 Pcam 5C Demo*. https://digilent.com/reference/learn/programmable-logic/tutorials/zybo-z7-pcam-5c-demo/start. Acedido: 2021-10-30 (ver p. 35).

[84] Xilinx. *AXI Reference Guide*. English. Xilinx. 15 Julho, 2017 (ver p. 35).

[85] Xilinx. *MIPI D-PHY*. English. Xilinx. 5 Abril, 2017 (ver p. 36).

[86] Xilinx. *MIPI CSI-2 Receiver Subsystem*. English. Xilinx. 16 Julho, 2020 (ver p. 36).

[87] T. Kappenman. *Embedded Vision Demo*. English. Digilent. 8 Agosto, 2019 (ver pp. 37, 39).

[88] Xilinx. *AXI Video Direct Memory Access*. English. Xilinx. 30 Novembro, 2016 (ver p. 37).

[89] Xilinx. *Video Timing Controller*. English. Xilinx. 22 Maio, 2019 (ver p. 38).

[90] Xilinx. *AXI4-Stream to Video Out*. English. Xilinx. 4 Outubro, 2017 (ver p. 38).

[91] E. Gyorgy. *RGB-to-DVI (Source)*. English. Digilent. 6 Setembro, 2017 (ver p. 38).

[92] U. Mistry. "lwIP: Light Weight IP". Em: *The George Washington University* (2011) (ver p. 52).

[93] J.-H. Huh. "Reliable user datagram protocol as a solution to latencies in network games". Em: *Electronics* 7.11 (2018), p. 295 (ver p. 52).

[94] *LwIP UDP Client*. https://github.com/Xilinx/embeddedsw/tree/master/lib/sw_apps/lwip_udp_perf_client/src. Acedido: 2021-06-20 (ver p. 52).

[95] *Halcon*. https://www.mvtec.com/products/halcon. Acedido: 2021-10-20 (ver p. 74).

[96] D. Ferreira, F. Moutinho e P. Deusdado. "Sistemas de Visão Industrial: Biblioteca Genérica de Pré Processamento de Imagem em FPGA". Em: *XVII Jornadas sobre Sistemas Reconfiguráveis (REC2021), Porto, Portugal, 5-6 Julho*. 2021. DOI: 10.5281/zenodo.5081110 (ver p. 83).