Victor Paşcan

Licenciado em Ciências da Engenharia Electrotécnica e de Computadores

Dispositivos Reconfiguráveis em Processamento de Imagem – Aplicação à detecção de faixas de rodagem

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Orientador: Prof. Doutor Luís Filipe dos Santos Gomes, Professor Associado com Agregação, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri

Presidente:	Doutor Arnaldo Manuel Guimarães Batista
Arguente:	Doutor Filipe de Carvalho Moutinho
Vogal:	Doutor Luís Filipe dos Santos Gomes

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2020

Dispositivos Reconfiguráveis em Processamento de Imagem – Aplicação à detecção de faixas de rodagem

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

O mundo é para quem nasce para o conquistar
E não para quem sonha que pode conquistá-lo

— Álvaro de Campos, *Tabacaria* —

Valete, Fratres.

Agradecimentos

Em primeiro lugar, queria agradecer ao meu orientador Professor Luís Gomes que possibilitou a realização da presente dissertação, encaminhou o trabalho desenvolvido apresentando conselhos importantes, e manifestou sempre disponibilidade para esclarecer as minhas dúvidas.

Gostaria de expressar um agradecimento especial a minha família, pois o meu inteiro percurso é resultado das suas lições, influências, esperanças, conselhos, sua coragem e energia inesgotável.

Queria agradecer ainda em particular aos meus prezados e amados pais, aos quais nunca consegui formular e transmitir devidamente a minha gratidão, por tudo que tiveram de abnegar e sacrificar em prol de proporcionarem melhores oportunidades para a auto-realização dos filhos.

"À querida alma irmã da minha, à minha Irmã"[1], que sempre foi o meu guia, e mentora, e cuja dedicação e resiliência foram sempre uma fonte de admiração e inspiração para mim.

[1] Florbela Espanca, *As Máscaras do Destino* (adaptado).

Resumo

O *Field-Programmable Gate Array (FPGA)* tem sido cada vez mais explorado e investigado como plataforma de prototipagem e implementação de sistemas em variadíssimas áreas, incluindo as de processamento de imagem e visão computacional, uma vez que a sua arquitectura massivamente paralela proporciona numerosos benefícios no desempenho, custos e gastos energéticos quando comparado com os processadores tradicionais.

O principal objectivo desta dissertação consiste em desenvolver e implementar uma aplicação de detecção de faixa de rodagem automóvel baseada em FPGA, que permita identificar diferentes tipos de linhas separadoras de faixa de rodagem, nomeadamente as linhas da faixa em que se localiza o carro, as linhas das potenciais faixas e as linhas verticais de transição de faixa, quando o carro efectua a passagem de uma faixa de rodagem para outra. Pretende-se que a aplicação abranja todos os processos de processamento, nomeadamente de aquisição de dados vídeo, o seu processamento propriamente dito e a apresentação do resultado final num monitor.

Os métodos e as funções desenvolvidas foram validadas utilizando as linguagens de programação mais populares no momento, C e C++, recorrendo também a várias bibliotecas C, sendo a mais usada a biblioteca *Video* do ambiente de desenvolvimento Vivado HLS, cujas funções de processamento de vídeo são compatíveis com funções existentes de OpenCV. Para a implementação do sistema foi usado um *Embedded Vision Bundle* da Digilent, ou seja, uma placa Zybo Z7-20 em conjunto com um módulo de imagem Pcam 5C.

Palavras-chave: *Field-Programmable Gate Array*, Detecção de Faixa de Rodagem Automóvel, Transformada de Hough, Vivado HLS.

Abstract

The Field-Programmable Gate Array (FPGA) has been increasingly explored and investigated as a platform for prototyping and system implementation in a wide range of areas, including areas of image processing and computer vision, since its massively parallel architecture provides numerous benefits in performance, cost and power consumption compared to traditional processors.

The main objective of this dissertation is to develop and implement an FPGA based lane detection application, that can identify different types of lane lines, namely, the lane lines in which the car is located, the lines of the potential lanes and the vertical lines, when the car moves from one lane to another. The intention is to create an application that covers all involved processes, namely video data acquisition process, the processing of the data, and the presentation of the final result on a monitor.

The developed methods and functions were validated using C and C++ programming languages, which are still the most popular worldwide. Also, several C libraries were used, Video library of Vivado HLS development environment being the most used, whose video processing functions are compatible with existing OpenCV functions. For the implementation of the system was used a Digilent Embedded Vision Bundle, which consists of a Zybo Z7-20 board and a Pcam 5C image module.

Keywords: Field-Programmable Gate Array, Lane Detection, Hough Transform, Vivado HLS.

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Siglas

ADAS Advanced Driver Assistance System

ADC Analog-to-Digital Converter

AMBA Advanced Microcontroller Bus Architecture

API Application Programming Interface

APSoC All Programmable System-on-Chip

APU Application Processing Unit

ARM Advanced RISC Machine

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

CAN Controller Area Network

CEC Consumer Electronics Control

CLB Configurable Logic Block

CNN Convolution Neural Network

CPU Central Processing Unit

CSI Camera Serial Interface

DDC Display Data Channel

DDR Double Data Rate

DDR3 Double Data Rate Type 3

DDR3L DDR3 Low Voltage

DMA Direct Memory Access

DRL Dynamically Reconfigurable Logic

DSP Digital Signal Processing

DVI Digital Visual Interface

FFC Flat-Flexible Cable

FIFO First In, First Out

FPGA Field-Programmable Gate Array

FPS Frames Per Second

FSM Finite State Machine

GND Ground

GPIO General Purpose Input/Output

GPU Graphics Processing Unit

HDL Hardware Description Language

HDMI High-Definition Multimedia Interface

HLD High-Level Design

HLL High-Level Language

HLS High-Level Synthesis

HNF Hessian Normal Form

HPC High-Performance Computing

HPD Hot Plug Detect

HPRC High-Performance Reconfigurable Computing

HT Hough Transform

I/O Input/Output

I2C Inter-Integrated Circuit

IBM International Business Machines Corporation

IMU Inertial Measurement Unit

IPM Inverse Perspective Mapping

JTAG Joint Test Action Group

LED Light-Emitting Diode

LUT Lookup-Table

MACC Multiplier Accumulator

MIO Multiplexed I/O

MMCM Mixed-Mode Clock Manager

MSPS Mega Sample Per Second

MUX Multiplexer

PCB Printed Circuit Board

PL Programmable Logic

PS Processing System

QVGA Quorter VGA

RAM Random-Access Memory

RANSAC RANdom SAmple Consensus

RC Reconfigurable Computing

RNN Recurrent Neural Network

ROB Re-Order Buffer

ROC Receiver Operating Characteristic

ROI Region Of Interest

RTL Register Transfer Level

SD Secure Digital

SDIO Secure Digital Input Output

SDK Software Development Kit

SDSoC Software-Defined System On Chip

SIMD Single Instruction, Multiple Data

SoC System-on-Chip

SPI Serial Peripheral Interface

SRAM Static Random-Access Memory

TLB Translation Lookaside Buffer

UART Universal Asynchronous Receiver/Transmitter

USB Universal Serial Bus

VDMA Video Direct Memory Access

VGA Video Graphics Array

ZIF Zero Insertion Force

Capítulo 1

Introdução

1.1 Enquadramento e motivação

Actualmente a tecnologia computacional evolui muito rapidamente e um dos seus domínios proeminente, de muito valor, interesse e sempre em contínuo desenvolvimento é o do processamento de imagem. O processamento de imagem digital realiza processamento, manipulação e interpretação automática da informação visual e desempenha um papel cada vez mais crescente em muitos aspectos da nossa vida diária, assim como numa variedade enorme de disciplinas, ramos científicos e tecnológicos.

As aplicações no âmbito das actividades humanas podem ser encontradas desde as áreas da segurança e saúde, até indústria e entretenimento [1]. Nos sistemas de manufactura o processamento de imagem é bastante usado nos sistemas automáticos de inspecção e classificação visual que conseguem identificar componentes defeituosos o que contribui para um aumento de produtividade e qualidade dos produtos. Na medicina é extremamente útil nas tarefas de diagnóstico, como por exemplo, nas imagens biomédicas conseguem-se identificar objectos de interesse (tumores, órgãos, etc.), fazer a sua medição e interpretar os seus resultados. Na segurança o processamento de imagem é usado para vigilância, classificação e identificação de objectos em zonas terrestres, marítimas ou aéreas de interesse. No entretenimento todos os anos aparecem novas tecnologias que tiram partido do processamento de imagem em tempo real, como por exemplo, nos jogos de futebol criam-se linhas virtuais que ajudam os árbitros ver o posicionamento dos jogadores e tomar decisões acerca da existência de fora-de-jogo, nos jogos da Associação Nacional de Basquetebol, época de 2019-2020, fizeram-se experiências com relógios virtuais nos círculos restritivos para facilitar ao telespectador a leitura do tempo.

Outra área onde o processamento de imagem é muito usado é na segurança rodoviária. Os sistemas avançados de assistência ao condutor, ou *Advanced Driver Assistance System*

(ADAS), através da automação e adaptação proporcionam uma condução melhor e mais segura. As principais causas dos acidentes rodoviários são os erros humanos, como a desatenção, ou comportamento indevido ao volante. Os sistemas ADAS reduzem a carga de trabalho do condutor e sempre que se deparam com uma situação perigosa eles alertam o condutor, ou então podem até assumir um papel activo ao realizar acções correctivas necessárias para evitar acidentes.

A detecção de faixa de rodagem automóvel é um aspecto fundamental da maioria dos sistemas ADAS actuais. Devido ao amplo conhecimento de processamento de imagem existente e ao baixo custo dos dispositivos de câmara, um número considerável de trabalhos de investigação foca-se no estudo de métodos de detecção baseados em visão computacional (*computer vision*) [2].

Como os domínios que usam técnicas de processamento de imagem são distintos não podem ser utilizadas sempre as mesmas abordagens. As características cada vez mais procuradas são o baixo consumo energético, velocidade e custos razoáveis.

Os FPGAs(*Field-Programmable Gate Array*) trazem imensos benefícios para processamento de imagem em tempo real devido ao seu *hardware* que é capaz de efectuar operações em paralelo. No entanto, isso requer adaptação dos algoritmos tradicionais para plataformas reconfiguráveis, uma vez que esses foram desenvolvidos com a arquitectura dos CPUs (*Central Processing Unit*) sequenciais em mente.

1.2 Objectivos

Os principais objectivos deste trabalho são apresentar e validar um conjunto de conceitos que indiciam os FPGAs como dispositivos alternativos, capazes de efectuar processamento de imagem.

Para comprovar que nestes dispositivos é possível desenvolver aplicações que beneficiem da sua arquitectura massivamente paralela mas sem muitos inconvenientes das linguagens de descrição de *hardware* tradicionais, pretende-se implementar e validar todos os algoritmos usando as linguagens de programação C/C++, que ainda permanecem as linguagens mais utilizadas e populares no mundo. Além disso tenciona-se tirar partido ao máximo de bibliotecas C optimizadas para implementações em FPGA, particularmente, a biblioteca *Video* de Vivado HLS cujas funções de processamento vídeo foram concebidas para acelerar funções de OpenCV.

Para a validação foi estabelecido desenvolver uma aplicação de processamento de imagem em tempo real baseada em FPGA, mais concretamente, uma aplicação de detecção de faixa de rodagem automóvel. A aquisição das imagens é realizada através de um módulo de imagem Pcam 5C e o tratamento da informação e a implementação dos algoritmos numa placa Zybo Z7-20.

1.3 Estrutura do Documento

A presente dissertação é composta por cinco capítulos. O primeiro capítulo serve como uma breve introdução aos objectivos e motivações do trabalho desenvolvido.

No segundo capítulo são apresentados conceitos e trabalhos relacionados sobre a temática da dissertação. Este é dividido em duas secções. Na primeira secção pretendeu-se indicar o FPGA como plataforma viável e potencial alternativa para a implementação de aplicações de processamento de imagem. Para este efeito, em primeiro lugar, é efectuada uma breve descrição da arquitectura do FPGA (2.1.1), seguindo-se os seus casos de uso em processamento de imagem, com especial consideração para a sua utilização como acelerador de *hardware* (2.1.2). São ainda apresentadas as caracteríticas mais diferenciadoras dos FPGAs, a sua reconfiguração dinâmica e parcial (2.1.3). Por fim o FPGA é comparado com outras plataformas em processamento de imagem (2.1.4).

Na segunda secção são apresentados vários métodos e trabalhos desenvolvidos para a detecção de faixas de rodagem automóvel (2.2.1), e são descritos os conceitos dos métodos que foram depois utilizadas na implementação deste trabalho (2.3).

No capítulo 3 são apresentadas plataformas, dispositivos e ferramentas usadas neste trabalho.

No quarto capítulo é apresentada a arquitectura do método proposto para detecção de faixas de rodagem automóvel, são também descritos alguns resultados obtidos, insucessos e limitações detectadas na implementação.

No quinto capítulo são apresentadas as principais conclusões acerca do trabalho desenvolvido e algumas sugestões para trabalhos futuros.

No fim do documento encontram-se todas as referências bibliográficas utilizadas para a escrita e o desenvolvimento da dissertação, bem como os anexos com algoritmos desenvolvidos.

Capítulo 2

Conceitos e Trabalhos Relacionados

Este capítulo está dividido em duas secções, na primeira são identificadas as vantagens, desvantagens e os casos de uso dos FPGAs no processamento de imagem, comparando também as suas características com outros dispositivos de prototipagem, os CPUs e as GPUs. Na segunda secção são explorados vários métodos, estratégias e conceitos de detecção de faixas de rodagem automóvel que serviram como apoio e inspiração para a parte práctica da dissertação.

2.1 Processamento de Imagem em FPGAs

2.1.1 Arquitectura do *Field-Programmable Gate Array*

Antes de mais, é importante descrever um pouco a arquitectura e o funcionamento dum *Field-Programmable Gate Array (FPGA)*, para a seguir tornar mais fácil a compreensão, interpretação e visualização dos seus componentes e características. Na figura 2.1 é apresentada uma arquitectura simples do mesmo. Um FPGA é composto por blocos CLB (*Configurable Logic Block*), que são programáveis, tal como os blocos de entradas e saídas, e por sua vez, são todos interconectados através de caminhos de roteamento programáveis. Um CLB básico é mostrado na figura 2.2. Ele é constituído por uma LUT (*Lookup-Table*), um *flip-flop* e por um *Multiplexer (MUX)*. Uma combinação de entrada é guardada na LUT, e o MUX é usado para escolher entre a saída directa da LUT ou saída do registo da LUT. A combinação das unidades CLB é usada para construir qualquer circuito lógico desejado. Uma caixa de conexões é usada para interconectar os vários recursos de encaminhamento e conectar as saídas dos CLBs aos caminhos de roteamento.

Para programar FPGAs baseados em *Static Random-Access Memory (SRAM)* são usados *bitstreams*. O fluxograma da figura 2.3 apresenta os vários passos envolvidos na implementação *hardware*. No primeiro passo descreve-se o *hardware* do sistema em questão,

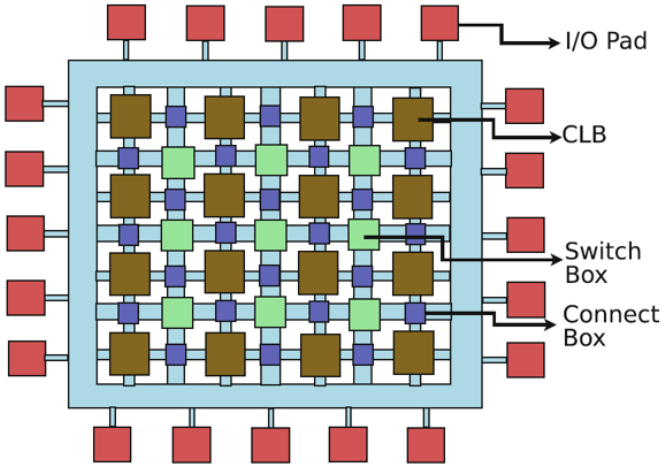


Figura 2.1: Arquitectura básica do FPGA [3].

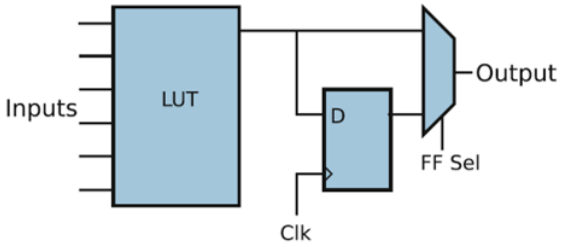


Figura 2.2: Estrutura básica dum CLB [3].

cujos circuitos lógicos podem ser descritos em HDL (*Hardware Description Language*), ou com esquemáticos. As linguagens de programação de alto nível, ou *High-Level Language (HLL)*, como C, também são usadas para descrever o *hardware*. As ferramentas de síntese fazem optimizações no *hardware* descrito e convertem-no em *netlist* de nível de portas. As ferramentas de tradução empacotam estas portas em LUTs. A ferramenta de *place and route* optimiza o posicionamento dos LUTs de modo a terem um caminho menos crítico. Tipicamente, isto consegue-se ao colocar os LUTs mais perto uns dos outros. A ferramenta de encaminhamento define a interconectividade entre os LUTs, ou seja, optimiza o uso de fios longos e curtos de modo que seja usado o número mínimo de blocos interruptores (*switch box*). Por fim, é gerado um *bitstream* para programar cada LUT e interconexão. É esse *bitstream* que é utilizado para programar o FPGA.

Muitos estudos foram realizados sobre a arquitectura do FPGA, tanto pela academia como pela indústria. Os autores de [3], livro em que baseei a descrição do FPGA, examinaram os resultados de muitos grupos de investigação (nomeadamente [4], [5], [6]), e relataram que o tamanho ideal do LUT, ou seja, o seu número de entradas, é entre 4 e 6, não obstante, os FPGAs de hoje disponíveis comercialmente chegam a ter tamanhos de LUT 8. O tamanho é maioritariamente dependente do tamanho da agregação dos CLBs e da largura do caminho de roteamento. Os LUTs maiores reduzem os atrasos de interconexão, mas ocupam mais espaço e são mais lentos devido aos circuitos internos necessários

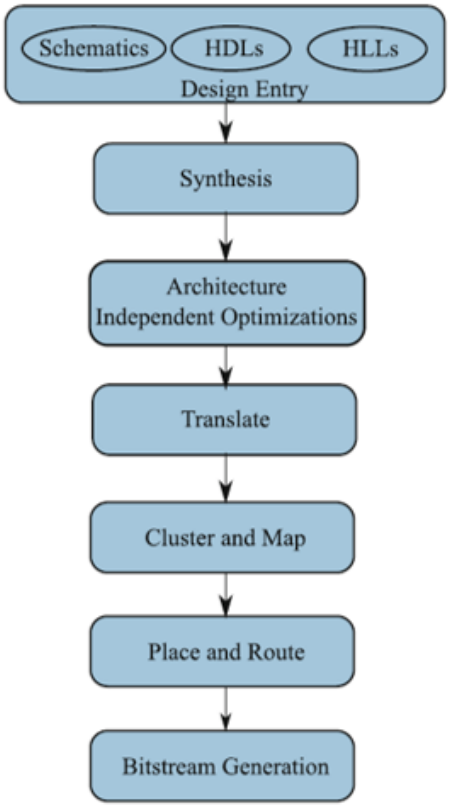


Figura 2.3: Fluxograma do FPGA [3].

para descodificar a saída.

Previamente os FPGAs eram somente usados como *glue logic* entre sistemas incompatíveis, mas agora com muito mais lógica pode ser implementada num FPGA. Daí os FPGAs serem usados cada vez mais como unidades independentes de computação ou lógica.

2.1.2 FPGAs como aceleradores de *Hardware*

Os FPGAs são uma plataforma de *hardware* reconfigurável que permitem a implementação de lógica e algoritmos. Sendo o seu *hardware* de granularidade fina[1], os FPGAs conseguem explorar o paralelismo inerente ao *hardware* enquanto ao mesmo tempo mantêm a reconfigurabilidade e programabilidade do *software*. Estas caraterísticas levaram os FPGAs a serem utilizados cada vez mais para aceleração de tarefas computacionalmente intensivas, como por exemplo, para a aceleração de algoritmos de processamento de imagem ([8], [9], [10], [11], [12], entre outros), e sobretudo para aplicações em tempo-real, onde as latências e os consumos energéticos são considerações importantes a ter em conta. No entanto, a natureza de granularidade fina do FPGA também traz os seus pontos negativos, um dos principais sendo a elevada complexidade de programação devido

[1] A granularidade pode ser descrita como uma proporção de computação para comunicação num programa paralelo. O paralelismo da granularidade fina implica a partição da aplicação em pequenas quantidades de trabalho realizado, o que conduz à um rácio de computação para comunicação baixo.[7]

ao baixo nível de abstracção. Nos FPGAs, ao contrário dos processadores convencionais, cuja arquitectura computacional fixa proporciona um nível alto de abtracção, para além da concepção do algoritmo propriamente dito, é exigida a concepção da arquitectura computacional, o que amplia bastante a complexidade do espaço de desenvolvimento.

Inúmeros estudos estão a ser realizados para implementar aceleradores de *hardware* para acelerar algoritmos de processamento de vídeo utilizando FPGAs. Nalguns desses estudos foram implementadas funções particulares de um algoritmo de processamento de vídeo usando lógica *hardware* do FPGA, e noutros estudos as funções foram implementadas usando os processadores *soft-core* que podem ser implementados nos FPGAs. Alguns exemplos que foram encontrados pelos autores de [13] são: [14], [15], [16], [17] e [18].

No geral, um acelerador de *hardware* é uma lógica de *hardware* dedicada, personalizada para realizar uma tarefa específica. Na maior parte das vezes, é utilizado para aliviar do processador geral a carga computacional de funções de processamento frequentemente utilizadas. O acelerador de *hardware* normalmente encontrado no ambiente do microprocessador seria a unidade do ponto flutuante e a unidade multiplicadora, cujo objectivo, nesses casos, é acelerar cálculos intensivos necessários, diminuindo assim os tempos de execução (exemplo: figura 2.4) [13]. Desta maneira, como o acelerador de *hardware* reduz o tempo exigido para a computação, também ajuda a reduzir os recursos exigidos para efectuar os tais cálculos. Para tirar partido dos benefícios do acelerador de *hardware*, os investigadores dos artigos acima mencionados, [14], [16] e [18], tentaram implementar o processamento de algoritmos de vídeo usando o *hardware* do FPGA, algumas das implementações usaram os processadores *soft-core* enquanto outros usaram unidades de processamento e execução personalizados. As arquitecturas implementadas não tiveram em conta a expansibilidade ou integração com outros sistemas, mas apesar disso, os resultados das implementações mostraram benefícios significativos da implementação dos algoritmos de processamento de vídeo usando lógicas do *hardware*.

Embora os aceleradores baseados em FPGA forneçam resultados promissores, o seu uso não é amplamente difundido, pois representam desafios significativos devido às limitações de ferramentas de desenvolvimento de aplicação e metodologias de design [19]. A construção de aceleradores é bastante trabalhosa e consome uma enorme quantidade de tempo. Os autores de [20] realizaram um estudo com o fim de investigar os principais desafios, identificar limitações das ferramentas FPGA existentes, e discutiram as limitações baseadas em várias fases de desenvolvimento de aplicação, nomeadamente, Formulação, *Design*, Translação e Execução.

Reiterando, os FPGAs são muitas vezes utilizados como aceleradores de aplicações computacionalmente intensivas, e uma fase crítica do desenvolvimento de aplicações em FPGAs é encontrar e mapear o modelo de computação apropriado. O primeiro passo na definição de modelos computacionais de alto nível baseados em FPGAs é considerar como os FPGAs obtêm o seu desempenho *High-Performance Computing (HPC)*. Na práctica, o enorme potencial de desempenho tem duas origens: paralelismo (é possível atingir

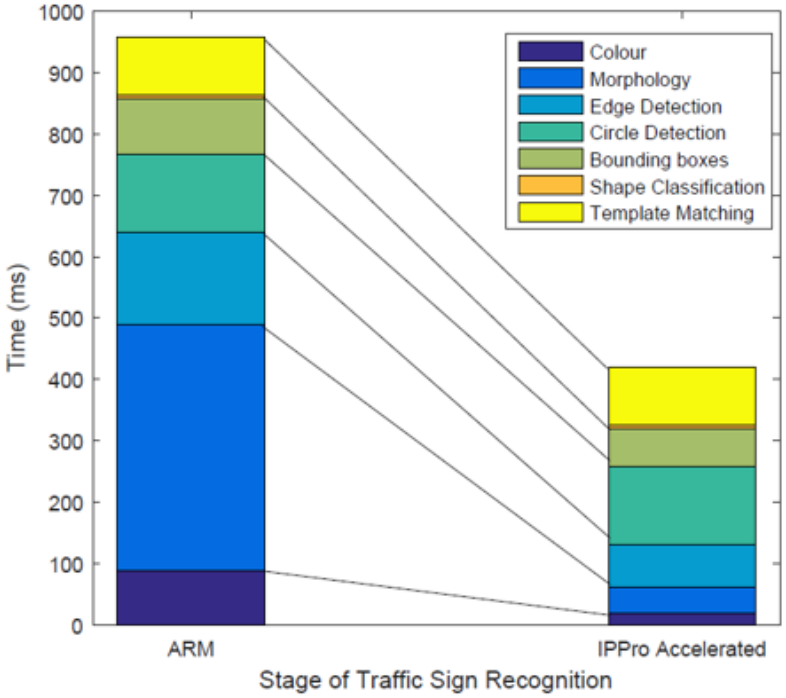


Figura 2.4: Comparação, por etapas, de reconhecimento de sinais rodoviários usando CPU ARM e modelo de aceleração proposto (IPPro) [12], que foi cerca de 9,6 vezes mais rápido.

um factor de 10000 para computações de baixo nível), e a carga útil por computação (o controlo é maioritariamente configurado na lógica, logo, os projectistas não necessitam de emular as instruções gerais, tais como, a indexação de arranjos e a computação de ciclos).

Computação do FPGA permite modelos com paralelismo de granularidade fina altamente flexível e operações associativas, tais como, *broadcast* e *collective response*. Em [21], os autores examinaram os atributos dos FPGAs que lhes permite obter desempenho para sistemas *High-Performance Computing* e *High-Performance Reconfigurable Computing*, e apresentaram cinco modelos computacionais de FPGAs.

Os modelos são vitais para muitas áreas da ciência e engenharia computacional, e vão desde os modelos formais usados na teoria da complexidade e na simulação até modelos intuitivos às vezes usados em arquitectura computacional e engenharia de *software*. Entende-se por modelo computacional, uma abstracção da máquina-alvo usada para facilitar desenvolvimento de uma aplicação. Esta abstracção permite ao desenvolvedor separar o *design* de uma aplicação, incluindo os algoritmos, da sua codificação e compilação. Em aplicações complexas, muitas vezes existem compromissos entre o esforço do programador, portabilidade e reutilização do programa, e o desempenho do programa. Quanto mais graus de liberdade tiver a arquitectura alvo, mais diversa é a selecção dos algoritmos, e menos provável é que um único modelo computacional possa permitir aos desenvolvedores de aplicações atingir os três compromissos simultaneamente. Os modelos que os autores de [21] identificaram como "úteis na fase inicial", são os seguintes: ***Streaming***, ***Associative Computing***, ***Standart hardware structure***, ***Functional Parallelism***.

Um bom modelo computacional permite-nos criar mapeamentos que tiram máximo partido de um ou mais níveis hierárquicos da memória do FPGA. Esses mapeamentos, tipicamente, contêm grandes quantidades de paralelismo de granularidade fina. Os elementos de processamento, muitas vezes, são conectados ou como poucos segmentos longos (com 50 etapas ou mais), ou como umas centenas de segmentos curtos. Outro factor crítico para um bom modelo FPGA é o tamanho do código, que por sua vez, se traduz em área de FPGA. Os melhores desempenhos, certamente, atingem-se quando é utilizado o FPGA inteiro, tipicamente através do paralelismo de granularidade fina. Por outro lado, se um único segmento não encaixar no *chip*, o desempenho pode ser fraco. Fraco desempenho também pode surgir em aplicações que têm muitas computações condicionais. Para um microprocessador, invocar diferentes funções provavelmente envolve pouca sobrecarga. Para o FPGA, no entanto, isto pode ser problemático porque cada função ocupa parte do *chip*, quer esteja em uso ou não. No pior caso, só uma fracção do FPGA está sempre em uso. Contudo, os projectistas podem manter utilização alta ao agendar as tarefas entre as funções e reconfigurar os FPGAs conforme necessário.

A vantagem de ter um bom modelo computacional, portanto não é para poupar tempo, mas sim para aumentar a qualidade do projecto. Neste respeito, o benefício é similar ao uso do modelo apropriado de computação paralela. Pode não demorar muito tempo para obter um sistema funcional com o uso dum modelo inapropriado, mas atingir bom desempenho pode ser impossível.

A memória é o maior factor limitante dos FPGAs para o seu uso ser generalizado em processamento de imagem de alto nível. Como os FPGAs têm recursos reduzidos de memória *on-chip*, o uso eficiente dos seus recursos é essencial para atingir metas de desempenho, tamanho e restrições de consumos energéticos. Os autores do artigo [22] investigaram a alocação dos recursos de memória *on-chip* com o fim de minimizar o uso de recursos e consumos energéticos, o que por sua vez, contribui para a realização de processamento de imagem de alto nível totalmente contido em FPGAs.

Na maioria dos casos, as aplicações de visão computacional implementadas em FPGAs são limitadas pelos requisitos de desempenho, potência e tempo real. Aplicações em tempo real do tipo *streaming* (ou seja, realizar processamento de imagem em transmissões de vídeo em tempo real) requerem tempos de aquisição, de processamento e de comunicação limitados que só podem ser alcançados, mantendo o poder computacional necessário, através da exploração do paralelismo de dados por blocos funcionais dedicados.

No entanto, como já foi referido mais acima, o maior fator limitante para o uso generalizado de FPGAs para aplicações complexas de processamento de imagens é a memória. Algoritmos que executam apenas operadores pontuais ou locais, como por exemplo, filtros de janela deslizante, são relativamente simples de implementar usando estruturas de *hardware*, como *buffers* de linha. No entanto, algoritmos complexos baseados em operações globais requerem que os *frames* completos sejam armazenados. Concretamente, alguns exemplos de aplicações essenciais para a área da segurança que requerem operações globais são as de detecção, identificação e rastreamento de objectos.

Uma abordagem possível para lidar com a memória reduzida dos FPGAs é refinar algoritmos de processamento de imagem para que estes possam operar em *frames* de tamanhos menores, para que esses possam estar contidos num FPGA. Os autores de [22] encontraram alguns estudos cujos algoritmos mantêm a sua robustez para imagens reduzidas, como é o exemplo de *Face Certainty Map* [23]. Outra solução é empregar esquemas de alocação de memória *on-chip* inteligentes para acomodar os *frames* inteiros, que requerem métodos para optimizar as configurações da memória *on-chip* a fim de maximizar o seu uso.

2.1.3 Reconfiguração Dinâmica e Parcial do FPGA

"Os *Field-Programmable Gate Array*s são classificados como dinamicamente reconfiguráveis (figura 2.5) se os seus circuitos de configuração de armazenamento integrados podem ser actualizados selectivamente, i.e., designados circuitos de configuração de armazenamento (e as correspondentes funções lógicas e interligações que controlam) podem ser alterados, sem perturbar as operações da lógica restante. Estes dispositivos podem então ser selectivamente reconfigurados enquanto ainda activos. A lógica resultante é referida como *Dynamically Reconfigurable Logic (DRL)*"[24]. Para um FPGA particular ser considerado dinamicamente reconfigurável, a definição implica que esse tem de ser capaz de efectuar reconfiguração parcial enquanto activo. No nível do sistema o módulo que contém múltiplos FPGAs pode ser classificado como dinamicamente reconfigurável se os seus FPGAs forem individualmente reconfiguráveis.

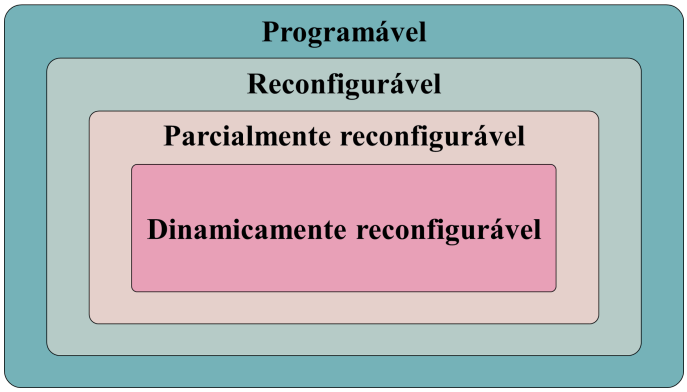


Figura 2.5: Classificação dos FPGAs de acordo com a sua configurabilidade (adaptado de [24]).

O que permite aos FPGAs atingir a sua reconfigurabilidade e flexibilidade é a sua composição que, conceptualmente, pode ser considerada por duas camadas distintas: camada de memória configurável e lógica *hardware*, como é apresentado na figura 2.6.

A reconfiguração dinâmica e parcial são as principais capacidades diferenciadores dos *Field-Programmable Gate Array*s. Os FPGAs evoluíram desde a sua utilização como *chips* para implementação de *glue logic* para plataformas de implementação de *System-on-Chip (SoC)* avançados de *software* e *hardware* combinados. À medida que as suas habilidades

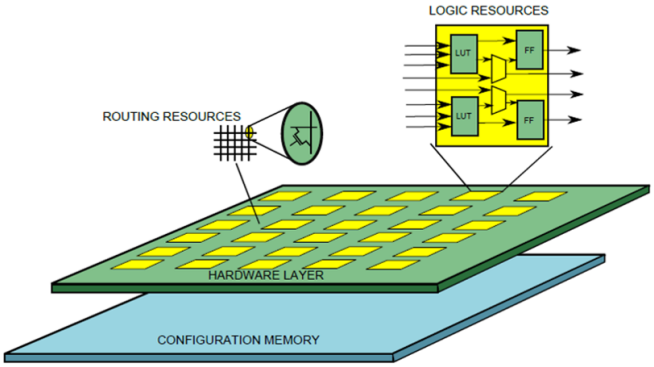


Figura 2.6: Arquitectura típica do FPGA composto por camada de memória configurável e camada de lógica *hardware* [25].

e tamanhos aumentaram, os FPGAs encontraram uso em diversos domínios, onde a sua reprogramação oferece uma vantagem distinta sobre as implementações em circuitos integrados de aplicação especifica, ou *Application Specific Integrated Circuit (ASIC)*. A capacidade ainda mais diferenciadora dos FPGAs é sua programação dinâmica, em que as suas funções são alteradas no momento de execução da aplicação, em resposta aos seus requisitos.

A reconfiguração parcial refere-se à modificação de uma ou mais partes da lógica FPGA enquanto as restantes partes não são alteradas. Repetidamente, os termos reconfiguração dinâmica e reconfiguração parcial têm sido usados indistintamente na literatura, mas eles podem ser diferentes. A operação de reconfiguração parcial pode ser estática ou dinâmica, o que significa que a operação de reconfiguração pode ocorrer enquanto a lógica do FPGA se encontra num estado de reinicialização (estático) ou num de execução (dinâmico).

Num estudo extenso sobre esta matéria, os autores de [25] apresentaram as principais vantagens da reconfiguração parcial, as características desejadas para uma plataforma parcialmente reconfigurável, as arquitecturas, ferramentas de simulação e implementação, entre outros temas. Também forneceram casos de uso e aplicações onde a reconfiguração parcial é benéfica.

A reconfiguração parcial pode trazer muitas vantagens para um projecto FPGA, nomeadamente, aumentar a lógica efectiva do *chip*, que permite uma aplicação maior poder ser contida num *chip* menor. A reconfiguração parcial tem tempo de reconfiguração reduzido quando comparada com uma reconfiguração completa, uma vez que este tempo é directamente proporcional ao tamanho do ficheiro de configuração, que por sua vez, é proporcional à área do *chip* a ser reconfigurado, significando que a reconfiguração pode ser aplicada em sistemas onde o tempo é um requisito crítico.

A reconfiguração parcial é benéfica em sistemas de *hardware* adaptativos, uma vez que a computação pode ser adaptada ao ambiente em mudança enquanto continua a processar dados. A reconfiguração parcial também tem a vantagem de reduzir a memória

necessária para os ficheiros de configuração, pois os ficheiros de configuração parciais são mais pequenos do que os de configuração completa, sendo isso especialmente benéfico para *embedded systems* com restrições de tamanho, custo e consumo energético.

Inicialmente, a reconfiguração dinâmica foi proposta para aumentar a capacidade lógica efectiva do FPGA e reduzir o tempo de reconfiguração, visto que os seus recursos criavam uma grande limitação na implementação de projectos grandes. As arquitecturas dinamicamente reconfiguráveis iniciais alargavam o número de contextos de configuração, o que permitia uma reconfiguração muito mais rápida e aumentava a capacidade lógica, como mostra a figura 2.7. Para FPGAs modernos, as motivações primárias para reconfiguração parcial são o compartilhamento de um único dispositivo físico entre vários utilizadores, mantendo as ligações de comunicação activas durante a reconfiguração do sistema, e aplicações adaptáveis com requisitos computacionais variados.

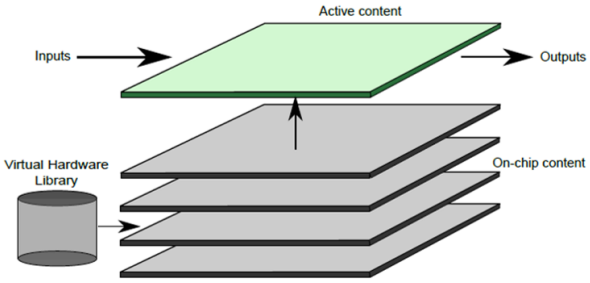


Figura 2.7: FPGAs multi-contexto aumentam a capacidade da lógica efectiva ao usar mais que um plano de configuração de memória [25].

Uma gama ampla de aplicações que exploram a reconfiguração parcial foi discutida em literatura, e podem ser classificadas com base nas características específicas da reconfiguração a serem exploradas, como a adaptabilidade, a redução de sobrecarga, redução de custos de sistemas, melhoria de fiabilidade e computação em *hardware*.

As tendências para sistemas mais autónomas em áreas como automóvel, comunicação, e aplicações aeroespaciais também apresentam uma oportunidade adequada para projectos de sistemas de reconfiguração parcial. Para verdadeiramente tornar convencional o *design* do sistema de reconfiguração reconfigurável, os autores de [25] acreditam que existe uma série de desafios de investigação que necessitam de atenção:

- No nível da arquitectura, como melhor suportar a ideia de múltiplos aceleradores carregáveis com realocação e reconfiguração simples, especialmente em dispositivos comerciais.

- Nos métodos, como juntar as investigações realizadas até agora para superar as limitações existentes, e abstrair os aspectos de *hardware* por meio da automação a partir de descrições de alto nível.

- Em estruturas e aplicações, encontrar melhores maneiras, orientadas para aplicações, de descrever sistemas adaptativos que podem ser automaticamente mapeados para implementações de reconfiguração parcial.

- Ao nível da gestão, aprimorar a abstração para permitir o carregamento e descarregamento de novos configurações, similares aos carregamentos e descarregamentos dinâmicos de módulos de *software*.

- Por fim, explorar como os sistemas auto-adaptativos podem ser concebidos de forma autónoma que combinam a capacidade de reconfiguração com inteligência e a habilidade de adaptar as capacidades *bitstream*.

2.1.4 Comparação de plataformas: FPGA vs GPU vs CPU

O desempenho é um critério muito importante a ter em conta quando se pretende criar uma aplicação de processamento de imagem, porque com a rápida evolução da tecnologia da imagem digital aumenta também a carga e esforço computacional, que se devem ao contínuo crescimento do tamanhos das imagens e da complexidade dos algoritmos de processamento de imagem. A concorrência entre as plataformas qualificadas para executar implementações de processamento de imagem é cada vez mais forte, e como um número substancial dessas aplicações têm paralelismo inerente, três plataformas reconfiguráveis são sempre comparadas: *Field-Programmable Gate Array (FPGA)*, *Graphics Processing Unit (GPU)* e *Central Processing Unit (CPU)*.

Estas plataformas têm demonstrado altos desempenhos porém cada uma consegue atingi-los em situações particulares e devido às suas propriedades diferenciadoras [26]. Em [27] o autor realizou uma pesquisa que resume com detalhe as principais vantagens do FPGA para atingir altos desempenhos no processamento de imagem. Elas são, primeiramente, a sua flexibilidade que torna possível personalizar e optimizar os circuitos para cada aplicação específica, e por outro lado, o seu número elevado de bancos de memória disponíveis que podem ser acedidos em paralelo. A estrutura do FPGA permite paralelismo temporal e espacial, o FPGA consegue executar múltiplas janelas de imagem em paralelo e múltiplas operações dentro duma janela também em paralelo. Além disso, o FPGA tem a capacidade de ter entradas-saídas em paralelo, ou seja, consegue ler da memória, processar os dados e escrever em memória simultaneamente. No entanto, para tirar melhor partido do paralelismo do FPGA, os projectistas têm de minimizar o número de operações e acessos à memória em consequência da sua baixa frequência operacional. Os processadores mais recentes têm vindo a aumentar o número de núcleos que suportam instruções SIMD (*Single Instruction, Multiple Data*) melhoradas para efectuar processamento paralelo. Apesar do paralelismo ser limitado nestas instruções do CPU, a sua frequência operacional é muito alta e em combinação com o grande tamanho da sua memória *cache*, suficiente para permitir o armazenamento de todos os dados da imagem, produzem bons resultados. O aspecto problemático das instruções SIMD é a sua programação, que pode

tornar-se complicada para atingir desempenhos desejáveis. A frequência operacional da GPU (*Graphics Processing Unit*) é mais rápida que a do FPGA, mas um pouco mais lenta que a do CPU, e como as GPUs suportam um número bastante superior de núcleos que correm em paralelo, quando comparado com CPU, o seu desempenho pico acaba também por ser superior. Para se explorar ao máximo o desempenho da GPU é necessário lidar com as seguintes limitações: os seus núcleos são agrupados e a transferência de dados entre os mesmos é muito lenta, mais, o tamanho da memória local disponibilizada à cada agrupamento é muito pequeno.

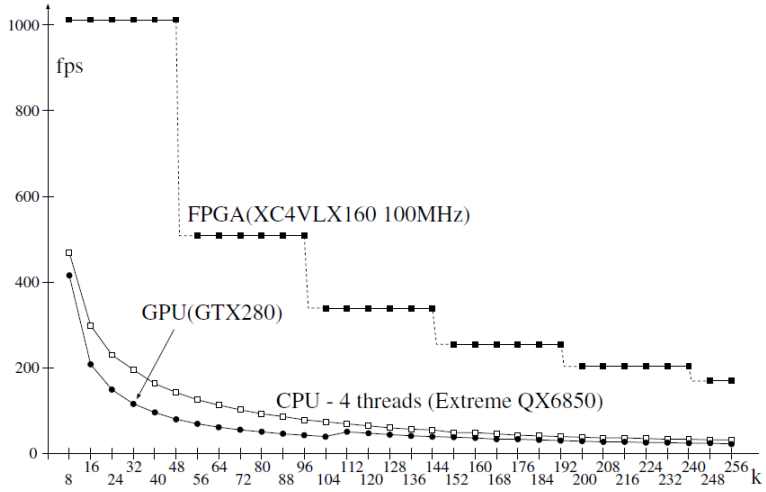


Figura 2.8: Desempenhos do algoritmo de agrupamento K-means (média de 1000 execuções) [26]

Em [26] foi comparado o desempenho dum FPGA (*Xilinx XC4VLX160*) com uma GPU (*XFX GeForce 280 GTX 1024MB DDR3 padrão, versão CUDA 2.1*) e CPU (*Intel Core 2 Extreme QX6850 (3GHz, 4 núcleos, 8MB L2 memória cache), compilador Intel C++ 10.0*) usando três problemas simples de processamento de imagem, mais especificamente, um filtro bidimensional não-separável, visão estéreo e o método de agrupamento K-means. A GPU conseguiu demonstrar o seu melhor desempenho na aplicação do filtro bidimensional, onde o filtro pode ser aplicado à píxeis individuais da imagem sem ser necessário o uso de variáveis partilhadas. Nos algoritmos mais sofisticados, a GPU não consegue competir com o FPGA, e mesmo com o CPU, porque não consegue executar algoritmos que usam matrizes partilhadas em consequência da sua pouca memória local. A figura 2.8 revela os resultados obtidos na execução de uma iteração[2] do algoritmo de agrupamento K-means em imagens a cores com tamanho 768×512 píxeis. Na implementação da GPU só foram disponibilizados 768 *threads*, com cada *thread* a processar 512 píxeis sequencialmente, para simplificar a atribuição de dados aos mesmos. O desempenho do FPGA, mesmo sendo limitado pelo seu tamanho e largura de banda da memória, é entre 7 e 12 vezes superior ao do CPU. O autor em [28] usou os mesmos testes para comparar os

[2] O desempenho do algoritmo de agrupamento K-means depende do número de iterações, que varia de imagem para imagem. O tempo de uma iteração é determinado pelo tamanho da imagem e pelo K.

desempenhos de FPGAs e CPUs e obteve resultados muito similares. O FPGA demonstrou ganhos entre 5 a 15 vezes superior ao CPU, e onde, mais uma vez, é realçado o facto de o desempenho do FPGA encontrar-se limitado pelo seu tamanho, evidenciado na figura 2.9, onde XC4VLX160 é duas vezes mais rápido que XC2V6000 porque o seu tamanho é duas vezes superior. O desempenho do FPGA ainda pode ser melhorado se as imagens forem divididas em sub-imagens e processadas em paralelo, como acontece nas execuções multi-fio nos processadores.

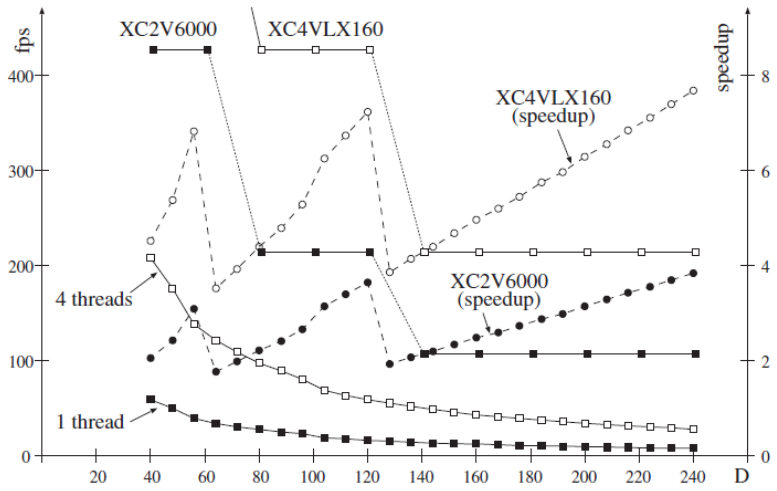
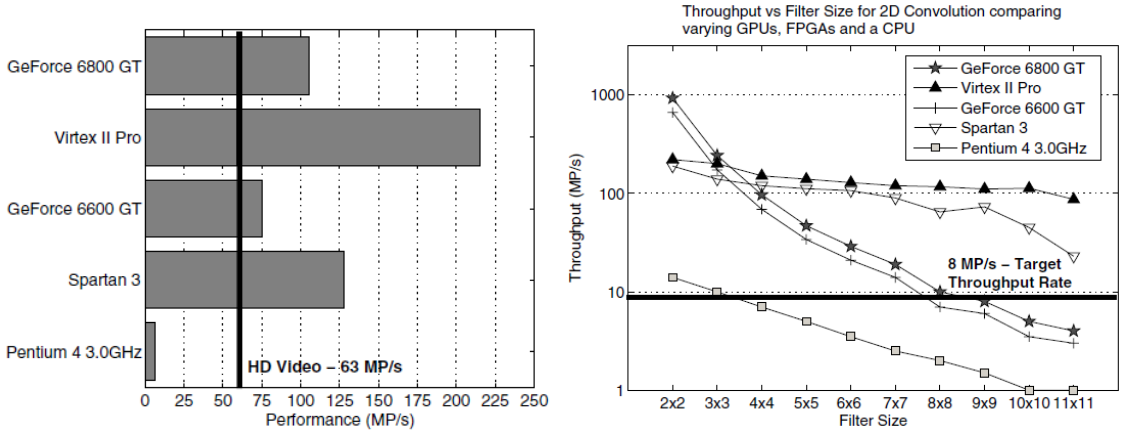


Figura 2.9: Desempenho do algoritmo visão estéreo (média de 1000 execuções) [28]

O autor do [29] comparou os desempenhos de FPGAs e GPUs com a métrica MP/s (*Million pixels per second*), relevante para processamento de imagem em tempo real, exemplificado através dos algoritmos de correcção de cor primária e convolução 2D. O estudo do caso da convolução 2D foi escolhido pelo motivo de se focar mais em acessos à memória, que é considerado problemático para as GPUs, e o estudo de caso da correcção de cor por ser separável em módulos com várias operações, deste modo, puderam ser avaliados os desempenhos de cada núcleo de *hardware*. As plataformas seleccionadas, GPU (*Nvidia's GeForce 6800 GT*) e FPGA (*Xilinx's Virtex II Pro*), no momento do estudo, eram consideradas topo de gama das suas categorias, e GPU (*GeForce 6600 GT*) e FPGA (*Spartan 3*) de reduzido custo. Foi também escolhido o processador Pentinum 4 3.0GHz para servir de referência. A figura 2.10a exibe os resultados dos desempenhos obtidos no teste da correcção de cor onde, tanto as GPUs como os FPGAs atingiram a meta dos 63MP/s[3], no entanto, ambos os FPGAs tiveram execuções mais rápidas que as GPUs, o que demonstra a superioridade do paralelismo e *pipelining* flexível do FPGA sobre o número fixo de *pipelines* paralelas da GPU. Além disso, fica evidenciada a vantagem financeira que os FPGAs podem ter sobre as GPUs e CPUs na prototipagem de aplicações de processamento de imagem sem abdicar de desempenho. Por sua vez, as GPUs demonstraram o benefício e a eficiência do seu conjunto de instruções optimizado, configurado até 16 *pipelines*, sobre

[3]Objectivo: vídeo de alta definição (1920×1080 a 30 imagens por segundo).

o CPU, onde, embora o CPU tenha a sua frequência de funcionamento 8,5 mais rápida que a GPU, obteve tempos de execução 18 vezes mais lentos. Para o caso da convolução 2D foram testados vários filtros $n \times n$ e o objectivo foi atingir 8MP/s[4]. As GPUs só conseguiram o resultado pretendido até o tamanho do filtro 7×7, que é mais do que que o CPU consegui, 4×4, mas não conseguiu acompanhar os desempenhos dos FPGAs. Os FPGAs também demonstraram mais consistência nos resultados quando comparados com as GPUs devido ao seu paralelismo flexível, *pipelining* e a transmissão de dados, o que verifica o acesso à memória ineficiente da GPU.



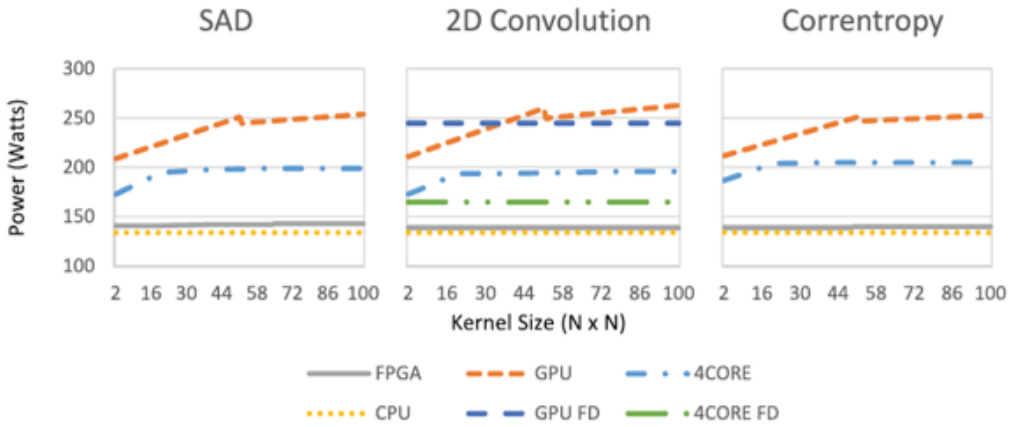
(a) Comparação de desempenhos dos FPGAs,(b) Desempenho vs Tamanho do filtro para ConGPUs e CPU para Correcção de Cores Primárias.volução 2D comparando os FPGAs, GPUs e CPU.

Figura 2.10: Resultados dos estudos de caso em [29].

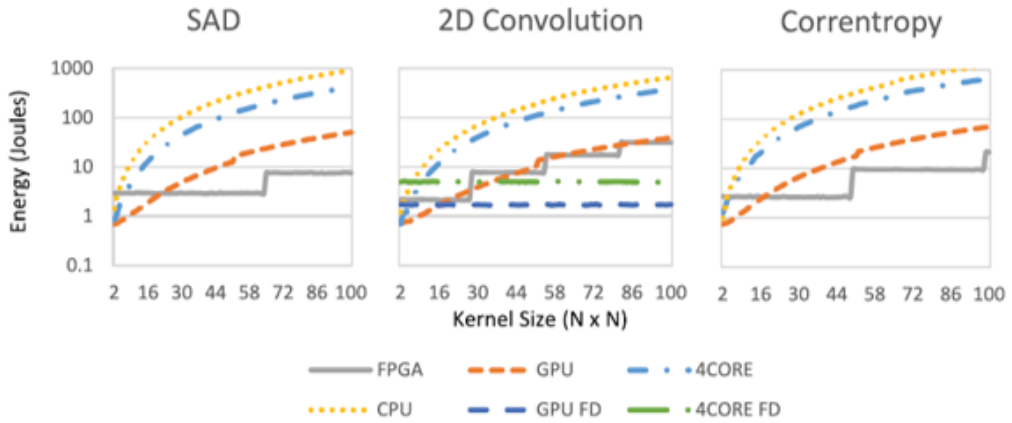
Por fim, a última métrica a ser comparada são os consumos energéticos das plataformas em questão. Em [30], o autor no seu estudo executou três algoritmos de janela deslizante, SAD (*Sum of Absolute Differences*), convolução 2D e *Correntropy* nas seguintes plataformas: FPGA (*GiDEL ProcStar IV*), GPU (*NVIDIA GeForce GTX 560*) e o CPU (*Intel Xeon Core i7-960 3.2GHz*). Para avaliar o consumo energético, foi medida a potência necessária para cada implementação de algoritmos para um subconjunto de tamanhos de máscara, e, uma vez obtidas as medidas para esses subconjuntos, foi usada interpolação linear para estimar consumos para outros tamanhos de máscara. As potências estáticas dos sistemas inactivos consumidos pelo FPGA, GPU e CPU foram 119W, 105W e 90W respectivamente. Para medir os consumos totais, foram removidos todos os componentes de *hardware* desnecessários à configuração, e em cada ponto de dados foi executado num ciclo uma funcção de janela deslizante por um mínimo de 15 segundos. Na figura 2.11b pode ser observado a energia consumida no processamento de um *frame* em cada implementação. Os resultados foram obtidos ao multiplicar o tempo de execução de cada implementação pelas potência obtidas na figura 2.11a. Os comportamentos dos consumos energéticos nos algoritmos SAD e *Correntropy* são muito similares, onde para máscaras

[4]Objectivo: vídeo com tamanho da imagem 512×512 a 30 imagens por segundo.

de tamanhos reduzidos, até 20×20, a GPU é a plataforma mais eficiente energeticamente, mas para máscaras com tamanhos superiores a 25×25 o FPGA consome até uma ordem de magnitude menos energia que qualquer outro dispositivo testado. Em convolução 2D a GPU demonstrou ser a plataforma com mais eficiência energética.



(a) Potência do sistema para SAD, convolução 2D e *Correntropy* em imagens 1280×720.



(b) Energia consumida no processamento de uma imagem em SAD, convolução 2D e *Correntropy* em imagens 1280×720. No eixo y é utilizada a escala $\log_{10}$.

Figura 2.11: Resultados dos estudos realizados em [30].

No estudo [31] o autor estudou o comportamento energético duma GPU e FPGA em relação a um CPU. No método experimental foram processados 1000 *frames* em escala de tons de cinza e resolução 1920×1080 para algoritmos de seis categorias de *kernels* de visão diferentes, que incrementavam a sua complexidade. Foram medidas as energias dinâmicas dos *kernels*, excluindo a energia estática que alimentava as plataformas, e a taxa de fotogramas máxima atingida, calculando assim o rácio de redução energia por *frame* (mais alto é melhor). Na tabela 2.1 ficam visíveis os resultados experimentais obtidos em cada algoritmo, e infere-se que tanto a GPU como o FPGA demonstram mais eficiência energética quando comparados com o CPU, mais, a medida que aumenta a complexidade dos algoritmos, de cima para baixo, aumenta também a lógica programável e os recursos exigidos às plataformas, onde o FPGA prova lidar melhor com os acessos à memória,

dependência de dados e condições de ramificação que a GPU, pois apresenta rácios de redução de energia/*frame* superiores.

	CPU	GPU	FPGA
Input Processing	1	**1,79**×	1,41×
Image Arithmetic	1	**3,19**×	2,93×
Image Filters	1	3,17×	**3,89**×
Image Analysis	1	2,34×	**5,67**×
Geometric Transform	1	10,3×	**16,6**×
Features/ OF/ /StereoBM	1	7,44×	**22,3**×

Tabela 2.1: Rácios de redução Energia/*Frame* (referência CPU) [31].

2.2 Detecção de Faixa de Rodagem Automóvel

Nesta secção serão abordadas estudos realizados, técnicas e plataformas existentes para a detecção de faixa de rodagem, bem como algumas limitações e desafios identificados na literatura.

2.2.1 Trabalhos Relacionados

A detecção de faixa de rodagem é um elemento fundamental da maior parte dos actuais sistemas avançados de assistência ao motorista, ou *Advanced Driver Assistance System (ADAS)*. As grandes empresas de automóveis desenvolveram os seus próprios produtos de detecção e seguimento de faixas de rodagem conseguindo obter resultados significativos tanto em aplicações do mundo real como de carácter de investigação. Quase todas as gamas de produtos de detecção de faixas de rodagem para assistir o condutor usam técnicas baseadas na visão computacional uma vez que as marcações da faixa são pintadas na estrada para percepção visual humana. A utilização de técnicas baseadas na visão detecta faixas recorrendo à câmaras e pode impedir o motorista de efectuar mudanças de faixas indesejadas, portanto, a precisão e robustez são duas propriedades essenciais para este tipo de sistemas.

Uma grande parte de sistemas de detecção de faixa baseados na visão são tipicamente desenvolvidos com técnicas de processamento de imagem dentro de *frameworks* semelhantes. Com o desenvolvimento de dispositivos de computação cada vez mais rápidos e desenvolvimento de teorias avançadas de aprendizagem automática, como aprendizagem profunda, os problemas de detecção de faixa podem ser resolvidos de formas mais eficientes usando procedimentos de detecção de ponta a ponta. No entanto, o desafio crítico que os sistemas de detecção de faixa de rodagem têm de enfrentar são os requerimentos de alta confiabilidade e o seu funcionamento em diversas condições.

A arquitectura típica dos sistemas de detecção de faixa de rodagem automóvel baseados em visão, descrita em [2], têm três procedimentos principais, que são: pré-processamento

de imagem, detecção de faixa, que compreende também o aspecto mais importante do sistema de detecção de faixa de rodagem, a extração de características e ajuste do modelo, e o seguimento de faixa (ou *lane tracking*). Esta arquitectura pode ser observada na figura 2.12. Os procedimentos mais comuns na etapa de pré-processamento incluem a selecção de região de interesse (ou *Region Of Interest (ROI)*), detecção de ponto de fuga, conversão da cor da imagem para a escala de cinza ou outro formato de cor diferente, remoção de ruído e desfoque, mapeamento de perspectiva invertida (ou *Inverse Perspective Mapping (IPM)*), segmentação, entre outras. Destas tarefas, destaca-se a determinação da ROI, que geralmente é a primeira etapa a ser realizada, pois permite aumentar a eficiência computacional e reduzir o número de falsas detecções de faixa de rodagem. A região de interesse pode ser uma selecção aproximada da parte inferior da imagem de entrada, ou determinada dinamicamente de acordo com as faixas detectadas. Também pode ser determinada mais eficientemente com conhecimento prévio das áreas rodoviárias [32], [33].

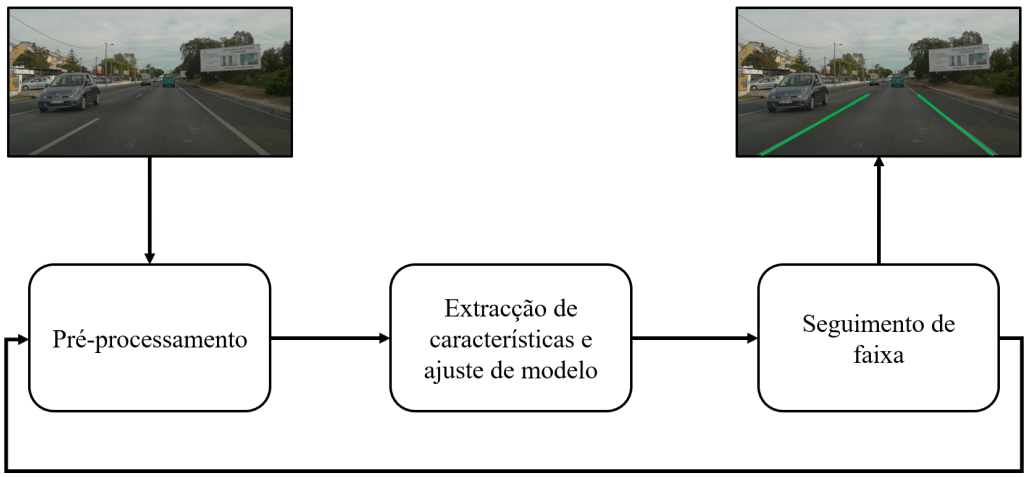


Figura 2.12: Arquitectura geral de um sistema de detecção de faixa de rodagem automóvel (adaptado de [2]).

Uma vez que as imagens de entrada foram pré-processadas, podem ser extraídas as características dos bordos, contornos e as cores. O algoritmo da transformada de Hough, que usa as imagens de contornos, é um dos algoritmos mais usados para detecção de faixas de rodagem automóvel em estudos de investigação. No entanto, este método é projectado para detectar linhas retas e não é muito eficiente na detecção de faixas curvas. Frequentemente, as faixas curvas podem ser detectadas com base em técnicas de ajuste de modelo, tais como *RANdom SAmple Consensus (RANSAC)*. RANSAC testa recursivamente a pontuação de ajuste do modelo para encontrar os parâmetros ideais do modelo, logo, ele tem uma grande capacidade de lidar com recursos atípicos. Por fim, após à detecção bem-sucedida das faixas, as posições da faixa podem ser determinadas com algoritmos de rastreamento, como filtro Kalman ou filtros de partículas, para refinar os resultados da detecção e prever as posições da pista de maneira mais eficiente.

Os autores do artigo [2] realizaram levantamento bibliográfico extenso sobre os avanços na detecção de faixas automóveis baseados em visão computacional. Reviram estudos prévios com base em três aspectos: algoritmos de detecção de faixa, integração, e métodos de avaliação.

Nesse artigo consta-se que a detecção de faixa automóvel baseada em visão, de maneira geral, pode ser classificada em duas categorias: baseado em características (*feature-based*) e baseado em modelos (*model-based*). Os métodos baseados em características para detecção de faixas de rodagem dependem das marcas particulares da estrada, como as suas cores, texturas e bordas da faixa.

Em geral, os métodos baseados em características têm melhores eficiências computacionais e são capazes de detectar com precisão as faixas quando as marcações dessas são claras. No entanto, desvantagens desses métodos incluem menor robustez para lidar com sombras e más condições de visibilidade quando comparados com o métodos baseado em modelos.

Os métodos baseados em modelos geralmente assumem que as faixas podem ser descritas com um modelo específico, como um modelo linear, um modelo parabólico ou vários tipos de modelos *spline*. Além disso, são necessárias algumas suposições sobre a estrada e as faixas, como por exemplo, se têm uma superfície plana ou não. Os modelos *spline* revelaram-se como mais populares em estudos anteriores, uma vez que esses modelos têm flexibilidade suficiente para recuperar quaisquer formas das faixas curvas. O algoritmo RANSAC é a maneira mais popular de estimar iterativamente os parâmetros do modelo da faixa de rodagem automóvel.

De modo geral, os métodos baseados em modelos são mais robustos do que os métodos baseados em características por causa do uso de técnicas ajuste de modelos. No entanto, estes métodos, tipicamente, requerem mais custos computacionais, uma vez que RANSAC não tem limites superiores no número de iterações. Além disso, as implementações dos métodos baseados em modelos são mais complicadas quando comparadas com sistemas baseadas em características.

Existem também autores que se focam na investigação de métodos de detecção de faixa que usam algoritmos de aprendizagem automática e aprendizagem profunda. Nestes casos, *Convolution Neural Network (CNN)* demonstra-se como uma das abordagens mais populares. CNN proporciona propriedades como alta precisão de detecção, aprendizagem automática de características de faixa e reconhecimento de ponta a ponta. Os autores de [34] conseguiram aumentar dramaticamente a precisão de detecção de faixa, de 80% para 90% quando comparado com os métodos tradicionais de processamento de imagem.

Pode afirmar-se que os algoritmos de aprendizagem automática ou os algoritmos inteligentes aumentam a precisão detecção da faixa significativamente e fornecem muitas técnicas e arquitecturas de detecção eficientes. Apesar desses sistemas geralmente requererem mais custos computacionais e necessitarem de muitos dados de treino, eles acabam por ser mais poderosos que os métodos convencionais. Daí a esperança de que num futuro próximo serão desenvolvidos muitos novos, eficientes e robustos métodos de detecção de

faixa, com menos treino e computação.

Os autores de [2] apresentaram alguns estudos de detecção de faixa automóvel representativos de várias áreas de estudo, que podem ser observados na tabela 2.2. Nessa tabela, a coluna de pré-processamento regista os métodos de processamento de imagem usados na literatura investigada. A coluna de integração indica os métodos de integração usados no estudo, que podem ser de diferentes níveis de integração, nomeadamente, ao nível de algoritmo, sistema e sensores. Na coluna de avaliação os *frame image* e *visual assessment* indicam que o algoritmo proposto foi apenas avaliado com imagens estáticas e com métodos de avaliação visual, sem qualquer comparação com *ground truth*. Geralmente, um sistema de detecção de faixa robusto e preciso combina algoritmos de detecção e seguimento (*tracking*). Mais, a maior parte de sistemas avançados de detecção de faixa cooperam com outros sistemas de detecção de objectos ou sensores para gerar uma rede mais compreensiva de detecção.

Tabela 2.2: Sumário de vários sistemas de detecção de faixa ([2]).

Ref.	Pré-processamento	Detecção de faixa	Seguimento de faixa	Integração	Avaliação	Comentários
[35]	IPM, limiar adaptativo	Filtros morfológicos	-	Câmara estéreo para detecção de faixa e obstáculos	*Frame images* e avaliação visual	As faixas são principalmente detectadas pelas características de cor que pode ser menos robusto em situações de mudança de iluminação
[36]	Extracção de textura de marcações da faixa	*Scanning line*	Filtro de Kalman	Câmara	*Frame images* e avaliação visual	Combinação de dois algoritmos de detecção de nível inferior
[37]	IPM, filtro direccionável	RANSAC	-	Câmara	Realizado sobre os dados KITTI usando as taxas de positivos correctos e falsos positivos	Algoritmo robusto à sobra, e fluxo óptico foi usado para construir sistema de reconhecimento de desvio de faixa
[38]	Ridge feature	RANSAC	-	Câmara	Análise quantitativa	Os dados *ground truth* de faixa sintetizados são gerados com parâmetros geométricos conhecidos
[39]	Detecção de bordas	Transformada de Hough	-	Câmara	*Frame images* e avaliação visual	A estrada é dividida em campo próximo e distante com modelo recto e curvo
[40]	Detecção de ponto de fuga, Canny	Detecção de ponto de controlo	-	Câmara	*Frame images* e avaliação visual	O modelo *B-snake* proposto é robusto à variação de sombra e iluminação

Continua na próxima página

Tabela 2.2 – *Continuação da página anterior*

Ref.	Pré-processamento	Detecção de faixa	Seguimento de faixa	Integração	Avaliação	Comentários
[41]	Transformação para o espaço de cores YCbCr, detecção de ponto de fuga	Detecção de ponto de viragem	-	Integração de faixa e veículo usando uma única câmara	*Frame images* e avaliação visual	Veículos que tenham a mesma cor que as faixas são distinguidos pela forma, tamanho , e informação de movimento
[42]	IPM, filtro Gaussiano	Transformada de Hough, RANSAC	-	Câmara	Análise quantitativa com o dados Caltech	O método proposto é robusto à sombras e curvas mas pode ser influenciado pelas passadeiras ou pinturas de estradas
[43]	ROI, geração de imagem artificial	CNN	-	Duas câmaras laterais viradas para a estrada	Avaliação de distância de nível de píxel	Procedimento de reconhecimento de faixa ponta a ponta que pode ser aplicado em tempo real
[44]	Desfoque temporal, IPM, limiar adaptativo	RANSAC	Filtro de Kalman	Câmara	Análise quantitativa e avaliação visual	ALD 2.0 proposto é usado para rotulagem eficiente de *ground truth* vídeo
[45]	ROI, IPM	CNN, RNN	-	Câmaras	Análise quantitativa com curva ROC	RNN proposto usa *long-short-termmemory* que pode capturar as estruturas da faixa durante um período de tempo nas sequências de vídeo
[46]	IPM, transformada *Top-Hat*, gradiente vertical Prewitt, limiar adaptativo	Transformada Probabilística Progressiva de Hough	-	IMU, GPS, Lidar e câmaras	Avaliação visual e taxa de detecções correctas	A detecção das marcas de faixa só é realizado depois de detectada a área ideal baseado em sensor

Continua na próxima página

Tabela 2.2 – *Continuação da página anterior*

Ref.	Pré-processamento	Detecção de faixa	Seguimento de faixa	Integração	Avaliação	Comentários
[47]	Filtro mediano, Extracção *ground plane*	Segmentação de faixa	-	Câmara e Lidar	Avaliação visual e taxa de detecção correcta	A posição da faixa detectada com visão e Lidar são fundidos com um esquema de voto
[48]	Limiar dinâmico, detecção de bordas Canny	Transformada de Hough e método dos mínimos quadrados	Filtro de Kalman	Câmara, IMU, Lidar, e GPS	Critério espacial e de inclinação para avaliação em tempo real e erro de média absoluto com posição *ground truth*	Método de detecção e medição lateral de deslocamento robustos baseados em câmaras e Lidar
[49]	Layered ROI, filtro direccionável	Transformada de Hough	Filtro de Kalman	Radar e câmara	Avaliação visual e métricas de taxa de detecção correcta	ROI adaptativo faz a detecção de faixa robusta perto de outros carros e marcas de estrada
[50]	IPM, filtro direccionável	RANSAC	Filtro de Kalman	Câmara	Múltiplas métricas em *frames* escolhidos	Método robusto em situações de trânsito com detecção e localização aprimorada de veículos à sua volta

2.2.2 Limitações e Desafios actuais

Na última década vários sistemas comerciais *Advanced Driver Assistance System* de detecção de faixa de rodagem automóvel têm sido amplamente estudados e implementados com êxito. Devido ao baixo custo dos dispositivos de câmara e ao amplo conhecimento de processamento de imagem, inúmeros estudos de investigação que usam algoritmos baseados em visão podem ser encontrados. Embora esses sistemas sofram com a variação de iluminação, sombras e mau tempo, ainda são amplamente adoptados e continuarão a dominar os mercados ADAS. As principais tarefas de um algoritmo de sistema de detecção de faixa de rodagem automóvel é ser preciso e robusto. As questões de robustez são o aspecto chave que determinam se um sistema pode ser aplicado na vida prática ou não. O grande desafio para os sistemas futuros baseados em visão é manter uma medição e detecção de faixa de rodagem estável e confiável sob condições de trânsito e condições climatéricas adversas.

Um método eficiente para lidar com estes problemas é usar técnicas de integração e fusão de vários sistemas. Um sistema de detecção de faixa de rodagem apenas baseado em visão é limitado para lidar com diferentes estradas e situações de tráfego. Portanto, é necessário preparar um sistema de *back-up* que possa enriquecer a funcionalidade do ADAS, um sistema que possa ser construído ou baseado em integração, quer seja maneira baseada em algoritmo, sistema, ou integração de nível de sensor. A integração do algoritmo é a escolha com o menor custo e de implementação mais fácil. Integração ao nível de sistema combina sistema de detecção com outros sistemas de percepção, como por exemplo, detecção de veículos à sua volta, para melhorar a precisão e robustez do sistema. Estes métodos de integração ainda são dependentes de sistemas de visão de câmara e têm as suas inevitáveis limitações. Por outro lado, a integração ao nível do sensor, é a maneira mais confiável de detectar faixas de rodagem em diferentes situações.

Por fim, outro desafio dos sistemas de detecção de faixa de rodagem automóvel é encontrar, ou definir um sistema de avaliação mais universal que possa verificar o desempenho do sistema. Hoje em dia, um problema comum é a falta de avaliações comparativas (*benchmarks*) públicas. Além disso, não existem métricas de avaliação padrão que possam ser utilizadas para avaliar de forma abrangente o desempenho do sistema em relação à sua precisão e robustez.

2.3 Conceitos

Nesta secção serão apresentados vários conceitos que foram utilizados na arquitectura proposta na secção 4.1.

2.3.1 Redução de ruído: Filtro Gaussiano

A aplicação do filtro Gaussiano, ou, desfoque Gaussiano é uma técnica frequentemente aplicada em etapas de pré-processamento de imagem, cujo objectivo é suavizar a imagem,

reduzir os seus níveis de ruído, o que por sua vez, melhora os resultados dos algoritmos subsequentes. Como o passo seguinte é aplicar um algoritmo de realce de contornos, maior parte dos quais são sensíveis ao ruído, e como comparativamente ao filtro de média, outra técnica de suavização, preserva melhor os contornos da imagem original, aplicar o filtro Gaussiano é a escolha natural.

Em processamento de imagem isto é alcançado ao efectuar computacionalmente uma convolução entre uma máscara de convolução, ou *kernel*, e a imagem. O *kernel* de um filtro Gaussiano é composto por uma conjunto de coeficientes cujos pesos se aproximam à forma de uma Gaussiana, expressão 2.1, nas horizontais, verticais e diagonais. Os valores absolutos são uma opção do programador embora devam ser mantidos pequenos de forma a evitar erros de cálculo significativos.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.1}$$

A convolução, em processamento de imagem, é um método espacial de acondicionamento de imagem que calcula o valor de cada píxel com base em operações efectuadas sobre os valores de uma vizinhança local, cujos pesos é indicado pelo *kernel*. A expressão 2.2 é a expressão geral da convolução, onde *g(x, y)* é a imagem filtrada, *f(x, y)* é a imagem original e ω é o *kernel* do filtro.

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x+dx, y+dy) \tag{2.2}$$

A convolução da matriz tipicamente necessita de ter acesso a valores de píxeis fora da imagem. O processamento das vizinhanças torna-se problemático nas regiões limite da imagem, onde não existem valores. As soluções passam por: 1) ou, não processar as margens, 2) ou, prolongar a imagem através de vários métodos, como por exemplo, duplicar as linhas e colunas necessárias, ou fazer o espelho das linhas e colunas necessárias (ver figura 2.13), entre outras.

a	a	a	b	c	c	c
a	a	a	b	c	c	c
a	a	a	b	c	c	c
d	d	d	e	f	f	f
g	g	g	h	i	i	i
g	g	g	h	i	i	i
g	g	g	h	i	i	i

e	d	d	e	f	f	e
b	a	a	b	c	c	b
b	a	a	b	c	c	b
e	d	d	e	f	f	e
h	g	g	h	i	i	h
h	g	g	h	i	i	h
e	d	d	e	f	f	e

Figura 2.13: Métodos de prolongamento da imagem, Duplicação e Espelho das margens respectivamente.

2.3.2 Realçe de contornos: Filtro Sobel

O filtro de Sobel é um operador frequentemente aplicado no processamento de imagem que tem como objectivo realçar os contornos da imagem original. Os contornos da imagem são encontrados nas partes onde os valores das intensidades alteram rapidamente [51], ou seja, onde existe maior variação de claro para escuro. No cálculo, a taxa de variação instantânea é representada pela derivada.

As derivadas de primeira ordem em processamento de imagem são implementadas com o uso da magnitude do gradiente. O gradiente de uma imagem f nas coordenadas (x, y) é definido como uma matriz coluna de duas dimensões:

$$\nabla f = G[f(x,y)] = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{2.3}$$

A aplicação do filtro de Sobel à imagem ocorre através da convolução desta com duas matrizes, que podem ser observadas na figura 2.14, onde a matriz G_x é utilizada para detectar as variações horizontais na intensidade da imagem, e a matriz G_y detecta as variações verticais.

$G_x =$

-1	0	1
-2	0	2
-1	0	1

$G_y =$

-1	-2	-1
0	0	0
1	2	1

z1	z2	z3
z4	z5	z6
z7	z8	z9

Figura 2.14: Filtros usados no cálculo de gradientes no método de Sobel (adaptado de [52]).

As expressões 2.4 e 2.5 são as aproximações digitais mais simples para derivadas parciais que utilizam máscaras de tamanho 3×3.

$$G_x = (z3 + 2z6 + z9) - (z1 + 2z4 + z7) \tag{2.4}$$

$$G_y = (z7 + 2z8 + z9) - (z1 + 2z2 + z3) \tag{2.5}$$

A magnitude G pode ser calculada através da expressão 2.6, que em algumas implementações, devido a carga computacional exigida, é mais desejável aproximar as operações de quadrados e raízes quadradas em valores absolutos, ao mesmo tempo, preservando as alterações nos níveis de intensidade.

$$G = \sqrt{{G_x}^2 + {G_y}^2} \approx \left|G_x\right| + \left|G_y\right| \tag{2.6}$$

2.3.3 Binarização da imagem: Método de Otsu

Para aplicar os algoritmos de extracção de características é necessário binarizar a imagem, ou seja, separar na imagem os objectos claros do fundo escuro. Na figura 2.15 encontra-se um exemplo de uma histograma duma imagem onde os pontos claros estão bem separados dos pontos escuros. Binarizar, tal como o nome indica, é converter a imagem numa representação binária, em apenas dois níveis de intensidade, preto ou branco. Tudo que não for importante, que não faz parte do objecto, é considerado fundo, ou *background*, e lhe é atribuído o tom de preto, e as regiões de interesse são consideradas como primeiro plano, ou *foreground*, e lhes é atribuída a cor branca.

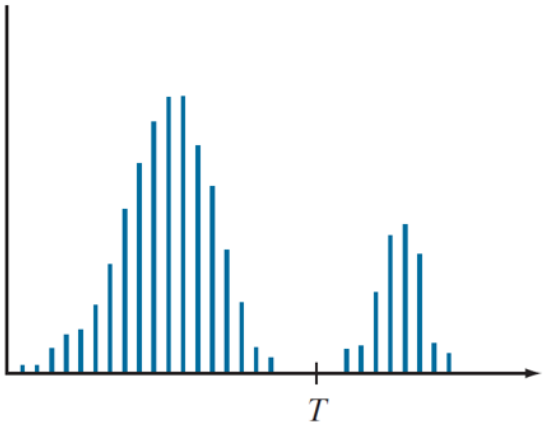


Figura 2.15: Histogramas de intensidades que podem ser divididos por um *threshold* [53].

O método mais básico e óbvio de segmentar a imagem em dois é indicar o limiar de intensidade, ou *threshold*, a partir do qual os píxeis tomam a cor branca em vez de preto. A imagem segmentada, denominada por $g(x,y)$, é dada pela expressão 2.7.

$$g(x,y) = \begin{cases} 1, & \text{if } f(x,y) > T \\ 0, & \text{if } f(x,y) \leq T \end{cases} \tag{2.7}$$

Actualmente existem inúmeros métodos que conseguem calcular automaticamente valor ideal a atribuir ao limiar. Uma das técnicas mais utilizadas para determinar o limiar duma imagem em que os pontos claros estão bem separados dos pontos escuros, ou seja, duma imagem representada por um histograma bimodal, é o método de Otsu.

O objectivo do método de Otsu é encontrar o limiar q de tal modo que as distribuições de *background* e *foreground* sejam separadas ao máximo [54], que seja minimizada a variância intra-classes. O critério de minimização da variância intra-grupos baseia-se na separação dos níveis de cinzento em dois grupos, o mais homogéneo possível.

Seja $P_0(q)$ a probabilidade do grupo de píxeis para os quais $i \leq q$ e $\sigma_0^2(q)$ representa a sua variância. Seja $P_1(q)$ a probabilidade do grupo de píxeis para os quais $i > q$ e $\sigma_1^2(q)$ é a sua variância. A variância conjunta será dada por:

$$\sigma_W^2(q) = P_0(q) \cdot \sigma_0^2(q) + P_1(q) \cdot \sigma_1^2(q),$$

onde

$$P_0(q) = \sum_{i=0}^{q} p(i), \qquad P_1(q) = \sum_{i=q+1}^{K-1} p(i),$$

$$\mu_0(q) = \frac{\sum_{i=0}^{q} i \times p(i)}{P_0(q)}, \qquad \mu_1(q) = \frac{\sum_{i=q+1}^{K-1} i \times p(i)}{P_1(q)},$$

$$\sigma_0^2(q) = \frac{\sum_{i=0}^{q} (\mu_0(q) - i)^2 \cdot p(i)}{P_0(q)}, \qquad \sigma_1^2(q) = \frac{\sum_{i=q+1}^{K-1} (\mu_1(q) - i)^2 \cdot p(i)}{P_1(q)}.$$

A variância intra-classe é simplesmente a soma das variâncias individuais ponderadas pelas probabilidades de grupo correspondentes. Analogamente, a variância inter-classe, que é dada por:

$$\sigma_b^2(q) = P_0(q) \cdot (\mu_0(q) - \mu)^2 + P_1(q) \cdot (\mu_1(q) - \mu)^2,$$

mede as distâncias entre as médias dos agrupamentos $\mu_0(q)$, $\mu_0(q)$ e média geral μ. A variância total da imagem é dada por:

$$\sigma^2 = \sigma_W^2(q) + \sigma_b^2(q).$$

O objectivo é portanto, encontrar o nível q que permite minimizar a variância conjunta (σ^2), e como essa, para uma dada imagem, é constante, minimizar a variância intra-classe (σ_W^2) é equivalente a maximizar a variância inter-classe (σ_b^2), cuja expressão é:

$$\sigma_b^2(q) = \sigma^2 - \sigma_W^2(q) = P_0(q) \cdot P_1(q) \cdot [\mu_0(q) - \mu_1(q)]^2.$$

Porque a variância inter-classe só depende de estatísticas de primeira ordem, ao invés de minimizar a variância intra-classe, maximizar a inter-classe acaba por ser escolha natural para a implementação computacional.

2.3.4 Extracção de características: Transformada de Hough

Em muitos casos, os ambientes de trabalho não são estruturados, ou seja, onde só se tem disponível um mapa de contornos e não existe informação acerca da localização dos objectos de interesse. Nestes casos, todos os píxeis são candidatos para ligação, e eles são considerados ou eliminados com base em propriedades predefinidas [53].

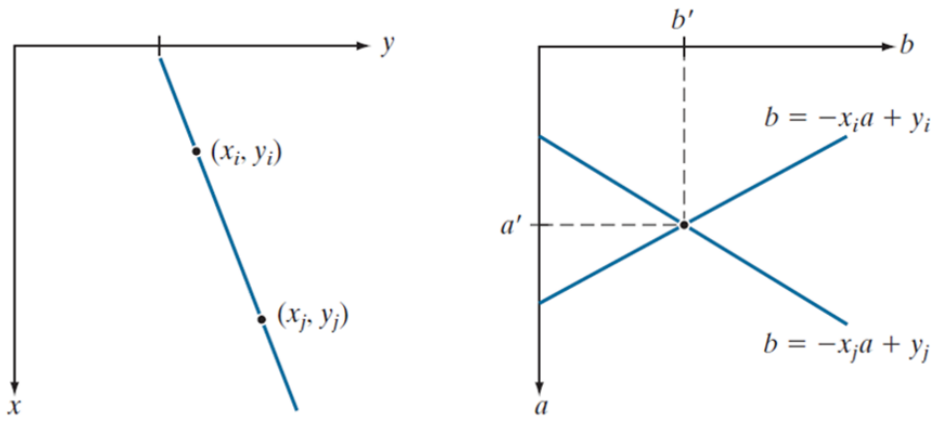
O método de Paul Hough, frequentemente referido como "transformada de Hough", ou, *Hough Transform (HT)*, é uma abordagem geral de localização de qualquer forma que pode ser definida parametricamente numa distribuição de pontos [54], como é o caso das linhas rectas, dos círculos ou elipses.

A transformada de Hough é talvez mais utilizada para detectar segmentos de linhas em mapas de contornos. Um segmento de linha no plano xy pode ser descrito com dois parâmetros de valores reais usando a equação reduzida da recta

$$y = ax + b, \tag{2.8}$$

onde a representa o declive e b a ordenada na origem (intersecção em y). Um segmento de linha que passa por dois pontos do contorno $p_i = (x_i, y_i)$ e $p_j = (x_j, y_j)$ quaisquer têm de satisfazer as condições $y_i = ax_i + b$ e $y_j = ax_j + b$ para $a, b \in \mathbb{R}$.

Sendo (x_i, y_i) um ponto no plano xy, ao escrever a equação 2.8 como $b = -x_i a + y_i$ e considerando o plano ab (também chamado espaço de parâmetros) produz a equação duma única linha para esse ponto fixo. Mais, um segundo ponto (x_j, y_j) também tem uma única linha no espaço de parâmetros associado, que intersecta a linha associada com (x_i, y_i) num ponto qualquer (a', b') no espaço de parâmetros, onde a' representa o declive e b' a intercepção da linha que contém p_i e p_j no plano xy. Na verdade, todos os pontos nessa linha de contornos têm linhas no espaço de parâmetros que intersectam em (a', b'). A figura 2.16 ilustra este conceito.



(a) Plano xy. (b) Espaço de parâmetros.

Figura 2.16: Plano xy e espaço de parâmetros [53].

Quantas mais linhas intersectarem num único ponto do espaço de parâmetros, mais pontos da imagem correspondem à linha no mapa de contornos. No geral, pode ser afirmado: Se N linhas intersectam na posição (a', b') no espaço de parâmetros, então N pontos da imagem fazem parte da linha $y = a'x + b'$ no plano xy.

A representação da linha na equação 2.8 não é usada muito na práctica porque o declive a aproxima-se do infinito quando a linha atinge a direcção vertical. Uma maneira de contornar esta dificuldade, ou limitação, é usar a chamada representação normal da linha, *Hessian Normal Form (HNF)*:

$$xcos(\theta) + ysin(\theta) = \rho \tag{2.9}$$

A figura 2.17a ilustra a interpretação geométrica dos parâmetros ρ e θ. Cada curva sinusoidal da figura 2.17b representa a família de linhas que passam através de um ponto particular (x_k, y_k) no plano xy. O ponto de intersecção (ρ', θ') na figura 2.17b corresponde à linha que passa por ambos os pontos da figura 2.17a, (x_i, y_i) e (x_j, y_j).

A atractividade computacional da transformada de Hough surge da subdivisão do espaço de parâmetros ρ em as chamadas células acumuladoras (*accumulator cells*), como ilustrado na figura 2.17e, onde (ρ_{min}, ρ_{max}) e $(\theta_{min}, \theta_{max})$ são os intervalos esperados dos valores dos parâmetros: $-90° \leq \theta \leq 90°$ e $-D \leq \rho \leq D$, onde D é a distância máxima entre os cantos opostos da imagem. O número de subdivisões no plano $\rho\theta$ determina a precisão da colinearidade dos pontos. O número de computações neste método é linear com respeito a n, o número de pontos não pertencentes ao fundo da imagem no plano xy.

2.3.5 Traçado de linhas

Uma linha simples para ser rastreada do ponto P_0 até P_1 para todas as posições x dos píxeis da linha, a posição y pode ser calculada com:

$$y = (x - x_0)(y_1 - y_0)/(x_1 - x_0) + y_0. \tag{2.10}$$

Este método necessita de uma multiplicação e uma divisão de ponto flutuante que apesar de não parecer difícil de calcular é uma desvantagem e é possível fazer isto de maneiras mais eficientes.

A expressão para calcular a posição y contém a razão $\triangle x/\triangle y$ do declive. Para evitar cálculos de ponto flutuante, em vez da fracção é possível fazer o cálculo através dos números inteiros do numerador e denominador.

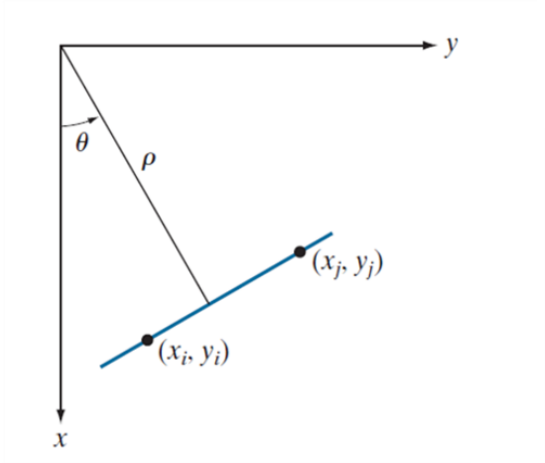
Um dos primeiros algoritmos desenvolvidos no campo da computação gráfica de desenho de linhas rectas entre dois pontos em dispositivos matriciais é o algoritmo de Bresenham [55]. Este algoritmo é um algoritmo de erro incremental, e permite determinar quais os pontos que devem ser destacados numa matriz de base quadriculada para atender ao grau de inclinação de um ângulo, usando apenas operações como adição, subtração e deslocamento de *bits* inteiros, todas as quais são muito baratas em arquitecturas de computador padrão.

A equação implícita para uma linha recta de um ponto $P_0 = (x_0, y_0)$ até $P_1 = (x_1, y_1)$ é:

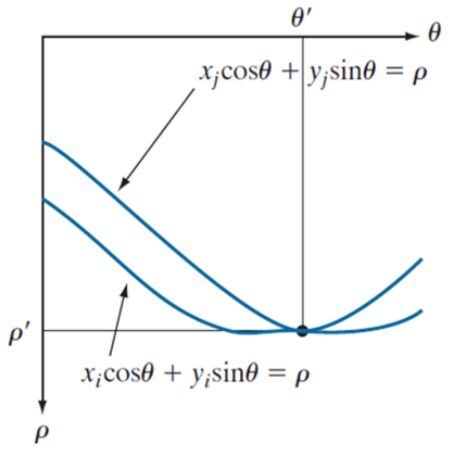
$$(x_1 - x_0)(y - y_0) - (x - x_0)(y_1 - y_0) = 0. \tag{2.11}$$

Com a definição de $dx = x_1 - x_0$ e $dy = y_1 - y_0$ o erro e, que é o desvio do píxel da linha, é então calculado pela expressão:

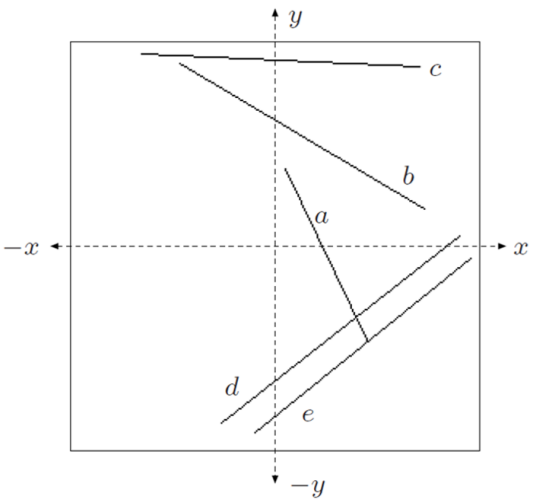
$$e = (y - y_0)dx - (x - x_0)dy. \tag{2.12}$$



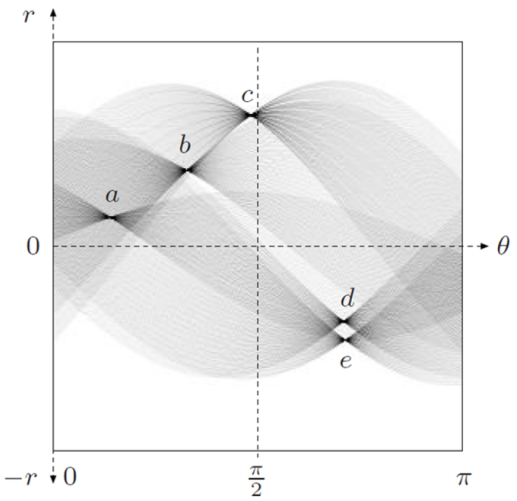
(a) (ρ, θ) parametrização de uma linha no plano xy [53].



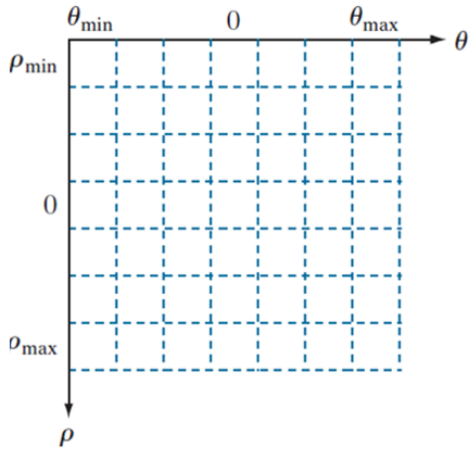
(b) Curvas sinusoidais no plano $\rho\theta$; o ponto de intersecção (ρ', θ') corresponde à linha que passa pelos pontos (x_i, y_i) e (x_j, y_j) no plano xy [53].



(c) Espaço da imagem, plano xy [54].



(d) Espaço de parâmetros usando a representação HNF [54].



(e) Divisão do plano $\rho\theta$ em células acumuladoras.

Figura 2.17

O valor do erro é zero para todos os píxeis que se encontram exactamente sobre a linha, positivo quando se encontram num lado da linha e negativo quando se encontram noutro lado. Este erro é usado para decidir qual será o próximo píxel da linha.

O erro do passo diagonal é dado por:

$$e_{xy} = (y + 1 - y_0)dx - (x + 1 - x_0)dy = e + dx - dy. \tag{2.13}$$

Os erros de cálculos das direcções x e y são obtidos através das expressões:

$$e_x = (y + 1 - y_0)dx - (x - x_0)dy = e_{xy} + dy \tag{2.14}$$

$$e_y = (y - y_0)dx - (x + 1 - x_0)dy = e_{xy} + dx \tag{2.15}$$

e erro do primeiro passo é igual a:

$$e_1 = (y_0 + 1 - y_0)dx - (x_0 + 1 - x_0)dy = dx - dy. \tag{2.16}$$

A figura 2.18 mostra uma linha com $dx = 5$ e $dy = 4$. O valor do erro do píxel ciano é 1. Os três píxeis cinzentos são as próximas escolhas possíveis. O aumento na direcção x subtrai 4 (dy) ao valor do erro e o aumento na direcção y adiciona 5 (dx) ao valor do erro. O píxel cinzento escuro tem o valor absoluto mais pequeno. Este é calculado com antecedência, já que o valor do erro é um píxel diagonal à frente.

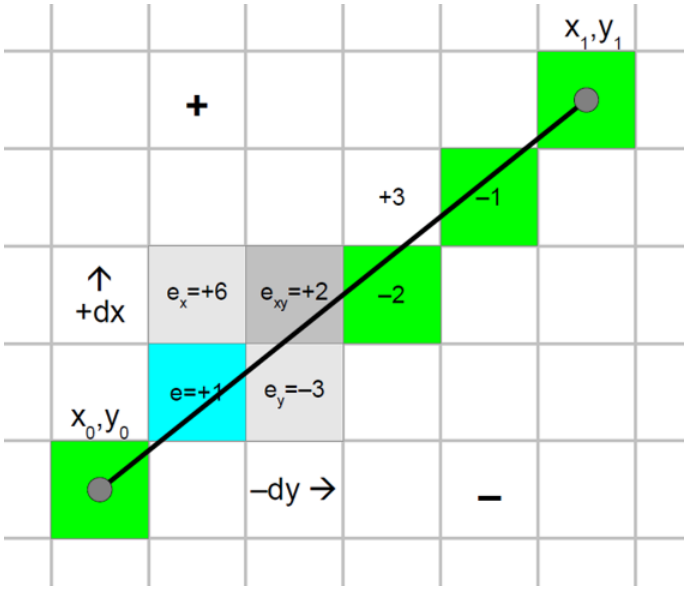


Figura 2.18: Linha com valores de erro ($dx = 5, dy = 4$) [56].

Como $e_{xy} + e_y = +2 - 3 = -1$ é inferior a zero a direcção y é incrementada e dx é adicionado ao valor do erro. O mesmo é feito para a outra direcção: $e_{xy} + e_x = +2 + 6 = +8$ é maior que zero, logo a direcção x é incrementada e dy é subtraído do valor do erro.

Para lidar com os gradientes negativos ou linhas invertidas os autores de [56] negam a direcção do passo, cujo algoritmo pode ser observado na listagem 2.1.

Não há aproximação pelo algoritmo. Portanto, o valor de erro do último píxel é sempre exatamente zero. Esta versão é optimizada para verificar o fim do ciclo apenas se a direção correspondente for incrementada.

```
1   void plot_line (int x0, int y0, int x1, int y1)
2   {
3     int dx =  abs (x1 - x0), sx = x0 < x1 ? 1 : -1;
4     int dy = -abs (y1 - y0), sy = y0 < y1 ? 1 : -1;
5     int err = dx + dy, e2; /* error value e_xy */
6
7     for (;;){  /* loop */
8       setPixel (x0,y0);
9       if (x0 == x1 && y0 == y1) break;
10      e2 = 2 * err;
11      if (e2 >= dy) { err += dy; x0 += sx; } /* e_xy+e_x > 0 */
12      if (e2 <= dx) { err += dx; y0 += sy; } /* e_xy+e_y < 0 */
13    }
14  }
```

Listagem 2.1: Programa para traçar uma linha recta [56].

Capítulo 3

Plataformas e Ambientes de Desenvolvimento

Neste capítulo são descritas todas as ferramentas utilizadas no projecto, tanto as de *hardware* como as de *software*.

Primeiro serão caracterizados os dispositivos do *Embedded Vision Bundle* da Digilent, isto é, a placa Zybo Z7-20 de Xilinx e o aparelho de aquisição de imagem, sensor Pcam 5C. Seguidamente serão especificadas as ferramentas que Xilinx oferece aos utilizadores para a criação de projectos baseados em FPGA.

3.1 DIGILENT Zybo Z7

A Zybo Z7 é uma placa de prototipagem de *software* integrado e circuitos digitais construída em volta da família Xilinx Zynq-7000 [57]. A família Zynq é baseada na arquitectura *All Programmable System-on-Chip (APSoC)* de Xilinx, que integra o processador de dois núcleos ARM Cortex-A9 com lógica *Field-Programmable Gate Array (FPGA)* da série 7 de Xilinx. A Zybo Z7 rodeia o Zynq com um conjunto rico de periféricos multimédia e de conectividade. As características com habilidades vídeo da Zybo Z7, incluindo, um conector MIPI CSI-2 compatível com Pcam, entrada HDMI, saída HDMI, e elevada largura de banda DDR3L, permitem criar soluções acessíveis para aplicações de *embedded vision* de alto nível, pelas quais são conhecidos os FPGAs de Xilinx. Os conectores Pmod da placa facilitam a anexação de *hardware* adicional. Xilinx tem um catálogo com mais de 70 periféricos disponíveis, incluindo controladores de motores, sensores, ecrãs, entre outros.

3.1.1 Características

- Processador ZYNQ
  - Processador *dual-core* Cortex-A9 667 MHz.
  - Controlador de memória DDR3L com 8 canais DMA e 4 portos escravos AXI3 de alto desempenho.
  - Controladores de periféricos de elevada largura de banda: Ethernet 1G, USB 2.0, SDIO.
  - Controladores de periféricos de baixa largura de banda: SPI, UART, CAN, I2C.
  - Programável a partir de JTAG, *Quad-SPI Flash*, e cartão microSD.
  - Lógica programável equivalente a FPGA Artix-7.
- Memória
  - 1GB de memória DDR3L com barramento 32-*bit* @ 1066 MHz.
  - 16MB *Quad-SPI Flash* com número aleatório de 128 *bits* programado de fábrica e identificador compatível globalmente único EUI-48/64™ de 48 *bits*.
  - *slot* microSD.
- Alimentação
  - Alimentado a partir de USB ou qualquer fonte de alimentação externa de 5V.
- USB e Ethernet
  - *Gigabit Ethernet PHY*.
  - Circuitos de programação USB-JTAG.
  - Ponte USB-UART.
  - USB 2.0 OTG PHY com suporte de *host* e dispositivo.
- Áudio e Vídeo
  - Conector de câmara Pcam com suporte MIPI CSI-2.
  - Porto *sink* HDMI (entrada) com CEC.
  - Porto de origem HDMI (saída) com CEC.
  - *Codec* de áudio com auscultador estéreo, *line-in* estéreo, e tomada de microfone.
- Interruptores, Botões de pressão, e LEDs
  - 6 botões de pressão (2 conectados ao processador).
  - 4 interruptores de corrediça.
  - 5 LEDs (1 conectado ao processador).
  - 2 LEDs RGB.
- Conectores de Expansão
  - 6 portos Pmod.
  - 8 I/O de Processador.
  - 40 I/O de FPGA.
  - 4 pares diferenciais de 0-1.0V com capacidade analógica para XADC.

3.1.2 Opções Disponíveis

A placa Zybo Z7 pode ser comprada ou com Zynq-7010 ou com Zynq-7020. Estas duas variações de produto são referidas como Zybo Z7-10 e Zybo Z7-20 (ver figura 3.1) respectivamente, e a diferença principal entre ambas é o tamanho do FPGA dentro do APSoC do Zynq. Os processadores Zynq têm as mesmas capacidades, mas o FPGA interno da variante -20 é cerca de 3 vezes maior que o da -10. As diferenças entre as duas variantes estão resumidas na tabela 3.1. Também é de notar que devido ao tamanho inferior do FPGA de Zynq-7010, este acaba por não ser adequado para aplicações de *embedded vision*.

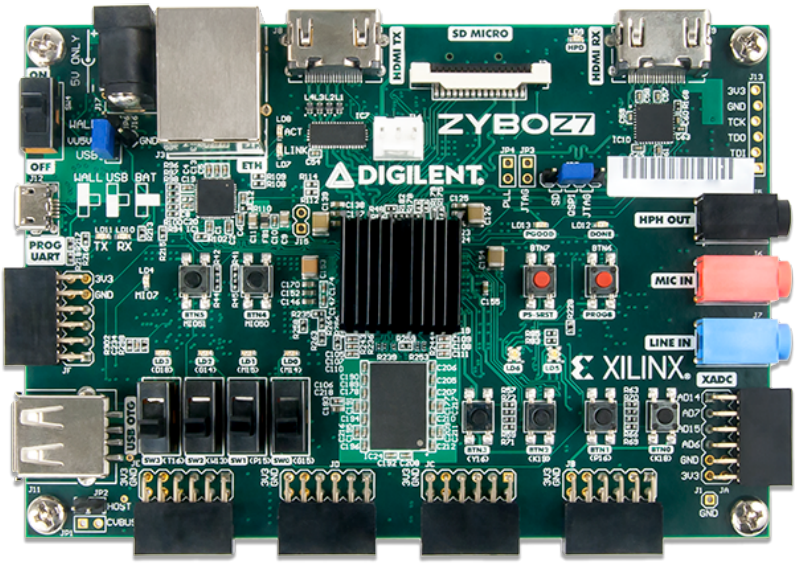


Figura 3.1: Placa Zybo Z7-20 ([57]).

3.1.3 Porto Pcam

O porto Pcam incluído na placa Zybo Z7 tem 15 pinos (passo de 1mm), conector *Zero Insertion Force (ZIF)* especificamente criado para ligar os módulos de sensores de câmaras aos sistemas anfitrião (*host systems*). Os pinos do conector Pcam têm definição rígida e incluem um barramento MIPI CSI-2 de duas pistas para dados da câmara, um barramento I2C para configuração da câmara, dois sinais adicionais de uso geral e um pino de 3,3V para alimentar o módulo da câmara, conforme ilustrado na figura 3.2. A Digilent desenvolveu um catálogo de módulos de câmara para periféricos Pcam com tipos diferentes de sensores, todos conforme este sistema de pinos.

Produto	**Zybo Z7-10**	**Zybo Z7-20**
Elemento Zynq	XC7Z010-1CLG400C	XC7Z020-1CLG400C
1 MSPS *On-chip* ADC	Sim	Sim
Programmable Logic Cells	28K	85K
Lookup-Table (LUT)	17.600	53.200
Flip-Flops	35.200	106.400
Blocos de RAM (Blocos de 36Kb)	2,1Mb (60)	4,9Mb (140)
Parcelas DSP (18×25 MACCs)	80	220
MMCM	2	4
Portos Pmod	5	6
Conector de ventoinha	Não	Sim
Dissipador de calor de Zynq	Não	Sim

Tabela 3.1: Comparação entre duas variantes de Zybo Z7 ([57], [58]).

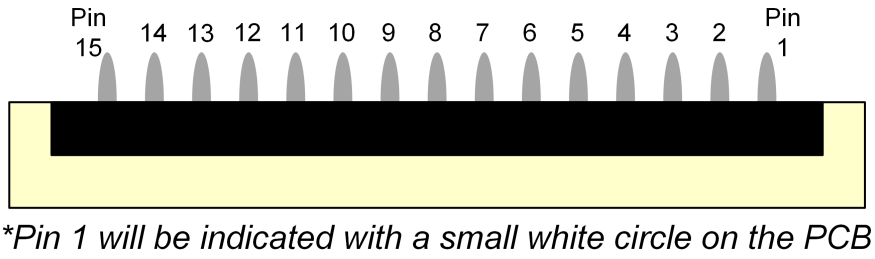


Figura 3.2: Porto Pcam ([57]).

3.1.4 Porto HDMI

O Zybo Z7 contém um porto de origem sem *buffer* (saída, rotulada como HDMI TX) e um porto *sink* com *buffer* (entrada, rotulado como HDMI RX). Ambos usam recipientes HDMI de tipo A e sinais de relógio terminados e conectados ao Zynq PL. O *buffer* usado no porto HDMI RX encerra, equaliza, condiciona, e encaminha o fluxo (*stream*) HDMI entre o conector e os pinos do FPGA.

O porto HDMI TX não inclui um *buffer* para melhorar a integridade do sinal, mas inclui um multiplexador HDMI configurado como um simples interruptor. O único objectivo deste dispositivo é evitar que os monitores forneçam energia ao Zybo Z7, que possibilita ligar e desligar o Zybo Z7 enquanto um monitor, ou *display*, é conectado ao HDMI TX.

Os conectores HDMI de 19 pinos incluem três canais de dados diferenciais, um canal de relógio diferencial, cinco conexões terra (GND), um barramento *Consumer Electronics Control (CEC)* de um fio, um canal *Display Data Channel (DDC)* de dois fios que é essencialmente um barramento I2C, um sinal *Hot Plug Detect (HPD)*, um sinal de 5V capaz de fornecer até 50mA e um pino reservado (RES). Todos os sinais *non-power* são ligados ao

Zynq PL com excepção de RES.

3.1.5 Suporte de *Software*

A Zybo Z7 é totalmente compatível com Vivado Design Suite de Xilinx. Este conjunto de ferramentas combina o *design* de lógica FPGA e desenvolvimento incorporado de *software* ARM num *design flow* intuitivo e fácil de usar. Pode ser usado para projectar sistemas de vários graus de complexidade, desde um simples programa que controla uns LEDs, até um sistema operativo funcional capaz de correr várias aplicações de servidor. Também é possível tratar o Zynq APSoC somente como um FPGA para quem não está interessado no uso do processador no seu projecto. Zybo Z7 é suportado pela licença gratuita WebPACK de Vivado, o que significa que o *software* é completamente grátis, incluindo as ferramentas *Logic Analyzer* e *High-Level Synthesis (HLS)*. O *Logic Analyzer* permite fazer *debugg* da lógica que corre no *hardware* e a ferramenta HLS permite compilar código C directamente em HDL.

Zybo Z7 também pode ser usado no ambiente SDSoC de Xilinx, o que permite a concepção fácil de programas FPGA acelerados e *pipelines* vídeo, inteiramente no ambiente C/C++. A placa Zybo Z7-20 é a recomendada para os que estão interessados em processamento vídeo, devido ao tamanho 3 vezes superior do seu FPGA interno quando comparado com Zybo Z7-10.

3.1.6 Arquitectura Zynq APSoC

Zynq APSoC, cuja visão geral da sua arquitectura está representada na figura 3.3, é dividido em dois subsistemas distintos: *Processing System (PS)* e *Programmable Logic (PL)*, onde PS tem a cor verde e PL cor amarela. Também, é de se notar que o controlador PCIe Gen2 e os transceptores Multi-gigabit não estão disponíveis nos dispositivos Zynq-7020.

PL é quase idêntica ao FPGA Artix da série 7 de Xilinx, com a diferença de que contém vários portos e barramentos dedicados que a ligam ao PS. A PL também não contém a mesma configuração *hardware* típico dum FPGA da série 7, e tem de ser configurada directamente pelo processador ou porto JTAG.

O PS é constituído por muitos componentes, incluindo *Application Processing Unit (APU)* (que inclui 2 processadores Cortex-A9), *Advanced Microcontroller Bus Architecture (AMBA) Interconnect*, controlador de memória DDR3, e vários controladores de periféricos com as suas entradas e saídas multiplexadas aos 54 pinos dedicados (chamados *Multiplexed I/O*, ou pinos MIO). Os controladores de periféricos são conectados aos processadores como escravos através do *AMBA Interconnect*, e contêm registos de controlo que podem ser endereçados no espaço da memória dos processadores. A lógica programável também está conectada ao *Interconnect* como escravo, e os projectos podem implementar vários núcleos no tecido do FPGA, que cada um também possui registos de controlo endereçáveis. Mais, os núcleos implementados em PL podem provocar interrupções nos processadores e efectuar acessos DMA à memória DDR3.

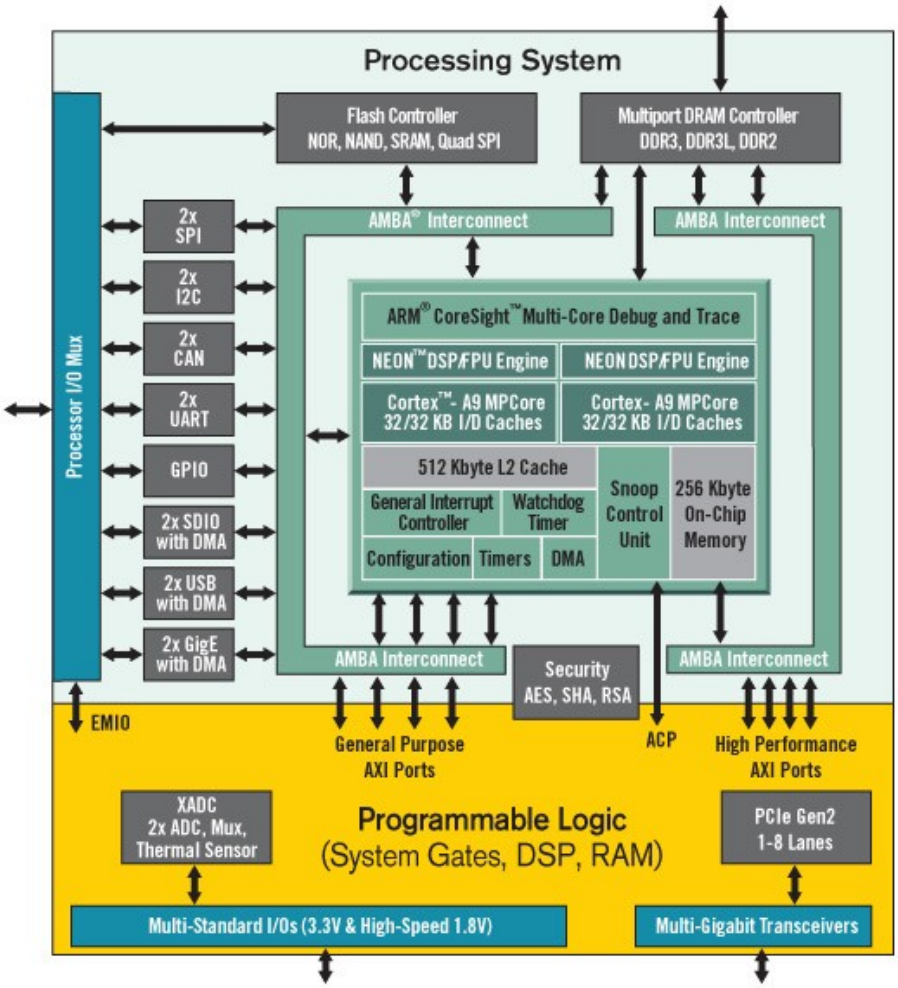
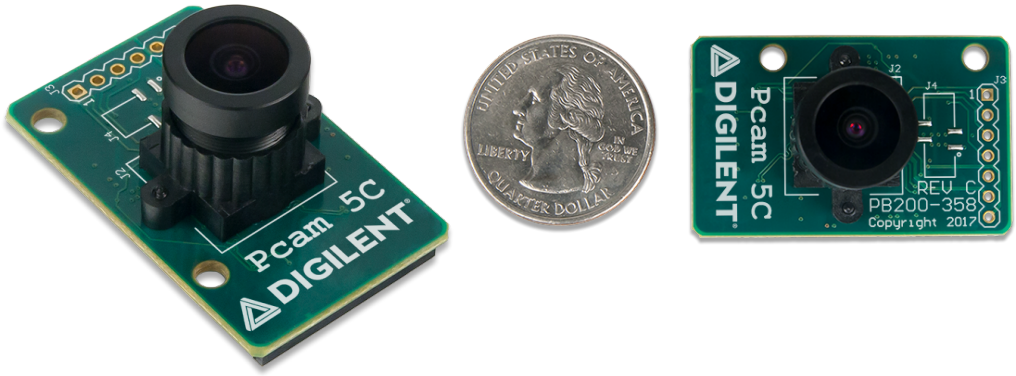


Figura 3.3: Arquitectura APSoC de Zynq ([57]).

3.2 DIGILENT Pcam 5C

Pcam 5C é uma câmara, ou melhor, um pequeno módulo de imagem (o seu PCB tem as dimensões de 40mm de comprimento e 25mm de largura) especificamente destinado para o uso em conjunto com as placas de desenvolvimento FPGA. O seu sensor de imagem é o *Omnivision OV5640 5MP*. Este sensor inclui várias funções de processamento internas que podem melhorar a qualidade de imagem, incluindo balanço automático de branco, calibração automática do nível de preto, e controlos para o ajuste de saturação, matiz, *gamma* e nitidez. Os dados são transferidos pela interface *dual-lane* MIPI CSI-2, que oferece largura de banda suficiente para suportar formatos típicos de transmissão de vídeo, tais como, 720p (a 60 *Frames Per Second*) e 1080p (a 30 *Frames Per Second*), entre outros (ver tabela 3.2). O módulo tem um chamado *Flat-Flexible Cable (FFC)* de 15 pinos e comprimento de 10cm com que é conectado ao FPGA.



(a) (b)

Figura 3.4: Sensor de imagem Pcam 5C ([59]).

Formato	**Resolução**	***Frames* por segundo**	**Método de dimensionamento**	**Relógio de píxel**
5 Mpixel	2592 × 1944	15 FPS	*full resolution (dummy 16 pixel horizontal, 8 lines)* 2608 × 1952 *with dummy*	96/192 MHz
1280 × 960	1280 × 960	45 FPS	*subsampling in vertical and horizontal* 1296×968 *supports* 2 × 2 *binning*	96/192 MHz
1080p	1920 × 1080	30 FPS	*cropping from full resolution* 1936 × 1088 *with dummy pixels*	96/192 MHz
720p	1280 × 720	60 FPS	*cropping* 2592 × 1944 *to* 2560 × 1440 *subsampling in vertical and horizontal* 1296 × 728 *with dummy supports 2x2 binning*	96/192 MHz
VGA	640 × 480	90 FPS	*subsampling from* 1280 × 960 648*x* × 84 *with dummy supports* 2 × 2 *binning*	48/96 MHz
QVGA	320 × 240	120 FPS	*subsampling from* 1280 × 960 324 × 242 *with dummy supports* 2 × 2 *binning*	24/48 MHz

Tabela 3.2: Formatos e FPS disponíveis ([60]).

3.2.1 Suporte de *Software*

Pcam 5C só pode ser controlado por um SoC com um controlador MIPI CSI-2 no *hardware*, e isso requer *software* complicado para configurar devidamente o controlador e o sensor. Existem duas opções quando se quer implementar um método ao usar este tipo de módulo com um FPGA:

- Pagar uma licença para adquirir um IP MIPI CSI-2 projectado especificamente para trabalhar com o FPGA, o que, em caso geral, acaba por fornece uma solução robusta com bom suporte de *software*. Problemas podem surgir, para além do custo em si, quando o código muitas vezes é encriptado e não pode ser estudado para finalidades educacionais.

- Desenvolver o *hardware* e *software* de raiz, o que requer um grande investimento temporal de alguém com habilidades avançadas. Também é necessário ter acesso à informação não acessível para maioria, como especificações fechadas e *datasheets*.

Para resolver este problema, a Digilent criou um conjunto de núcleos Vivado IP *open source* que funcionam com Pcam 5C em placas FPGA e Zynq. Para realizar isso na práctica, foram tomados alguns atalhos que impedem este IP de funcionar de forma tão robusta como algumas das soluções licenciadas disponíveis. Além disso, actualmente, o *software* emparelhado com o IP não aproveita todas as vantagens e todos os recursos do sensor OV5640, o que poderá afectar a qualidade da imagem produzida.

3.3 Vivado HLS (versão 2017.4)

Xilinx oferece uma vasta gama de ferramentas para o utilizador criar os seus projectos. A ferramenta Vivado HLS transforma uma especificação C para uma implementação *Register Transfer Level (RTL)* que, por sua vez, pode ser sintetizada num *Field-Programmable Gate Array*. As especificações podem ser escritas em C, C++, ou *SystemC* e o FPGA oferece uma arquitectura intensamente paralela com benefícios no desempenho, custo e energia sobre os processadores tradicionais (como já foi abordado na secção 2.1.4). *High-Level Synthesis* conecta os domínios de *hardware* e *software*, fornecendo benefícios como, melhor productividade para os projectistas de *hardware*, uma vez que, esses conseguem trabalhar num nível de abstracção mais alto enquanto criam *hardware* de alto desempenho. Também melhoram o desempenho de sistemas para os projectistas de *software*, pois, esses conseguem acelerar as partes computacionalmente intensivas do seu algoritmo no novo alvo de compilação, o FPGA. A metodologia de *design* de *High-Level Synthesis* permite [61]:

- Desenvolver algoritmos ao nível C, abstraindo os detalhes de implementação, o que permite poupar tempo de desenvolvimento.

- Efectuar a verificação ao nível C, ou seja, permite validar o projecto mais rapidamente do que as tradicionais *Hardware Description Languages*.
- Controlar o processo da síntese C através das directivas de optimização ([62]), ou seja, criar implementações *hardware* específicas de alto desempenho.
- Criar múltiplas implementações do código fonte C usando as directivas de optimização, e explorar o espaço de desenvolvimento, o que aumenta a possibilidade e encontrar a implementação ideal.
- Criar fontes de código C portáteis e legíveis que poderão ser incorporados em novos projectos.

High-Level Synthesis inclui as fases de **Agendamento** (*Scheduling*), **Ligação** (*Binding*), **Extracção da Lógica de Controlo** (*Control logic extraction*). O *Scheduling* determina que operações ocorrem em cada ciclo de relógio baseando-se na duração do ciclo de relógio, ou a sua frequência, tempo que demora para concluir a operação, conforme definido pelo dispositivo de destino, e directivas de optimização definidas pelo utilizador.

Dependendo se o período de relógio for mais longo ou se o dispositivo alvo for especificado um FPGA mais rápido, mais operações serão concluídas dentro de um único ciclo de relógio, e todas as operações podem ser concluídas num único ciclo. Por outro lado, se o período de relógio for mais curto ou se for especificado um FPGA mais lento, HLS agenda automaticamente as operações para mais ciclos de relógio, e algumas operações podem precisar ser implementadas como recursos multi-ciclo.

O *Binding* determina quais são os recursos *hardware* que implementam cada operação agendada. Para se implementar a solução ideal HLS usa informações do dispositivo alvo.

O *Control logic extraction* extrai a lógica de controlo para criar uma máquina de estados finita, ou *Finite State Machine (FSM)*, que sequencia as operações no *design* RTL.

A ferramenta Vivado HLS de Xilinx sintetiza uma função C num bloco IP que pode ser integrado num sistema de *hardware*. É fortemente integrado com o resto das ferramentas do ecossistema Xilinx e proporciona suporte compreensivo de línguas e recursos para criar a implementação ideal para o algoritmo C.

O *design flow* de Vivado HLS pode ser observado na figura 3.5, e os seus passos são:

1. Compilar, executar (simular), e fazer *debug* do algoritmo C[1].
2. Sintetizar o algoritmo C numa implementação RTL, opcionalmente usando as directivas de optimização do utilizador.
3. Gerar relatórios compreensivos e analisar o *design*.
4. Verificar a implementação RTL.

[1] Nota: Em HLS, a execução do programa C compilado é referido como simulação C. A execução do algoritmo C simula a função para validar se a funcionalidade do algoritmo é correcta.

5. Empacotar a implementação RTL numa selecção de formatos IP.

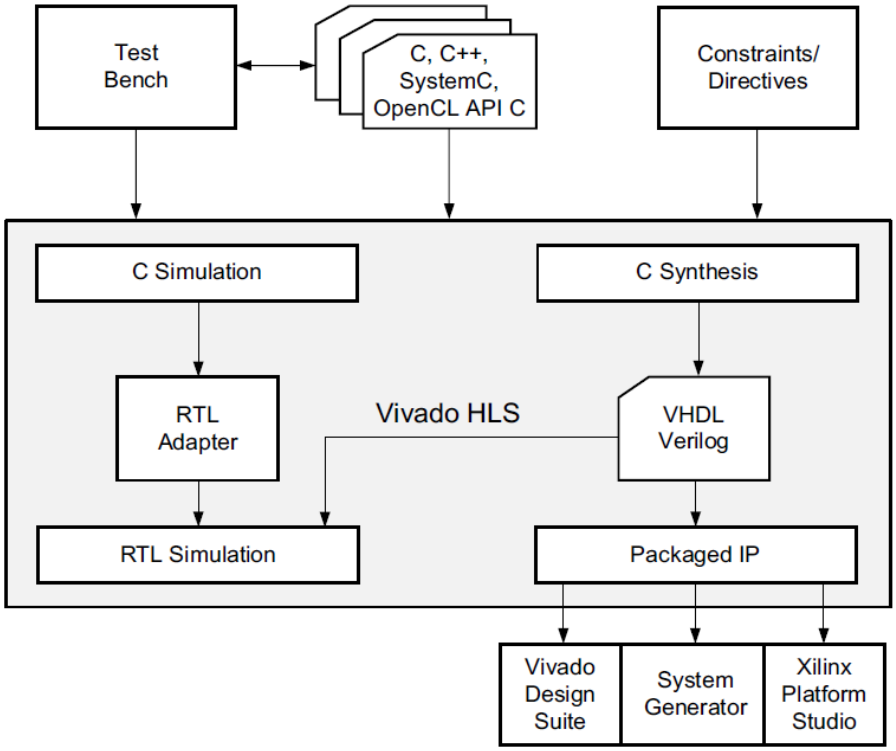


Figura 3.5: *Design Flow* de Vivado HLS ([61]).

O código C é sintetizado em *High-Level Synthesis* da seguinte maneira [61]:

- Os argumentos da função *top-level* são sintetizados em portos I/O de RTL.

- As funções C sintetizam em blocos na hierarquia RTL.
  Se o código C incluir uma hierarquia de sub-funções, o *design* de RTL final incluirá uma hierarquia de módulos ou entidades que têm uma correspondência um a um com a hierarquia original da função C. Todas as instâncias duma função usam a mesma implementação ou bloco RTL.

- Os ciclos das funções C são mantidos enrolados (*rolled*) por definição.
  Quando os ciclos são enrolados, a síntese cria a lógica para uma iteração do ciclo, e o *design* RTL executa sequencialmente essa lógica para cada iteração do ciclo. Com o uso de directivas de optimização, os ciclos podem ser desenrolados (*unrolled*), o que permite que todas as iterações ocorram em paralelo.

- Os arranjos (*arrays*) no código C são sintetizados em blocos RAM ou UltraRAM no *design* final de FPGA. Se o arranjo for na interface da função *top-level*, HLS implementa os arranjos como portos para ter acesso aos blocos RAM fora do *design*.

O *High-Level Synthesis* para criar a implementação ideal baseia-se no comportamento predefinido, nas restrições, e outras directivas de optimização quaisquer que o utilizador especificar. As directivas de optimização podem ser usadas para modificar e controlar o comportamento predefinido da lógica interna e dos portos I/O. Isso possibilita, a partir do mesmo código C, gerar variações da implementação de *hardware*.

Para determinar se o *design* cumpre os requisitos pretendidos, HLS gera um relatório da síntese onde podem ser revistas e analisadas as métricas de desempenho, para que de seguida, com a ajuda de directivas de optimização, a implementação possa ser refinada. O relatório da síntese contém informações sobre as seguintes métricas de desempenho:

- Área: Quantidade de recursos *hardware* necessários para implementar o *design* com base nos recursos disponíveis no FPGA, incluindo os LUTs, registos, blocos de RAM, e os DSP.
- Latência: Número de ciclos de relógio necessários para a função computar todos os valores de saída.
- Intervalo de iniciação: Número de ciclos de relógio antes que a função possa aceitar novos dados de entrada.
- Latência de iteração de ciclo: Número de ciclos de relógio necessários para completar uma iteração do ciclo.
- Intervalo de iniciação de ciclo: Número de ciclos de relógio antes de a próxima iteração do ciclo começar a processar dados.
- Latência do ciclo: Número de ciclos para executar todas as iterações do ciclo.

As ferramentas *High-Level Synthesis* podem proporcionar imensos benefícios para implementação de algoritmos de processamento de imagem em FPGAs. As representações de alto nível, tipicamente em C, permitem expressar os algoritmos de uma forma mais simplificada, reduzindo, desse modo, os tempos de desenvolvimento das aplicações. No entanto, elas também apresentam ainda algumas limitações [63].

Apenas traduzir e recompilar o código C em HLS produzirá desempenhos relativamente fracos. Na maior parte das ferramentas, os algoritmos têm que ser escritos seguindo um estilo particular para que possa ser identificado e explorado o paralelismo dos FPGAs, o que requer a reestruturação do código. Os algoritmos muito baseados em apontadores e aritmética com apontadores não sintetizam bem em *hardware*. A recursão é outra técnica de *software* que acaba por não traduzir adequadamente para *hardware*. Por fim, o RTL produzido pelas ferramentas HLS não é muito legível, o que torna mais difícil efectuar alterações nesse nível, caso seja necessário determinar saídas com falhas.

É preciso sempre recordar que o factor chave do *design* baseado em FPGA é o *hardware* e não o *software*, e simplesmente usar linguagens de alto nível não alivia a necessidade de conhecer e criar o *design* apropriado de *hardware*.

3.3.1 *Test Bench*, Suporte de Linguagens, e Bibliotecas C

Em qualquer programa C, a função *top-level* é chamada de `main()`. No *design flow* de Vivado HLS qualquer sub-função abaixo de `main()` pode ser especificada como a função *top-level* para a síntese. No entanto, a função `main()` em si não pode ser sintetizada como *top-level*.

Existem ainda umas regras adicionais, tais como, apenas é permitido que uma só função do programa seja como *top-level* para a síntese. Além disso, qualquer sub-função na hierarquia abaixo da função *top-level* também é sintetizada. Adicionalmente, se se pretender sintetizar funções que não estão na hierarquia abaixo da função *top-level*, elas têm de ser combinadas numa única função *top-level* para a síntese [61].

Quando é usado o *design flow* de Vivado HLS, sintetizar uma função C incorrecta e, em seguida, analisar os detalhes da implementação para determinar porquê é que a função não funciona conforme o esperado consome muito tempo. Para melhorar a produtividade, é recomendado o uso de uma bancada de teste antes da síntese final para validar se a função C tem funcionamento correcto.

A bancada de teste C inclui a função `main()` e outras sub-funções quaisquer que não estão na hierarquia abaixo da função *top-level* para síntese. Essas funções fornecem estímulos e consomem as saídas da função para verificar se o funcionamento da função *top-level* corresponde ao esperado.

Vivado HLS usa a bancada de teste para compilar e executar a simulação C. Durante o processo de compilação pode ser seleccionada a opção *Launch Debugger* para abrir um ambiente de *debug* completo, que permite analisar a simulação C.

Para a compilação/simulação em C, Vivado HLS suporta as normas *ANSI-C (GCC 4.6)*, *C++ (G++ 4.6)*, *OpenCL API (1.0 embedded profile)* e *SystemC (IEEE 1666-2006, version 2.2)*. Vivado HLS suporta muitas construções das linguagens C, C ++ e SystemC e todos os tipos de dados nativos para cada idioma, incluindo tipos como *float* e *double*. No entanto, a síntese não suporta algumas construções, como a alocação dinâmica de memória, pois o FPGA tem um número fixo de recursos, e a criação e libertação dinâmica desses recursos não é aceite.

Outras construções não suportadas são as operações do sistema operativo. Todos os dados do FPGA têm de ser lidos dos portos de entrada ou escritos para os portos de saída. As operações do sistema operativo, tais como, a leitura e escrita de ficheiros ou *queries* como tempo e data, não são suportadas. Em vez disso, a bancada de teste C pode realizar essas operações e passar os dados para a função de síntese como argumentos de função.

Outra característica importante da ferramenta Vivado HLS são as suas bibliotecas de funções que visam facilitar a modelação de programas em C e a sua síntese para RTL [61]. O uso dessas bibliotecas ajuda a garantir a alta qualidade dos resultados, ou seja, o saída final é um *design* de alto desempenho que faz o uso ideal dos recursos. Algumas dessas bibliotecas são, *HLS Stream Library*, *HLS Math Library*, *HLS IP Library*, *HLS Linear Algebra Library*, *HLS DSP Library*, e talvez a mais importante para este trabalho, *HLS Video*

Library. Cada uma pode ser incluída num projecto ao introduzir o seu cabeçalho (exemplo: `#include <hls_video.h>`).

3.3.2 Biblioteca *Video* de HLS

Para esta biblioteca poder ser usada num projecto, primeiro é necessário incluir o seu ficheiro de cabeçalho `hls_video.h`. Este ficheiro inclui todos os tipos e funções específicas de processamento de imagem e vídeo disponíveis em Vivado HLS.

O projecto é escrito em C++, e ao usar a biblioteca de vídeo Vivado HLS, o único requisito de uso adicional é usar o *namespace* `hls`:

```
#include <hls_video.h>

hls::rgb_8 video_data[1920][1080]
```

ou, alternativamente:

```
#include <hls_video.h>
using namespace hls;

rgb_8 video_data[1920][1080]
```

Os tipos de dados disponíveis na biblioteca *Video* são usados para garantir que o RTL criado pela síntese possa ser integrado sem problemas com qualquer bloco *Xilinx® Video IP* usado no sistema.

Existe também uma classe C++, o *LineBuffer*, que permite declarar e gerir com facilidade *buffers* de linha dentro do código algorítmico. Esta classe fornece todos os métodos necessários para instanciar e trabalhar com os *buffers* de linha. As suas características principais são: o suporte para todos os tipos de dados através da parametrização, número de linhas e colunas definidos pelo utilizador, empilhamento automático das linhas para separar bancos de memória com o fim de obter uma maior largura de banda.

A classe *memory window* é parecida com a classe *line buffer* e tem características semelhantes, sendo a principal diferença a dimensão da memória utilizada. *Line buffer* funciona numa dimensão e *memory window* a duas.

Por fim, a biblioteca *Video* inclui imensas funções de processamento vídeo que são compatíveis com as funções de OpenCV, e similarmente denominadas (ver figura 3.6), mas elas não as substituem directamente. O tipo de dados usados nas funções de processamento vídeo, `hls::Mat`, permite que as funções sejam sintetizadas e implementadas como *hardware* de alto desempenho.

Existem três tipos de funções disponibilizadas pela biblioteca *Video* de Vivado HLS [61]:

Video Data Modeling		AXI4-Stream IO Functions	
Linebuffer class	Window class	AXIvideo2Mat	Mat2AXIvideo
OpenCV Interface Functions			
cvMat2AXIvideo	AXIvideo2cvMat	cvMat2hlsMat	hlsMat2cvMat
IplImage2AXIvideo	AXIvideo2IplImage	IplImage2hlsMat	hlsMat2IplImage
CvMat2AXIvideo	AXIvideo2CvMat	CvMat2hlsMat	hlsMat2CvMat
Video Functions			
AbsDiff	Duplicate	MaxS	Remap
AddS	EqualizeHist	Mean	Resize
AddWeighted	Erode	Merge	Scale
And	FASTX	Min	Set
Avg	Filter2D	MinMaxLoc	Sobel
AvgSdv	GaussianBlur	MinS	Split
Cmp	Harris	Mul	SubRS
CmpS	HoughLines2	Not	SubS
CornerHarris	Integral	PaintMask	Sum
CvtColor	InitUndistortRectifyMap	Range	Threshold
Dilate	Max	Reduce	Zero

Figura 3.6: Biblioteca HLS Video [61].

- Funções de interface OpenCV: Estas funções convertem os dados do tipo *streaming* AXI4 para os tipos padrão OpenCV. Elas permitem transferir dados a qualquer função OpenCV executada em *software*, através das funções de *streaming* AXI4, de e para o bloco de *hardware* criado pelo HLS.
- Funções de interface AXI: As funções da interface AXI4 são usadas para transmitir dados para dentro e para fora da função a ser sintetizada. As funções de vídeo a serem sintetizadas usam o tipo de dados `hls::Mat` para uma imagem.
- Funções de processamento vídeo: Estas funções de manipulação e processamento de imagens vídeo são compatíveis com as funções padrão OpenCV. Elas usam o tipo de dados `hls::Mat` e são sintetizadas pelo Vivado HLS.

Para garantir uma implementação de alta qualidade e desempenho, as funções de vídeo HLS são previamente optimizadas . As funções já incluem as directivas de optimização necessárias para processar os dados a uma taxa de uma amostra por relógio. As métricas exactas de desempenho das funções de vídeo dependem da taxa de relógio e das especificações do dispositivo de destino.

Capítulo 4

Algoritmo de Detecção de Faixa de Rodagem Automóvel

Neste capítulo é descrita a arquitectura proposta para a aplicação de detecção de faixa de rodagem. Seguidamente são apresentados alguns resultados obtidos em simulação e implementação em *hardware*.

4.1 Arquitectura do Sistema

O desenvolvimento da aplicação foi dividida em várias etapas. Esta secção focar-se-á na descrição mais pormenorizada de cada uma dessas etapas. Na figura 4.1 podemos observar o diagrama de blocos com a sequência de todos os passos da arquitectura proposta.

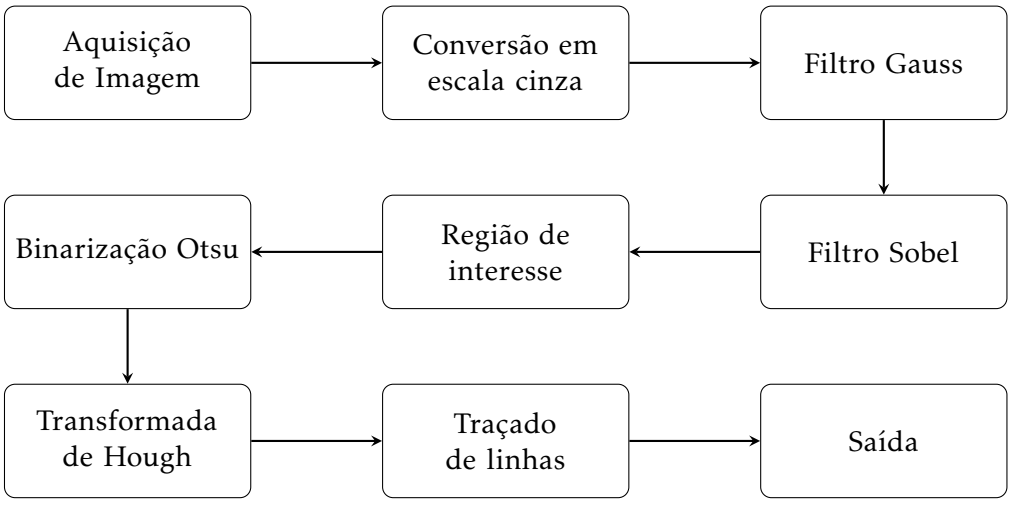


Figura 4.1: Diagrama de blocos da arquitectura do sistema.

4.1.1 Aquisição de imagem

As imagens provenientes do sensor de imagem Pcam 5C, cujos dados são do formato *stream* de vídeo AXI4 (`hls::stream`), são convertidas para imagens do formato `hls::Mat` com a função `hls::AXIvideo2Mat`.

No formato Mat as imagens podem ser trabalhadas com a linguagem de programação C/C++, incluindo também as funções da biblioteca *Video* de Vivado HLS, que permite a síntese e implementação das funções como *hardware* de alto desempenho. Relembro que as funções da biblioteca *Video* são programadas para acelerar as funções correspondentes de OpenCV, que têm assinaturas e usos semelhantes.

A biblioteca OpenCV fornece `cv::Mat` para representar as imagens. No seguinte exemplo, a variável imagem é definida como 1080×1920 elementos do tipo `CV_8UC3`, imagem de 3 canais de inteiros sem sinal de 8 *bits*.

```
cv::Mat image(1080, 1920, CV_8UC3);
```

A biblioteca *Video* fornece um equivalente sintetizável, a classe `hls::Mat`, onde a imagem é definida da seguinte maneira:

```
hls::Mat<1080, 1920, HLS_8UC3> image;
```

O uso desta classe assegura que Vivado *High-Level Synthesis* pode determinar o tamanho da memória para ser usada no armazenamento de *hardware*. A classe `hls::Mat` também garante que a implementação depois da síntese é optimizada para *hardware*.

4.1.2 Conversão para escala de cinza

Todos os algoritmos dos passos subsequentes requerem que as imagens estejam em escala de cinza. Para o tal, as imagens originais são convertidas para a escala de cinza usando a função `hls::CvtColor` da biblioteca *Video*.

O tipo da conversão da função é definido pelo valor de código `HLS_RGB2GRAY` que usa a expressão $Y = 0,299 \times R + 0,587 \times G + 0,114 \times B$ para traduzir três intensidades numa. Para a reversão utiliza-se o código `HLS_GRAY2RGB`, onde $R = Y, G = Y, B = Y$.

```
hls::CvtColor<HLS_RGB2GRAY>(src, dst);
hls::CvtColor<HLS_GRAY2RGB>(src, dst);
```

4.1.3 Redução de ruído: Filtro Gaussiano

Neste trabalho foi usada a função `hls::GaussianBlur` da biblioteca *Video*, cuja máscara, de tamanho 5×5, é apresentada na figura 4.2. As margens da imagem foram prolongadas com um método muito semelhante ao método de duplicação das margens, mas com uma pequena diferença. Exemplificando, dada uma série de píxeis de entrada **ABCDEFGH**, os valores das margens são completados da seguinte forma: FEDCB|**ABCDEFGH**|GFEDC, em vez de EDCBA|**ABCDEFGH**|HGFED do método de duplicação.

$$\frac{1}{84}\begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 5 & 6 & 5 & 2 \\ 3 & 6 & 8 & 6 & 3 \\ 2 & 5 & 6 & 5 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}$$

Figura 4.2: *Kernel* aplicado à imagem na função hls::GaussianBlur.

```
hls::GaussianBlur<5,5>(src, dst, 5, 5);
```

4.1.4 Realce de contornos: Filtro Sobel

Para a implementação do filtro de Sobel foi usada a função `hls::Sobel` da biblioteca *Video*. Esta função tem duas limitações: 1) o único tamanho da máscara suportado é 3×3, 2) e só é possível estimar as derivadas verticais e horizontais separadamente.

A primeira limitação é irrelevante para este trabalho, mas a segunda tem de ser resolvida. Para a solucionar, os filtros de Sobel, vertical e horizontal, são calculados independentemente, e em seguida as duas imagens resultantes são somados com a função `hls::AddWeighted`, também da biblioteca *Video*.

Esta função, para além de somar de duas imagens, permite atribuir pesos diferentes aos elementos de cada imagem.

```
hls::Sobel<1,0,3>(src, dst_x);
hls::Sobel<0,1,3>(src, dst_y);
hls::AddWeighted(dst_x, x_wheight, dst_y, y_wheight, scalar, dst);
```

4.1.5 Região de interesse

A região de interesse, ou *Region Of Interest (ROI)*, é um rectângulo definido pelo utilizador que pode ser usado para limitar computações em certas composições dentro dos seus limites [64].

Na figura 4.3 está representada a região de interesse proposta, que foi baseada noutros trabalhos de detecção de faixa de rodagem, nomeadamente [65] e [66], para a implementação com os seus parâmetros, *HPT* (*Height Percentage Top* ou percentagem de altura de cima) e *HPB* (*Height Percentage Bottom* ou percentagem de altura de baixo). Ambos os parâmetros são valores inteiros que podem ter valores entre 0 e 100 que representam percentagens, e cuja soma nunca pode ser superior a 100. *HPT* representa a percentagem

de linhas do topo a eliminar e *HPB* representa a percentagem de linhas a eliminar da parte inferior da imagem.

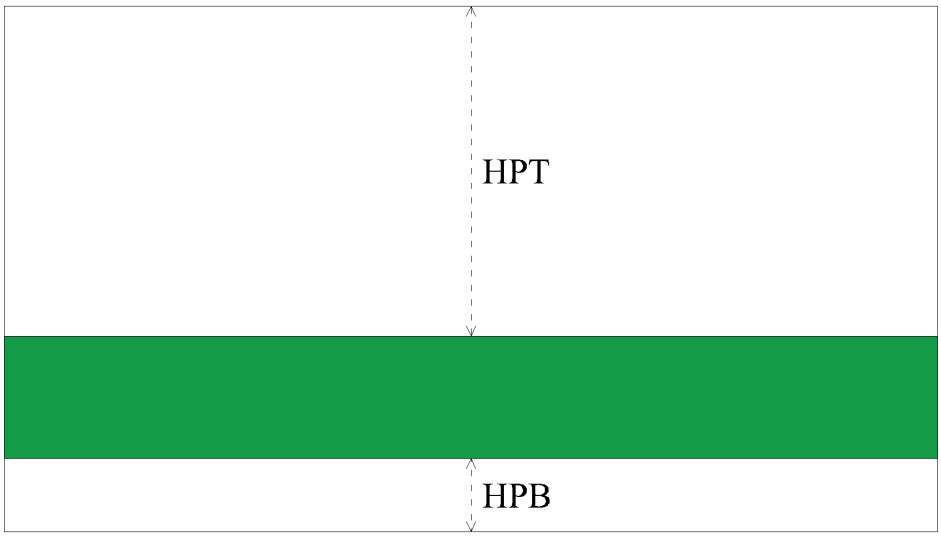


Figura 4.3: Região de interesse proposta.

Em primeiro lugar são calculados dois valores resultantes a partir dos parâmetros de entrada:

$$ht = \text{linhas da imagem} \times \text{HPT}/100 \qquad hb = \text{linhas da imagem} \times (100 - \text{HPB})/100 \tag{4.1}$$

Seguidamente, a medida que são percorridos os píxeis da imagem, esses tomam o valor de fundo (0 - preto) até atingir a linha cujo número é igual a *ht*. Entre as linhas *ht* e *hb* os píxeis da imagem mantêm os seus valores originais, resultando numa área útil como é exemplificado na figura 4.3.

O código completo da implementação em C encontra-se no Anexo I na Listagem I.2.

4.1.6 Binarização da imagem: Método de Otsu

Em primeiro lugar é preciso referir que a tradução para código C de todos os conceitos descritos na secção 2.3.3 acerca do método de Otsu encontra-se nos anexos I.1.

No presente trabalho o método de Otsu foi adaptado de modo a ignorar certas intensidades de escuro quando é calculado o histograma da imagem, porque o objectivo não é apenas separar as partes escuras das claras mas também, de certo modo, também diferenciar os tons claros para dar prioridade às linhas brancas das faixas de rodagem automóvel. Numa primeira passagem pelo algoritmo de Otsu, cujo histograma inclui todos os valores das intensidades dos píxeis dentro da região de interesse, é calculado o valor do limiar que separa os tons claros dos escuros, depois é efectuada uma segunda passagem onde são ignoradas fracções dos tons escuros. Por outras palavras, da primeira

passagem resulta um limiar t_1 que é usado na segunda passagem para desprezar todas as intensidades cujos valores ficam abaixo de t_1/x. O valor x é definido pelo utilizador, mas tipicamente tem o valor de 2 ou 3.

Uma vez que a imagem foi binarizada implementa-se uma técnica adicional de detecção de linhas baseada em convolução com o fim de reduzir a grossura das linhas resultantes do filtro de Sobel. Para o tal foram aplicados quatro filtros que identificam as linhas em quatro direcções, horizontal (R_1), vertical (R_2) e oblíquas (±45 graus, R_3 e R_4). As máscaras são apresentadas na figura 4.4 e na práctica o resultado das convoluções é $R(x,y) = max(\left|R_1(x,y)\right|, \left|R_2(x,y)\right|, \left|R_3(x,y)\right|, \left|R_4(x,y)\right|)$.

-1	-1	-1
2	2	2
-1	-1	-1

-1	2	-1
-1	2	-1
-1	2	-1

-1	-1	2
-1	2	-1
2	-1	-1

2	-1	-1
-1	2	-1
-1	-1	2

Figura 4.4: Máscaras de convolução usados para detectar linhas horizontais (R_1), verticais (R_2), oblíquas (+45 graus, R_3), oblíquas (−45 graus, R_4).

Esta operação, para além de reduzir a área útil da imagem, que resultará numa implementação mais eficiente da transformada de Hough, também acaba por funcionar como um filtro de redução de ruído adicional.

Mais uma vez, a implementação em código C pode ser encontrada no Anexo I na Listagem I.4.

4.1.7 Extracção de características: Transformada de Hough

Para implementar a transformada de Hough, mais uma vez, foi usada uma função da biblioteca *Video*, neste caso `hls::HoughLines2`.

Esta função tem como parâmetros de entrada a imagem binarizada na qual se tenciona encontrar linhas rectas, e um *threshold*, que indica o número mínimo de píxeis que têm de "aterrar"na linha antes de ela ser considerada como tal. Este *threshold* é calculado a partir de outro parâmetro definido pelo utilizador, mais uma vez, em forma de percentagem (ML de *Min Length*, ou comprimento mínimo).

Para exemplificar, considere-se que está a ser tratada uma imagem com resolução de 1280×720. Considerem-se também os parâmetros definidos pelo utilizador, $HPT = 70$, $HPB = 10$ e $ML = 40$, valores típicos que serão discutidos mais adiante nos resultados. A altura da região de interesse é igual a $roiH = (linhasdaimagem \times (100 - HPT - HPB))/100$, logo, $roiH = (720 \times (100 - 70 - 10))/100 = 144$. Por fim, o número mínimo de pontos para se ser considerada uma linha é igual a $(roiH \times ML)/100 = 58$ píxeis.

O único parâmetro de saída é um vector com as linhas encontradas. Cada linha é representada por um vector de dois elementos (ρ, θ), onde ρ é a distância da coordenada

da origem $(0,0)$ (conto superior esquerdo da imagem) e θ é o ângulo de rotação da linha em radianos. Estes dados serão utilizados na próxima, e última etapa, para traçar as linhas na imagem de entrada.

4.1.8 Traçado de linhas

O processo do traçado das linhas da faixa também é dividido em várias fases, que podem ser resumidas com a seguinte ordem de trabalhos:

1. traduzir as coordenadas polares das linhas para coordenadas cartesianas;
2. filtrar e classificar as linhas resultantes da transformada de Hough para encontrar os vários tipos de linhas da faixa de rodagem;
3. traçar as linhas;

A única informação acessível das linhas rectas encontradas na imagem são dois valores reais obtidos da função Hough: ρ medido em píxeis e θ em radianos.

O valor do ângulo θ é utilizado para calcular os valores de *seno* e *coseno* do ângulo com a ajuda da função *sincos* da biblioteca *cordic* (`hls::cordic::sincos`). Agora já se dispõe de todos os dados para converter a linha do espaço de parâmetros para os espaço xy.

Em primeiro lugar calculam-se as coordenadas do ponto P_0, utilizando as expressões 4.2. Tendo o ponto inicial e o ângulo podemos calcular as extremidades da linha. Considere-se como extremidade superior, P_1, a linha onde começa a região de interesse, e como extremidade inferior, P_2, o limite inferior da imagem.

$$P_0.x = \rho \times \cos(\theta) \qquad P_0.y = \rho \times \sin(\theta) \tag{4.2}$$

Já foi definido que a região de interesse tem o começo na linha $y = (\textit{linhas da imagem} \times HPT)/100$, (ver secção 4.1.5), logo $P_1.y$ será igual a esse valor. De seguida calcula-se a distância A do ponto P_0 até P_1, que é obtida pela expressão $A = (P_1.y - P_0.y)/cos(\theta)$, e por fim calcula-se $P_1.x = P_0.x + A \times (-sin(\theta))$. Seguindo a o mesmo procedimento calcula-se o ponto P_2 a partir de P_0.

$$P_1.y = linhas \times HPT/100 \quad A = (P_1.y - P_0.y)/\cos(\theta) \quad P_1.x = P_0.x + A \times (-\sin(\theta)) \tag{4.3}$$

$$P_2.y = linhas \quad B = (P_2.y - P_0.y)/\cos(\theta) \quad P_2.x = P_0.x + B \times (-\sin(\theta)) \tag{4.4}$$

O passo seguinte consiste em dividir as linhas em 5 tipos admitidos. Esses tipos são: as duas linhas da faixa de rodagem principal, na qual está localizado o carro, (esquerda e direita), que na imagem final serão representadas pela cor verde; duas potenciais linhas de faixa, que apesar de o algoritmo não as aceitar como linhas principais também não as

descarta completamente, que terão representação vermelha; e uma linha quase vertical quando é efectuada a transição de uma faixa para outra, que será representada pela cor azul.

Da transformada de Hough podem resultar várias linhas com variações mínimas entre si. Para filtrar as linhas semelhantes na figura 4.5 pode ser observado o método proposto (as distâncias entre as linhas foram propositamente exageradas para melhor ilustrar o procedimento).

Considere-se o caso das linhas do lado direito. A primeira linha a ser considerada como uma linha nova é a azul. Os pontos limites dessa linha, P_0 e P_1, inicializam os pontos mínimos e máximos da linha final. De seguida considera-se linha azul ciano. Se ela não ultrapassar uma distância definida da primeira linha, actualizam-se os pontos da linha final, $P_0.min = P_0$ e $P_1.max = P_1$. O mesmo é efectuado à vermelha. Quando aparecer uma linha que se afasta significativamente, neste caso linha magenta, são calculados os pontos da linha final, linha verde, usando as expressões simples 4.5, e inícia-se o mesmo processo para as linhas seguintes. No final, 5 linhas da transformada de Hough deram lugar a apenas 2 linhas (verdes).

$$P_0 = (P_0.min + P_0.max)/2 \qquad P_1 = (P_1.min + P_1.max)/2 \tag{4.5}$$

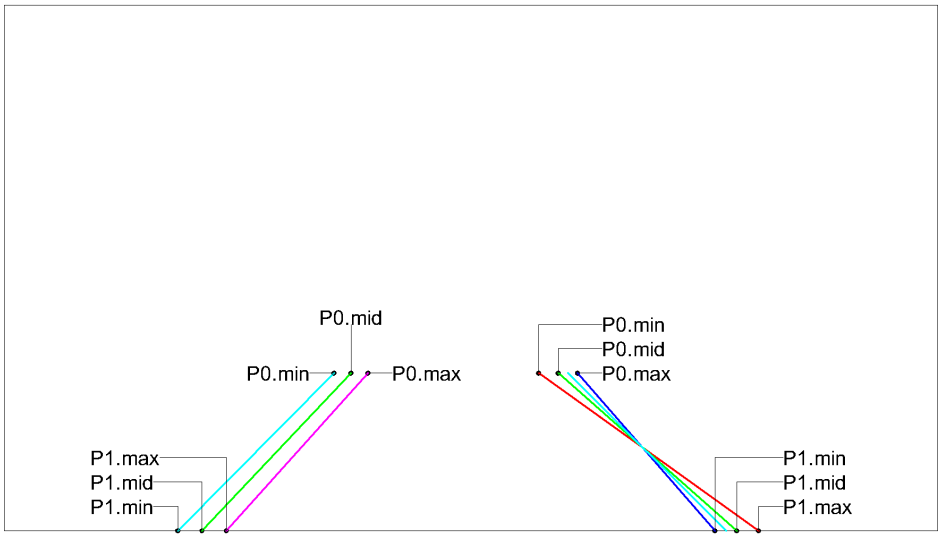


Figura 4.5: Método proposto de escolha de linhas.

Como já foi mencionado mais acima, no máximo serão admitidas 5 linhas. O processo de selecção de cada tipo é o seguinte:

- as linhas da faixa de rodagem, esquerda e direita, têm de ter as suas extremidades obrigatoriamente dentro dos limites da imagem, tanto P_0 como P_1, e os seus ângulos, baseado em [67], entre 30 e 80 graus, no caso das linhas esquerdas, e entre 100 e 150

graus no caso das linhas direitas. Além disso, as duas extremidades têm de fazer parte da mesma metade da imagem, esquerda ou direita.

- para uma linha ser considerada como uma potencial linha de faixa o seu limite superior, P_0, tem de se encontrar dentro da imagem, ou seja, entre zero e ponto médio da imagem, no caso da linha esquerda, e entre o ponto médio e o número de colunas da imagem, no caso da linha direita. Ao contrário das linhas da faixa, os seus limites inferiores têm necessariamente de estar fora dos limites da imagem. No caso da linha esquerda $P_1.x$ tem de ser inferior a zero, e no caso da linha direita $P_1.x$ tem de ser maior que o número de colunas da imagem.

- por fim, a linha de transição de faixa, é uma linha que não é selecionada para os outros tipos e ainda cumpre outros requisitos, nomeadamente, ter ângulo entre 0 e 30 graus, e ter as suas extremidades dentro da região que não se afasta muito do ponto médio da imagem, mais precisamente, uma distância que equivale a um quinto da largura da imagem.

Em todos os casos, se para com um certo tipo emergirem varias potenciais linhas com características válidas, a linha cujos limites se localizarem mais perto do centro da imagem será a escolhida.

Uma vez definidos todos os tipos de linhas, essas podem ser traçadas na imagem original. Para traçar uma linha numa só passagem contínua pela imagem, ou seja, da esquerda para a direita, de cima para baixo, na implementação do presente trabalho foram efectuadas algumas alterações ao algoritmo proposto pelos autores de [56] descrito na secção 2.3.5, pois nesse algoritmo o x e y, portanto, a posição do píxel da linha na imagem, podem incrementar e diminuir.

A diferença no programa só é notada quando se pretende traçar uma linha da direita para esquerda, e acaba por traduzir-se da seguinte maneira: sempre que o valor x do próximo píxel da linha diminuir mas o valor de y permanecer inalterado, procura-se o próximo píxel da linha até o valor de y incrementar.

Isto pode resultar em algumas descontinuidades nas linhas, mas acaba por não ser problemático uma vez que depois de ser traçada a linha será efectuada uma operação de dilatação, que para além de fechar os buracos das linhas, engrossa-as para facilitar o seu visionamento no ecrã.

Cada tipo de linha será traçada numa variável de imagem `hls::Mat` separadamente. Isto possiblita usar a função `hls::PaintMask` da biblioteca *Video*, que permite juntar uma máscara à uma imagem, podendo essa máscara ser pintada de uma cor qualquer.

```
hls::PaintMask(src, mask, dst, color);
```

Deste modo às linhas da faixa são pintadas de verde, as potenciais linhas de vermelho, e a linha de transição de faixa de cor azul.

A listagem completa do algoritmo é apresentada no Anexo I na Listagem I.5.

4.2 Implementação em *Hardware*

Para implementar o algoritmo proposto na plataforma escolhida, Zybo Z7-20, foram utilizados como ponto de partida outros projectos disponibilizados pela Digilent, nomeadamente, o projecto *Zybo Z7 Pcam 5C Demo* ([68]) e *DIGILENT Embedded Vision Demo* ([69]).

O projecto *Zybo Z7 Pcam 5C Demo* tem como objectivo demonstrar como usar o módulo de imagem Pcam 5C como uma fonte de vídeo, e como é possível encaminhar os dados de imagem para o porto HDMI TX.

Embedded Vision Demo é uma demonstração distribuída pela Digilent que pretende demonstrar o uso de *Digilent Embedded Vision Bundle*, Xilinx Vivado, Xilinx Vivado HLS, e Xilinx SDK para a criação de sistemas de processamento vídeo. *Digilent Embedded Vision Bundle* é o nome atribuído ao conjunto da placa de desenvolvimento Zybo Z7-20 e módulo de câmara Pcam 5C.

Nesta demonstração o módulo Pcam 5C fornece, em tempo real, vídeo de alta-definição à entrada HDMI da placa Zybo Z7-20. Dois interruptores (sw0 e sw1) da placa foram configurados para escolher entre três operações de processamento vídeo, detecção de bordas (*Blur Edge Detection*), inversão de cor (*Invert*) e conversão para o nível de cinza (*Color to B/W*). Por fim, os sinais processados são enviados para a saída HDMI de Zybo Z7-20 para a sua apresentação final num ecrã de alta definição (1280×720).

Esta demonstração abrange todos os processos de aquisição de dados vídeo, o seu processamento e a sua saída. Na figura 4.6 está a representação simplificada do canal vídeo (*video pipeline*), já com as devidas alterações realizadas, ou seja, no processamento de imagem, os três IPs mencionados foram substituídos pelo IP HLS, a verde na imagem, com a implementação do Algoritmo de Detecção de Faixa de Rodagem Automóvel, com o nome *Lane Detection* (IP HLS).

4.2.1 Núcleos IP do *Video Pipeline*

Nesta secção serão descritos os núcleos IP que são usados no integrador IP de Vivado. Os IPs são criados pela Digilent, são *open source* e podem ser descarregados do *Github* oficial da Digilent.

- **MIPI_D_PHY_RX** - Implementa a camada física da interface *MIPI D-PHY*. Os dados são desserializados em duas faixas de dados que são combinadas conjuntamente.

- **MIPI_CSI_2_RX** - Implementa a camada do protocolo da interface *MIPI Camera Serial Interface 2 (CSI-2)*. Interpreta os dados como píxeis, linhas, e *frames*. Os dados vídeo são formatados como dados brutos RGB e são empacotados numa interface *AXI-Stream* que formam a entrada do *video pipeline*.

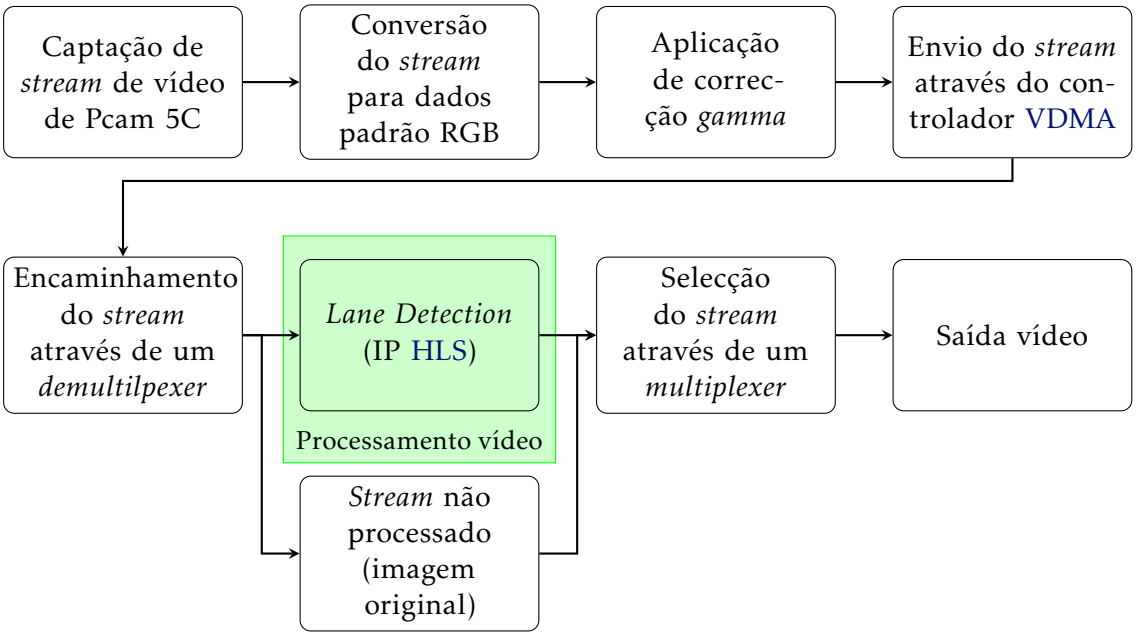


Figura 4.6: Diagrama de blocos do *video pipeline* (adaptado de [69]).

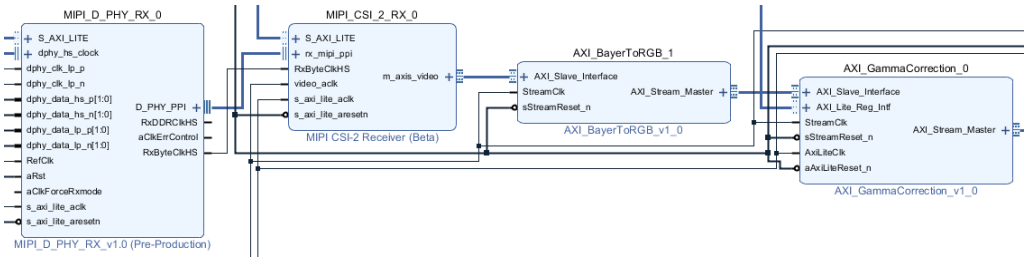


Figura 4.7: MIPI RX e pré processamento [69].

- **AXI_BayerToRGB** - Interpola os dados brutos RGB filtrados pelo filtro de Bayer em dados RGB padrão (32 *bits*: 2 de *padding*, e 10 para cada cor, *red*, *green*, *blue*).

- **AXI_GammaCorrection** - Aplica a correcção *gamma* ao *stream* de vídeo, e reduz a profundidade de cor para 8 *bits* por cor (24 *bits*: 8 *red*, 8 *green*, 8 *blue*). O bloco aplica à imagem um dos cinco factores *gamma* possíveis: 1, 1/1.2, 1/1.5, 1/1.8 ou 1/2.2.

- **AXI_Video Direct Memory Access** - Implementa três *frame buffers* em memória DDR permitindo acesso de dentro do processador. Este bloco é conectado ao subsistema de memória do Zynq através de dois portos alto desempenho de acesso à memória. A duplicação da taxa de entrada das imagens de 30Hz para 60Hz é alcançado com *frame buffering*.

- **AXI4-Stream Switch 0** - Efectua a multiplexagem do *stream* entre dois locais: a passagem da imagem original ou algoritmo de detecção de faixas.

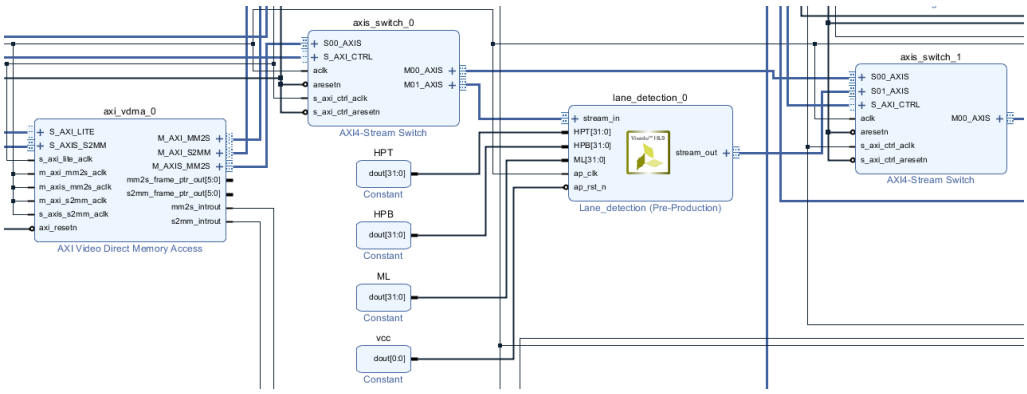


Figura 4.8: *Stream* Vídeo e *Mux/Demux*.

- **AXI4-Stream Switch 1** - Executa a multiplexagem inversa do *stream* vídeo para uma fonte.

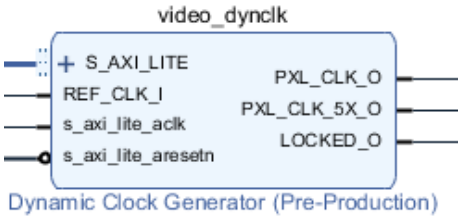


Figura 4.9: Gerador dinâmico de relógio [69].

- **Dynamic Clock Generator** - Gera o relógio dos píxeis usado para a saída HDMI. A frequência de relógio dos píxeis muda de acordo com a resolução da saída definida pelo processador.

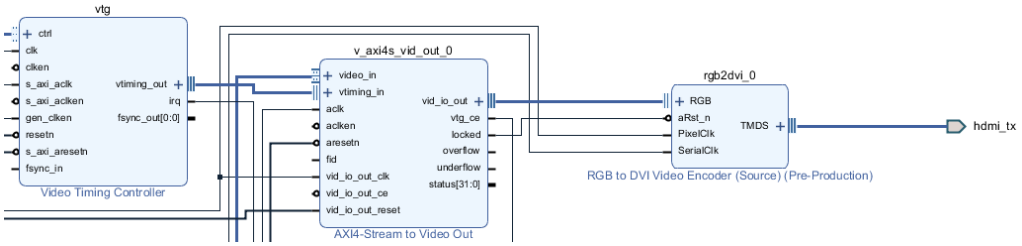


Figura 4.10: Saída vídeo [69].

- **Video Timing Controller** - Gera os sinais de controlo vídeo usados pela saída HDMI, incluindo *VSync*, *HSync*, *VBlank* e *HBlank*.
- **AXI4-Stream to Video Out** - Converte o *stream* vídeo AXI para uma interface de saída. Este bloco usa os sinais vídeo do controlador do *timing* de vídeo e emparelha-os com os dados RGB provenientes do *demux*.
- **RGB to DVI** - Codifica os dados RGB como DVI para a saída HDMI.

4.2.2 *Software*

O código do processador é escrito em Xilinx SDK. Em primeiro lugar, o programa inicializa o *hardware* usado no projecto e depois verifica se existem alterações nos interruptores para encaminhar o *stream* de vídeo.

Na inicialização do *hardware* são executados os seguintes passos:

1. A plataforma principal é inicializada. Isto activa as *caches* e depois repõe o PL do FPGA;
2. É inicializado o controlador de interrupções;
3. São inicializados os controladores GPIO (pinos de activação da câmara) e I2C e são conectados ao controlador de interrupções no *software*;
4. São inicializados os interruptores, o *multiplexer* e o *demultiplexer* de *stream*;
5. A câmara é então inicializada usando o I2C e os controladores de activação da câmara;
6. O VDMA é inicializado e é ligado ao controlador de interrupções;
7. São inicializados o gerador de relógio dinâmico (*dynamic clock generator*) e o controlador de *timing* de vídeo;
8. É configurado e inicializado o formato de vídeo. A resolução escolhida é 1280×720.

A listagem completa pode ser encontrada no Anexo II na Listagem II.1.

4.3 Resultados de Simulações *High-Level Synthesis*

Nesta secção são apresentados alguns resultados obtidos nas simulações em Vivado HLS que foram considerados como os mais representativos das funcionalidades, capacidades e limitações da arquitectura proposta.

4.3.1 Parâmetros Definidos pelo Utilizador e Programador

Em várias etapas da secção da arquitectura do sistema foram mencionados parâmetros definidos pelo utilizador, ou programador. Testaram-se muitas combinações, e por fim foram escolhidos os parâmetros que produziram os melhores resultados para a maior parte dos casos.

Seguindo a ordem das etapas do algoritmo proposto, primeiro foram definidos os valores dos parâmetros de programador do filtro de Sobel. Quando são somados os resultados das derivadas em x e y com a função `AddWeighted`, atribuiu-se o peso 2 para a direcção x e 4 para a direcção y. Foram testadas todas as combinações com os valores entre 1 e 4 para ambas as direcções.

Na região de interesse, para HPT foi definido o valor 70 porcento e para HPB 10 porcento. Estes parâmetros numa implementação comercial seriam definidos pelo utilizador.

No método de Otsu para calcular o limiar da segunda passagem (t_2) necessita de desprezar uma fracção das intensidades do limiar da primeira passagem (t_1). Os melhores resultados surgiram quando foram ignoradas todas as intensidades abaixo de $t_1/3$, uma vez que quando foram testadas com $t_1/2$ ou $t_1/1$, perdia-se demasiada informação da faixa de rodagem. Este parâmetro é de programador.

Na transformada de Hough é preciso definir dois parâmetros, o número mínimo de píxeis numa linha (ML) e o número máximo de linhas que a transformada pode encontrar. Para ML, parâmetro de utilizador, foi estabelecido o valor 40 e para o número máximo de linhas permitidas, parâmetro de programador, 50.

4.3.2 Resultados

Na figura 4.11 podemos observar resultados obtidos em casos onde as faixas são rectas, e onde as linhas estão bem definidas e existe boa visibilidade. Estes são os melhores casos possíveis para o algoritmo e parâmetros propostos. Nestes casos a arquitectura falha poucas vezes.

(a) Imagem original.

(b) Resultado do pré-processamento.

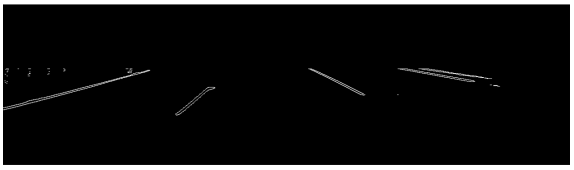
(c) Resultado final.

Figura 4.11: Exemplo 1 - Faixa recta.

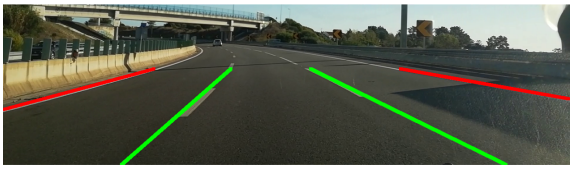
Outros exemplos onde os resultados apresentados são satisfatórios são os casos das autoestradas, onde mais uma vez, os limites das faixas estão bem definidos e as curvas não são muito acentuadas. O exemplo 4.12 apresenta os resultados obtidos numa curva ligeira para o lado esquerdo e o exemplo 4.13 para o lado direito. Em ambos os exemplos foram identificadas as faixas de condução (verdes) e as faixas laterais, ou potenciais linhas de faixa (vermelhas). É pertinente chamar a atenção para o facto de que é possível distinguir

quando se está perante uma curva ou uma faixa recta. Nas faixas rectas as suas linhas são quase simétricas, enquanto nas faixas curvas uma das linhas é bastante mais inclinada que a outra. Mais, se a linha mais inclinada for a direita, então estamos perante uma curva que se dirige para a esquerda, e o contrário também se verifica, se a linha mais inclinada for a esquerda, então a curva tende para o lado direito.

(a) Imagem original.



(b) Resultado do pré-processamento.

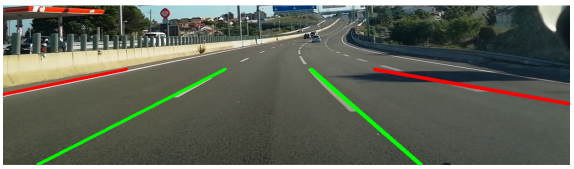


(c) Resultado final.

Figura 4.12: Exemplo 2 - Faixa com curva ligeira para o lado esquerdo.

(a) Imagem original.

(b) Resultado do pré-processamento.



(c) Resultado final.

Figura 4.13: Exemplo 3 - Faixa com curva ligeira para o lado direito.

No lado direito das figuras 4.13b e 4.14b podemos observar o efeito que as sombras podem provocar, linhas adicionais, neste caso linhas horizontais, que podem influenciar

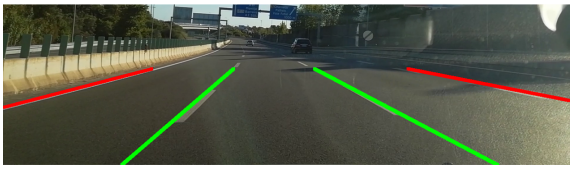
o algoritmo de selecção de linhas, mas como este é programado para ignorar as linhas horizontais os resultados acabam por ser satisfatórios.

No entanto, também no exemplo 4.14 encontramos uma das limitações da função `hls::HoughLines2`. Se o número máximo de linhas for 50, como nos exemplos anteriores, então não será encontrada uma linha esquerda da faixa, uma vez que, a partir do resultado do pré-processamento com o parâmetro $ML = 40$, a função `hls::HoughLines2` consegue descobrir mais do que 50 linhas, muitas com diferenças mínimas entre si (ver secção 4.1.8). Para resolver este problema têm de ser disponibilizadas mais linhas para a função (4.14d), mas isto acaba por incrementar muito o uso dos recursos de memória do FPGA, o que faz com que esta solução não seja viável numa implementação práctica.

(a) Imagem original.

(b) Resultado do pré-processamento.

(c) Resultado final com 50 máximas linhas.



(d) Resultado final com 100 máximas linhas.

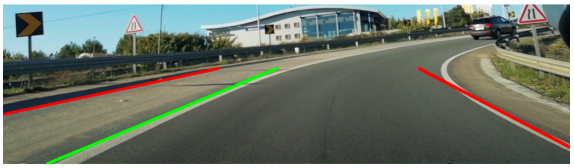
Figura 4.14: Exemplo 4 - Faixa recta com parâmetros diferentes.

Na figura 4.15 temos um caso de curvas mais acentuadas. Nestes casos as linhas calculadas não se sobrepõem exactamente sobre as linhas da faixa porque no algoritmo proposto não é efectuado o chamado *lane tracking*, logo as linhas são umas aproximações de primeira ordem inexactas. Ainda assim, é possível retirar alguma informação útil, como, para que direcção tende a curva, seguindo o mesmo raciocínio das curvas menos acentuadas.

O último caso abordado é um exemplo com uma linha vertical, linha de transição de faixa (4.16). Neste tipo de casos o algoritmo de detecção de faixa proposto apresenta os

(a) Imagem original.

(b) Resultado do pré-processamento.



(c) Resultado final.

Figura 4.15: Exemplo 5 - Curva acentuada.

piores resultados. O problema deve residir no método de detecção de linhas, ou seja, o pré-processamento, que como se pode observar na figura 4.16b, a linha do meio não apresenta pontos de referência suficientes para que a transformada de Hough as conseguir detectar.

(a) Imagem original.

(b) Resultado do pré-processamento.

(c) Resultado final.

Figura 4.16: Exemplo 6 - Linha de transição de faixa.

4.4 Resultados de Implementações em *Hardware*

Uma vez finalizadas todas as simulações em Vivado HLS, os próximos passos são sintetizar e exportar o código C.

Quando foi sintetizada a arquitectura proposta completa revelou-se que a plataforma disponível, Zybo Z7-20, não tinha recursos suficientes para acomodar o algoritmo inteiro. Após alguma análise mais detalhada do relatório elaborado pelo Vivado HLS ficou apurada a causa principal. A transformada de Hough mesmo sendo uma função da biblioteca *Video* de Vivado HLS, optimizada e acelerada, é computacionalmente intensiva e requer muitos recursos. Só a sua síntese necessitaria de mais memória do que a placa tem disponível.

Por isso, para pelo menos testar a aquisição de dados vídeo e umas funções básicas de pré-processamento foi sintetizada e exportada uma versão muito mais aligeirada da arquitectura, uma implementação parcial da arquitectura. Só alguns passos foram implementados, e mesmo assim, alguns desses métodos tiveram de ser alterados para o sistema trabalhar correctamente. Mas compreender as razões de porquê o IP sintetizado e exportado sem apresentar quaisquer erros não resulta num funcionamento normal do sistema, será tema para trabalhos futuros.

Na tabela 4.1 podem ser observados os resultados das exportações da arquitectura completa e de uma arquitectura parcial, ou seja, os recursos exigidos pelos IPs resultantes de Vivado HLS. Na arquitectura parcial apenas são executados os passos de conversão de cor para escala cinza, aplicação do filtro Gaussiano, aplicação do filtro de Sobel, escolha da região de interesse e, em vez de binarização pelo método de Otsu, a imagem é binarizada através de uma função simples de *thresholding* manual, cujo valor do limiar definido foi 125.

Implementação	**Completa**	**Parcial**	**Disponível**
LUT	133.902	3.868	53.200
Flip-Flops	58.274	33.49	106.400
Blocos RAM (18Kb)	544	21	280
Parcelas DSP	208	56	220

Tabela 4.1: Comparação entre as duas implementações em *hardware*.

Na figura 4.17 pode ser observado o processo descrito na secção 4.2 e representado na figura 4.6. Como os dois interruptores (sw0 e sw1) de Zybo Z7-20 têm o valor 0, o *stream* de vídeo não é processado, mas sim, enviado directamente para o monitor.

Quando o interruptor sw0 passa de 0 para 1 o FPGA executa o programa *Lane Detection* parcial dentro do IP como está representado na figura 4.18.

O IP sintetizado da arquitectura completa necessitaria de pelo menos 2,6 vezes mais *Lookup-Table*s e 2,0 vezes mais blocos RAM (de 18Kb) do que a Zybo Z7-20 tem disponíveis. O próximo modelo Zynq da mesma família, Zynq-7000 SoC, que tem recursos

Figura 4.17: *Stream* não processado.

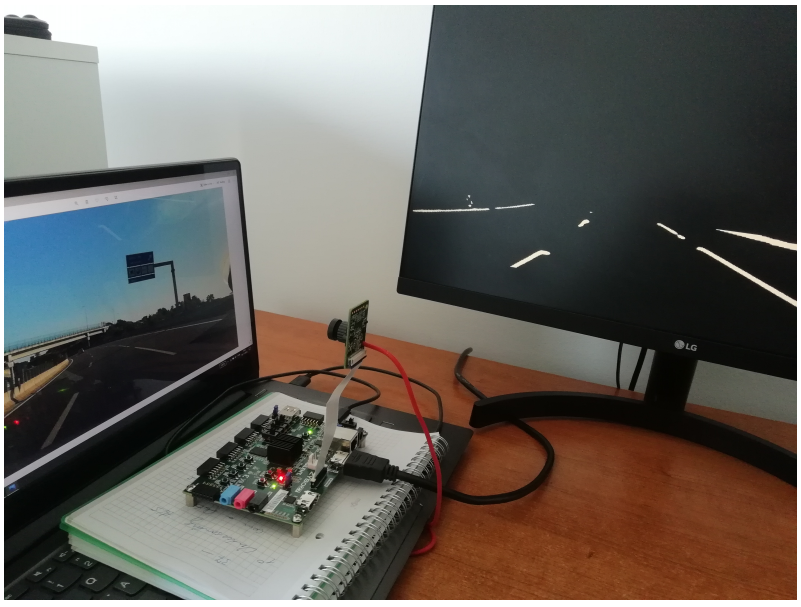


Figura 4.18: *Lane Detection* parcial.

suficientes é Z-7035 ([58]), mas este tem um preço acima de 1.500€, o que vai contra os objectivos inicialmente propostos de apresentar os FPGAs competitivos financeiramente.

Capítulo 5

Conclusões e Trabalhos Futuros

Para concluir, é feita uma reflexão relativamente às dificuldades sentidas ao longo da realização desta dissertação, e com base na análise desses aspectos, serão propostos trabalhos futuros e possíveis melhorias ao sistema.

5.1 Conclusões

No geral, por um lado esta dissertação permitiu validar alguns conceitos utilizados nos algoritmos de detecção de faixa de rodagem, tais como, na fase de pré-processamento é utilizado o filtro de Sobel para detectar os bordos das linhas das faixas na imagem, como a região de interesse reduz o espaço de processamento e o número de computações necessárias, e como o método de Otsu é robusto no cálculo automático do limiar que separa as intensidades escuras das claras. Ou como a transformada de Hough, que apesar de ter limitações de implementação, é impressionante na detecção de linhas rectas nas simulações.

Os resultados obtidos na validação do algoritmo apresentado, simulações no ambiente Vivado HLS, foram satisfatórios, tanto na detecção de vários tipos de linhas de faixa nos casos relativamente simples, faixas rectas com boas condições de visibilidade, como nos casos um pouco mais complexos, onde as faixas apresentavam curvas ligeiras e onde podiam aparecer factores externos, como por exemplo sombras.

Por outro lado, um dos principais objectivos deste trabalho foi desenvolver e implementar um sistema de visão computacional completo baseado em FPGA, usando apenas linguagens de programação comuns e acessíveis, visto que, construir sistemas com linguagens de descrição de *hardware* é bastante mais trabalhoso e demorado, mas devido a falta de recursos da plataforma alvo este objectivo não foi cumprido na sua totalidade.

Neste trabalho as maiores dificuldades surgiram na implementação da arquitectura

proposta em *hardware*. As funções e os algoritmos requeriam muitos mais recursos do que Zybo Z7-20 tinha disponíveis. A escolha da transformada de Hough para o algoritmo proposto, um método mais tradicional e muito investigado nos sistemas de detecção de faixa de rodagem, revelou-se ineficiente e caro demais em termos de recursos.

5.2 Trabalhos Futuros

Tendo em conta a conclusão obtida, uma sugestão inicial é encontrar outro método de extração de características, ou seja, substituir o método da transformada de Hough visto que esse requer muitos recursos. Uma possibilidade é investigar com mais cuidado as alternativas existentes, como por exemplo, trabalhos que utilizam algoritmos RANSAC para estimar as linhas ([70], [71]).

Outro ponto importante seria estudar todas as potencialidades do ambiente *High-Level Synthesis* de Vivado, nomeadamente, as imensas directivas de melhoria de desempenho que podem ser aplicadas às funções e aos dados, quer para melhorar o paralelismo, quer de melhoria do *pipeline*, melhoria da área ou redução da latência do sistema. O documento com o guia das metodologias de optimização Vivado HLS (*Vivado HLS Optimization Methodology Guide* [62]) é bastante extenso e requer uma especial atenção e compromisso.

No algoritmo proposto de detecção de faixa de rodagem podem ser acrescentadas mais funcionalidades para enriquecer o seu funcionamento e torná-lo mais interventivo. Uma vez que do processo de selecção de linhas conhecem-se os seus valores das extremidades, estes podem ser utilizados para indicar ao condutor em que direcção segue a estrada, fazendo uns cálculos das diferenças entre ângulos das faixas, como descritos na secção 4.3.2.

Outra habilidade a acrescentar poderá ser um sistema simples de *Lane Tracking* que calcula o desvio do ponto médio da imagem de entrada, ou seja, a posição do carro na via, em relação ao ponto médio das linhas de faixa detectadas. Ou então pesquisar métodos um pouco mais sofisticados.

Por fim, tentar tirar partido da capacidade mais diferenciadora dos FPGAs, a sua reconfiguração dinâmica e parcial. Tal como foi referido na secção 2.1.3, a reconfiguração parcial pode permitir uma aplicação maior poder ser contida num *chip* mais pequeno, que é um problema relevante no presente trabalho.

Bibliografia

[1] Acharya. *Image Processing: Principles and Applications [book review]*. Vol. 18. 2. 2007, pp. 1–6. ISBN: 9780471719984. DOI: 10.1109/tnn.2007.893088.

[2] Y. Xing, C. Lv, L. Chen, H. Wang, H. Wang, D. Cao, E. Velenis e F. Wang. "Advances in Vision-Based Lane Detection: Algorithms, Integration, Assessment, and Perspectives on ACP-Based Parallel Vision". Em: *IEEE/CAA Journal of Automatica Sinica* 5.3 (2018), pp. 645–661. DOI: 10.1109/JAS.2018.7511063.

[3] B. S. C. Varma, K. Paul e M. Balakrishnan. *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*. Vol. 55. 2016. Cap. 2.2. ISBN: 978-981-10-0589-3. DOI: 10.1007/978-981-10-0591-6. URL: http://link.springer.com/10.1007/978-981-10-0591-6.

[4] I. Kuon e J. Rose. "Area and delay trade-offs in the circuit and architecture design of FPGAs". Em: *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA* (2008), pp. 149–158. DOI: 10.1145/1344671.1344695.

[5] E. Ahmed. "The Effect of Logic Block Granularity on Deep-Submicron FPGA Performance and Density". Em: 12.3 (2001), pp. 288–298.

[6] H. Gao, Y. Yang, X. Ma e G. Dong. "Analysis of the effect of LUT size on FPGA area and delay using theoretical derivations". Em: *Proceedings - International Symposium on Quality Electronic Design, ISQED* (2005), pp. 370–374. ISSN: 19483287. DOI: 10.1109/ISQED.2005.20.

[7] R. Oshana. *Multicore Software Development Techniques: Applications, Tips, and Tricks*. 2015, pp. 1–221. ISBN: 9780128010372. DOI: 10.1016/C2013-0-15268-3.

[8] G. F. Weidle, F. Viel, D. R. De Melo e C. A. Zeferino. "A Hardware Accelerator for Anisotropic Diffusion Filtering in FPGA". Em: *Proceedings - IEEE International Symposium on Circuits and Systems* 2018-May.5 (2018). ISSN: 02714310. DOI: 10.1109/ISCAS.2018.8351279.

[9] T. H. Tsai, Y. C. Ho e M. H. Sheu. "Implementation of FPGA-based Accelerator for Deep Neural Networks". Em: *Proceedings - 2019 22nd International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2019* 7 (2019), pp. 73–76. DOI: 10.1109/DDECS.2019.8724665.

[10] Z. Zhao, F. Wang e Q. Ni. "An FPGA-based hardware accelerator of ransac algorithm for matching of images feature points". Em: *Proceedings of International Conference on ASIC* (2019), pp. 8–11. ISSN: 2162755X. DOI: 10.1109/ASICON47005.2019.8983656.

[11] R. MacHupalli, H. Zhou e M. Mandal. "Hardware Accelerator for Nuclei Detection in Histopathology Images". Em: *2019 IEEE Canadian Conference of Electrical and Computer Engineering, CCECE 2019* (2019), pp. 1–4. DOI: 10.1109/CCECE.2019.8861896.

[12] F. Siddiqui, S. Amiri, U. I. Minhas, T. Deng, R. Woods, K. Rafferty e D. Crookes. "FPGA-based processor acceleration for image processing applications". Em: *Journal of Imaging* 5.1 (2019). ISSN: 2313433X. DOI: 10.3390/jimaging5010016.

[13] K. F. K. Wong, V. Yap e T. Peh Chiong. "Hardware accelerator implementation on FPGA for video processing". Em: *2013 IEEE Conference on Open Systems, ICOS 2013* (2013), pp. 47–51. DOI: 10.1109/ICOS.2013.6735046.

[14] J. Oliveira, A. Printes, R. C. Freire, E. Melcher e I. S. Silva. "FPGA architecture for static background subtraction in real time". Em: *SBCCI 2006 - 19th Symposium on Integrated Circuits and Systems Design* 2006 (2006), pp. 26–31. DOI: 10.1145/1150343.1150356.

[15] U. Muehlmann, M. Ribo, P. Lang e A. Pinz. "A new high speed CMOS camera for real-time tracking applications". Em: *Proceedings - IEEE International Conference on Robotics and Automation* 2004.5 (2004), pp. 5195–5200. ISSN: 10504729. DOI: 10.1109/robot.2004.1302542.

[16] W. He e K. Yuan. "An improved canny edge detector and its realization on FPGA". Em: *Proceedings of the World Congress on Intelligent Control and Automation (WCICA)* (2008), pp. 6561–6564. DOI: 10.1109/WCICA.2008.4594570.

[17] N. H. Tan, N. H. Hamid, P. Sebastian e Y. V. Voon. "Resource minimization in a real-time depth-map processing system on FPGA". Em: *IEEE Region 10 Annual International Conference, Proceedings/TENCON* (2011), pp. 706–710. DOI: 10.1109/TENCON.2011.6129200.

[18] C. Gentsos, C. L. Sotiropoulou, S. Nikolaidis e N. Vassiliadis. "Real-time canny edge detection parallel implementation for FPGAs". Em: *2010 IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2010 - Proceedings* (2010), pp. 499–502. DOI: 10.1109/ICECS.2010.5724558.

[19] I. Kuon, R. Tessier e J. Rose. "FPGA architecture: Survey and challenges". Em: *Foundations and Trends in Electronic Design Automation* 2.2 (2007), pp. 135–253. ISSN: 15513939. DOI: 10.1561/1000000005.

[20] T. El-Ghazawi, A. George, I. Gonzalez, H. Lam, S. Merchant, P. Saha, M. Smith, G. Stitt, N. Alam e E. El-Araby. "Exploration of a Research Roadmap for Application Development and Execution on Field-Programmable Gate Array (FPGA)-Based Systems". Em: (out. de 2008), p. 45.

[21] M. C. Herbordt, Y. Gu, T. VanCourt, J. Model, B. Sukhwani e M. Chiu. "Computing models for FPGA-based accelerators". Em: *Computing in Science and Engineering* 10.6 (2008), pp. 35–45. ISSN: 15219615. DOI: 10.1109/MCSE.2008.143.

[22] P. Garcia, D. Bhowmik, R. Stewart, G. Michaelson e A. Wallace. "Optimized memory allocation and power minimization for FPGA-based image processing". Em: *Journal of Imaging* 5.1 (2019). ISSN: 2313433X. DOI: 10.3390/jimaging5010007.

[23] S. Jin, D. Kim, T. T. Nguyen, D. Kim, M. Kim e J. W. Jeon. "Design and implementation of a pipelined datapath for high-speed face detection using FPGA". Em: *IEEE Transactions on Industrial Informatics* 8.1 (2012), pp. 158–167. ISSN: 15513203. DOI: 10.1109/TII.2011.2173943.

[24] P. Lysaght e J. Dunlop. "Dynamic Reconfiguration of FPGAs". Em: *Selected Papers from the Oxford 1993 International Workshop on Field Programmable Logic and Applications on More FPGAs*. Oxford, United Kingdom: Abingdon EE & CS Books, 1994, 82–94. ISBN: 0951845314.

[25] K. Vipin e S. A. Fahmy. "FPGA Dynamic and Partial Reconfiguration". Em: *ACM Computing Surveys* 51.4 (2018), pp. 1–39. ISSN: 03600300. DOI: 10.1145/3193827.

[26] 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.

[27] S. Mittal, S. Gupta e S. Dasgupta. "FPGA: An Efficient And Promising Platform For Real-Time Image Processing Applications". Em: *National Conference On Research and Development In Hardware Systems (CSI-RDHS)* (2008).

[28] T. Saegusa, T. Maruyama e Y. Yamaguchi. "How fast is an FPGA in image processing ?" Em: *Proceedings - 2008 International Conference on Field Programmable Logic and Applications, FPL* (2008), pp. 77–82. DOI: 10.1109/FPL.2008.4629911.

[29] B. Cope, P. Y. Cheung, W. Luk e S. Witt. "Have GPUs made FPGAs redundant in the field of Video Processing?" Em: *Proceedings - 2005 IEEE International Conference on Field Programmable Technology* 2005 (2005), pp. 111–118. DOI: 10.1109/FPT.2005.1568533.

[30] P. Cooke, J. Fowers, G. Brown e G. Stitt. "A tradeoff analysis of FPGAs, GPUs, and multicores for sliding-window applications". Em: *ACM Transactions on Reconfigurable Technology and Systems* 8.1 (2015), pp. 1–24. ISSN: 19367414. DOI: 10.1145/2659000.

[31] M. Qasaimeh, J. Zambreno, P. H. Jones, K. Denolf, J. Lo e K. Vissers. "Analyzing the energy-efficiency of vision kernels on embedded cpu, GPU and FPGA platforms". Em: *Proceedings - 27th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019* (2019), p. 336. DOI: 10.1109/FCCM.2019.00077.

[32] J. Fritsch, T. Kuhnl e A. Geiger. "A new performance measure and evaluation benchmark for road detection algorithms". Em: *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC* Itsc (2013), pp. 1693–1700. DOI: 10.1109/ITSC.2013.6728473.

[33] M. Beyeler, F. Mirus e A. Verl. "Vision-based robust road lane detection in urban environments". Em: *Proceedings - IEEE International Conference on Robotics and Automation* (2014), pp. 4920–4925. ISSN: 10504729. DOI: 10.1109/ICRA.2014.6907580.

[34] B. He, R. Ai, Y. Yan e X. Lang. "Lane marking detection based on convolution neural network from point clouds". Em: *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC* (2016), pp. 2475–2480. DOI: 10.1109/ITSC.2016.7795954.

[35] M. Bertozzi e A. Broggi. "GOLD: A parallel real-time stereo vision system for generic obstacle and lane detection". Em: *IEEE Transactions on Image Processing* 7.1 (1998), pp. 62–81. ISSN: 10577149. DOI: 10.1109/83.650851.

[36] J. Douret, R. Labayrade, J. Laneurit e R. Chapuis. "A reliable and robust lane detection system based on the parallel use of three algorithms for driving safety assistance". Em: *Proceedings of the 9th IAPR Conference on Machine Vision Applications, MVA 2005* (2005), pp. 398–401.

[37] M. Haloi e D. B. Jayagopi. "A robust lane detection and departure warning system". Em: *IEEE Intelligent Vehicles Symposium, Proceedings* 2015-August.Iv (2015), pp. 126–131. DOI: 10.1109/IVS.2015.7225674.

[38] A. López, J. Serrat, C. Cañero, F. Lumbreras e T. Graf. "Robust lane markings detection and road geometry computation". Em: *International Journal of Automotive Technology* 11.3 (2010), pp. 395–407. ISSN: 12299138. DOI: 10.1007/s12239-010-0049-6.

[39] C. R. Jung e C. R. Kelber. "A robust linear-parabolic model for lane following". Em: *Brazilian Symposium of Computer Graphic and Image Processing* (2004), pp. 72–79. ISSN: 15301834. DOI: 10.1109/SIBGRA.2004.1352945.

[40] Y. Wang, E. K. Teoh e D. Shen. "Lane detection and tracking using B-Snake". Em: *Image and Vision Computing* 22.4 (2004), pp. 269–280. ISSN: 02628856. DOI: 10.1016/j.imavis.2003.10.003.

[41] A. S. Huang, D. Moore, M. Antone, E. Olson e S. Teller. "Finding multiple lanes in urban road networks with vision and lidar". Em: *Autonomous Robots* 26.2-3 (2009), pp. 103–122. ISSN: 09295593. DOI: 10.1007/s10514-009-9113-3.

[42] M. Aly. "Real time detection of lane markers in urban streets". Em: *2008 IEEE Intelligent Vehicles Symposium*. 2008, pp. 7–12. DOI: 10.1109/IVS.2008.4621152.

[43] A. Gurghian, T. Koduri, S. V. Bailur, K. J. Carey e V. N. Murali. "DeepLanes: End-To-End Lane Position Estimation Using Deep Neural Networks". Em: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops* (2016), pp. 38–45. ISSN: 21607516. DOI: 10.1109/CVPRW.2016.12.

[44] J. Niu, J. Lu, M. Xu, P. Lv e X. Zhao. "Robust Lane Detection using Two-stage Feature Extraction with Curve Fitting". Em: *Pattern Recognition* 59 (2016), pp. 225–233. ISSN: 00313203. DOI: 10.1016/j.patcog.2015.12.010. URL: http://dx.doi.org/10.1016/j.patcog.2015.12.010.

[45] J. Li, X. Mei, D. Prokhorov e D. Tao. "Deep Neural Network for Structural Prediction and Lane Detection in Traffic Scene". Em: *IEEE Transactions on Neural Networks and Learning Systems* 28.3 (2017), pp. 690–703. ISSN: 21622388. DOI: 10.1109/TNNLS.2016.2522428.

[46] Q. Li, L. Chen, M. Li, S. L. Shaw e A. Nüchter. "A sensor-fusion drivable-region and lane-detection system for Li, Q., Chen, L., Li, M., Shaw, S. L., & Nüchter, A. (2014). A sensor-fusion drivable-region and lane-detection system for autonomous vehicle navigation in challenging road scenarios. IEEE Tran". Em: *IEEE Transactions on Vehicular Technology* 63.2 (2014), pp. 540–555. ISSN: 00189545.

[47] S. Shin, I. Shim e I. S. Kweon. "Combinatorial approach for lane detection using image and LIDAR reflectance". Em: *2015 12th International Conference on Ubiquitous Robots and Ambient Intelligence, URAI 2015* Urai (2015), pp. 485–487. DOI: 10.1109/URAI.2015.7358810.

[48] C. Rose, J. Britt, J. Allen e D. Bevly. "An integrated vehicle navigation system utilizing lane-detection and lateral position estimation systems in difficult environments for GPS". Em: *IEEE Transactions on Intelligent Transportation Systems* 15.6 (2014), pp. 2615–2629. ISSN: 15249050. DOI: 10.1109/TITS.2014.2321108.

[49] T. Y. K. LEE e S. H. "Combustion and Emission Characteristics of Wood Pyrolysis Oil-Butanol Blended Fuels in a Di Diesel Engine". Em: *International Journal of ...* 13.2 (2012), pp. 293–300. ISSN: 1229-9138. DOI: 10.1007/s12239. URL: http://link.springer.com/article/10.1007/s12239-012-0027-2.

[50] S. Sivaraman e M. M. Trivedi. "Integrated lane and vehicle detection, localization, and tracking: A synergistic approach". Em: *IEEE Transactions on Intelligent Transportation Systems* 14.2 (2013), pp. 906–917. ISSN: 15249050. DOI: 10.1109/TITS.2013.2246835.

[51] D. Vikram Mutneja. "Methods of Image Edge Detection: A Review". Em: *Journal of Electrical & Electronic Systems* 04.02 (2015). DOI: 10.4172/2332-0796.1000150.

[52] I. D. Craig. *Undergraduate Topics in Computer Science*. 2009. ISBN: 9781846287732. URL: http://books.google.com/books?hl=en{\&}lr={\&}id=2LIMMD9FVXkC{\&}oi=fnd{\&}pg=PR5{\&}dq=Principles+of+Digital+Image+Processing+fundamental+techniques{\&}ots=HjrHeuS{_}Ei{\&}sig=AJTy37mBFfUgcndo5kCTFLkfBKc.

[53] R. C. Gonzalez e R. E. Woods. *Digital image processing*. Upper Saddle River, N.J.: Prentice Hall, 2008. ISBN: 9780131687288 013168728X 9780135052679 013505267X. URL: http://www.amazon.com/Digital-Image-Processing-3rd-Edition/dp/013168728X.

[54] W. Burger e M. Burge. *Principles of Digital Image Processing: Advanced Methods*. 2013, pp. 1–3. ISBN: 978-1-84882-918-3. URL: http://link.springer.com.proxy.cc.uic.edu/book/10.1007{\%}2F978-1-84882-919-0{\%}5Cnhttp://dx.doi.org/10.1007/978-1-84882-919-0{_}1.

[55] J. E. Bresenham. "Algorithm for computer control of a digital plotter". Em: *IBM Systems Journal* 4.1 (1965), pp. 25–30. ISSN: 0018-8670. DOI: 10.1147/sj.41.0025. URL: http://ieeexplore.ieee.org/document/5388473/.

[56] A. Zingl. "A Rasterizing Algorithm for Drawing Curves". Em: 2012.

[57] Digilent. "Zybo Z7 Board Reference Manual". Em: *Digilent, Inc.* (2017). URL: https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/reference-manual.

[58] Xilinx Inc. "Zynq-7000 SoC Data Sheet: Overview". Em: (2018). URL: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[59] *Pcam 5C Reference Manual*. Digilent, Inc. URL: https://reference.digilentinc.com/reference/add-ons/pcam-5c/reference-manual.

[60] *OV5640 datasheet*. OmniVision Technologies, Inc. 2011. URL: https://cdn.sparkfun.com/datasheets/Sensors/LightImaging/OV5640_datasheet.pdf.

[61] Xilinx Inc. "Vivado Design Suite User Guide High-Level Synthesis". Em: (2018). URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf.

[62] Xilinx Inc. "Vivado HLS Optimization Methodology Guide". Em: (2017). URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1270-vivado-hls-opt-methodology-guide.pdf.

[63] D. G. Bailey. "The Advantages and Limitations of High Level Synthesis for FPGA Based Image Processing". Em: *Proceedings of the 9th International Conference on Distributed Smart Cameras*. ICDSC '15. Seville, Spain: Association for Computing Machinery, 2015, 134–139. ISBN: 9781450336819. DOI: 10.1145/2789116.2789145. URL: https://doi.org/10.1145/2789116.2789145.

[64] R. Brinkmann. *The Art and Science of Digital Compositing, Second Edition: Techniques for Visual Effects, Animation and Motion Graphics (The Morgan Kaufmann Series in Computer Graphics) | Ron Brinkmann | digital library BookOS*. 2008.

[65] J. Xiao, S. Li e B. Sun. "A Real-Time System for Lane Detection Based on FPGA and DSP". Em: *Sensing and Imaging* 17.1 (2016), pp. 1–13. ISSN: 15572072. DOI: 10.1007/s11220-016-0133-8.

[66] Q. Lin, H. Youngjoon e H. Hahn. "Real-time lane detection based on extended edge-linking algorithm". Em: *2nd International Conference on Computer Research and Development, ICCRD 2010* (2010), pp. 725–730. DOI: 10.1109/ICCRD.2010.166.

[67] J. Wu, J. Sun e W. Liu. "Design and implementation of video image edge detection system based on FPGA". Em: *Proceedings - 2010 3rd International Congress on Image and Signal Processing, CISP 2010* 1.1 (2010), pp. 472–476. DOI: 10.1109/CISP.2010.5647070.

[68] *Zybo Z7 Pcam 5C Demo (versão 2017.4)*. Digilent, Inc. URL: https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-z7-pcam-5c-demo/start.

[69] T. Kappenman. *Embedded Vision Demo*. Digilent, Inc. 2019. URL: https://s3-us-west-2.amazonaws.com/digilent-file-share-public/Zybo_Z7_Embedded_Vision_Demo.zip.

[70] X. Wang, C. Kiwus, C. Wu, B. Hu, K. Huang e A. Knoll. "Implementing and Parallelizing Real-time Lane Detection on Heterogeneous Platforms". Em: *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors* 2018-July (2018), pp. 1–8. ISSN: 10636862. DOI: 10.1109/ASAP.2018.8445110.

[71] S. Kim, R. Woo, E. J. Yang e D. W. Seo. "Real time multi-lane detection using relevant lines based on line labeling method". Em: *4th International Conference on Intelligent Transportation Engineering, ICITE 2019* (2019), pp. 301–305. DOI: 10.1109/ICITE.2019.8880189.

ANEXO I

Implementações de algoritmos em Vivado HLS

Listagem I.1: Implementação do método de Otsu.

```
int otsu_threshold(gray_hd &src, int HPT, int HPB, int th)
{
  int const rows = H_HD;
  int const cols = W_HD;

  gray_pixel pixel;

  int x, y;
  int ht = rows*HPT / 100;
  int hb = rows*(100-HPB) / 100;

  // Image histogram
  int histogram_gray[256];

  // reset counters
  otsu_threshold_label0:for (x = 0; x < 256; x++)
  {
    histogram_gray[x] = 0;
  }
  int total_pixels = 0;
  // create image histogram
  otsu_threshold_label1:for (y = 0; y < rows; y++)
  {
    otsu_threshold_label2:for (x = 0; x < cols; x++)
    {
      src >> pixel;
      if (y > ht && y < hb && pixel.val[0] >= th)
```

```
28        {
29          histogram_gray[pixel.val[0]]++;
30          total_pixels++;
31        }
32      }
33    }
34
35    // Otsu algorithm
36    float sum = 0;
37    otsu_threshold_label3:for (int i = 0; i < 256; i++)
38    {
39      sum += histogram_gray[i] * i;
40    }
41
42    float sumB = 0;
43    int wB = histogram_gray[0];
44    int wF;
45
46    float varMax = 0;
47    int threshold = 0;
48
49    otsu_threshold_label4:for (int t = 1; t < 255; t++)
50    {
51      wB += histogram_gray[t];      // Weight Background
52      if (wB == 0) { continue; }
53
54      wF = total_pixels - wB;         // Weight Foreground
55      if (wF == 0) { break; }
56
57      sumB += (float)(t * histogram_gray[t]);
58
59      float mB = sumB / wB;           // Mean Background
60      float mF = (sum - sumB) / wF;   // Mean Foreground
61
62      // Calculate between Class Variance
63      float varBetween = (float)wB * (float)wF * (mB - mF) * (mB - mF);
64
65      // Check if new maximum found
66      if (varBetween > varMax)
67      {
68        varMax = varBetween;
69        threshold = t;
70      }
71    }
72    return threshold;
73  }
74  void otsu_binarization(gray_hd &src, gray_hd &dst, int HPT, int HPB)
75  {
76    int const rows = H_HD;
77    int const cols = W_HD;
```

```
78
79    gray_hd src1(rows,cols), src2(rows,cols), src3(rows,cols), src_tmp(rows, cols);
80    hls::Duplicate(src, src_tmp, src1);
81    hls::Duplicate(src_tmp, src2, src3);
82
83    // Search Otsu threshold value
84    int t1 = otsu_threshold(src1, HPT, HPB, 0);
85    int t2 = otsu_threshold(src2, HPT, HPB, t1/OTSU_DIV);
86
87    // Image segmentation
88    hls::Threshold(src3, dst, t2, FOREGROUND, HLS_THRESH_BINARY);
89  }
```

Listagem I.2: Implementação da Região de interesse.

```
1  void roi_mask(gray_hd &src, gray_hd &dst, int HPT, int HPB)
2  {
3    int const rows = H_HD;
4    int const cols = W_HD;
5
6    gray_pixel pixel;
7    int const ht = rows*HPT / 100;
8    int const hb = rows*(100-HPB) / 100;
9
10   roi_mask_label0:for (int y = 0; y < rows; y++)
11   {
12     roi_mask_label1:for (int x = 0; x < cols; x++)
13     {
14       // Streaming in
15       src >> pixel;
16
17       // Remove top and bottom
18       if (y < ht || y > hb) { pixel.val[0] = BACKGROUND; }
19
20       // Streaming out
21       dst << pixel;
22     }
23   }
24 }
```

Listagem I.3: Implementação do algoritmo adaptado de Bresenham.

```
1  void plot_line(gray_hd &dst, int x0, int y0, int x1, int y1)
2  {
3    int const rows = H_HD;
4    int const cols = W_HD;
5
6    gray_pixel pixel;
7
8    int dx = hls::abs(x1-x0),   sx = x0 < x1 ? 1 : -1;
9    int dy = -hls::abs(y1-y0),  sy = y0 < y1 ? 1 : -1;
```

```
  int err = dx + dy, e2;

  int i = x0, j = y0;
  int done = 0;
  int next_x, next_y;

  plot_line_label0:for (int y = 0; y < rows; y++)
  {
    plot_line_label1:for (int x = 0; x < cols; x++)
    {
      // default
      pixel = BACKGROUND;

      // pixel part of the line
      if (x == i && y == j && done == 0)
      {
        pixel.val[0] = FOREGROUND;

        // plotting is finished
        if (i == x1 && j == y1) { done = 1; }

        // next line pixel
        next_x = 0; next_y = 0;
        e2 = 2 * err;
        if (e2 >= dy) { err += dy; i += sx; next_x = 1; } // e_xy+e_x > 0
        if (e2 <= dx) { err += dx; j += sy; next_y = 1; } // e_xy+e_y < 0

        // error, x decreases and y doesn't change
        if (next_x == 1 && sx == -1 && next_y == 0)
        {
          while (true)
          {
            next_x = 0; next_y = 0;
            e2 = 2 * err;
            if (e2 >= dy) { err += dy; i += sx; next_x = 1; } // e_xy+e_x > 0
            if (e2 <= dx) { err += dx; j += sy; next_y = 1; } // e_xy+e_y < 0

            if (next_y == 1) { break; }
          }
        }
      }
      dst << pixel;
    }
  }
}
```

Listagem I.4: Implementação do algoritmo de detecção de linhas.

```
void line_detection(gray_hd &src, gray_hd &dst)
{
```

```
  const int k_val_r1[3][3] = {{-1,-1,-1},   // horizontal lines
                {2,2,2},
                {-1,-1,-1}};
  const int k_val_r2[3][3] = {{-1,2,-1},    // vertical lines
                {-1,2,-1},
                {-1,2,-1}};
  const int k_val_r3[3][3] = {{-1,-1,2},    // oblique
                {-1,2,2},
                {2,-1,-1}};
  const int k_val_r4[3][3] = {{2,-1,-1},    // oblique
                {-1,2,-1},
                {-1,-1,2}};

    hls::Window<3, 3, int>  kernel_r1, kernel_r2,
              kernel_r3, kernel_r4;
    hls::Point_<int>    anchor;

    for (int i = 0; i < 3; i++)
    {
      for (int j = 0; j < 3; j++)
      {
        kernel_r1.val[i][j] = k_val_r1[i][j];
        kernel_r2.val[i][j] = k_val_r1[i][j];
        kernel_r3.val[i][j] = k_val_r1[i][j];
        kernel_r4.val[i][j] = k_val_r1[i][j];
      }
    }

    anchor.x = -1;
    anchor.y = -1;

    gray_hd src_r[4], dst_r[4];
    hls::DuplicateImageN(src, src_r, 4);

    hls::Filter2D(src_r[0], dst_r[0], kernel_r1, anchor);
    hls::Filter2D(src_r[1], dst_r[1], kernel_r2, anchor);
    hls::Filter2D(src_r[2], dst_r[2], kernel_r3, anchor);
    hls::Filter2D(src_r[3], dst_r[3], kernel_r4, anchor);

    gray_hd dst1, dst2, dst3, dst4;

    hls::Max(dst_r[0], dst_r[1], dst1);
    hls::Max(dst_r[2], dst_r[3], dst2);
    hls::Max(dst1, dst2, dst);
}
```

Listagem I.5: Implementação do algoritmo de selecção de linhas.

```
int lines_selection(gray_hd &dst1, gray_hd &dst2, gray_hd &dst3, gray_hd &dst4,
          gray_hd &dst5, float lines[MAX_LINES][2], int HPT)
```

```
{
  int const rows = H_HD;
  int const cols = W_HD;

  int const max_lines = MAX_LINES;
  int const r_min = MIN_RHO;

  float   a, r;

  int const min_mid = W_HD / 2 - W_HD / 5;
  int const max_mid = W_HD / 2 + W_HD / 5;

  bool  inn_ll=false, inn_rl=false, out_ll=false, out_rl=false, mid_l=false,
      first_line=true, last_line=false, new_line=false;

  int const lines_dis = LINES_DIS;
  hls::Point_<int> p1_min, p1_max, p2_min, p2_max;
  hls::Point_<int> p1_old, p2_old, p1_new, p2_new;
  hls::Point_<int> l_new[2], ll_inn[2], rl_inn[2], ll_out[2], rl_out[2], l_mid[2];

  ll_inn[0].x = 0; ll_inn[0].y = 0; ll_inn[1].x = 0; ll_inn[1].y = 0;
  rl_inn[0].x = 0; rl_inn[0].y = 0; rl_inn[1].x = 0; rl_inn[1].y = 0;

  int i, j;
  // search lines
  lines_plotting_label0:for (i = 0; i < max_lines; i++)
  {
    a = lines[i][0];
    r = lines[i][1];

    if ( (a == 0 && r == 0) || i == max_lines-1 ) { last_line=true; }

    if ( last_line || (a > 0 && a < LEFT_MAX_ANGLE) || (a > RIGHT_MIN_ANGLE && a <
    ↪ RIGHT_MAX_ANGLE) )
    {
      polar_to_xy_limits(a, r, HPT, p1_new, p2_new);    // calculates line limits

      if (first_line)
      {
        p1_min = p1_new;
        p1_max = p1_new;

        p2_min = p2_new;
        p2_max = p2_new;

        p1_old = p1_new;
        p2_old = p2_new;

        first_line = false;
      }
```

```
52      else
53      {
54        if (last_line || hls::abs(p1_new.x - p1_min.x) < lines_dis || hls::abs(p1_new.x -
    ↪  p1_max.x) < lines_dis)
55        {
56          if ( p1_new.x < p1_min.x && (a != 0 && r != 0) ) { p1_min.x = p1_new.x; }
57          if ( p1_new.x > p1_max.x && (a != 0 && r != 0) ) { p1_max.x = p1_new.x; }
58          if ( p2_new.x < p2_min.x && (a != 0 && r != 0) ) { p2_min.x = p2_new.x; }
59          if ( p2_new.x > p2_max.x && (a != 0 && r != 0) ) { p2_max.x = p2_new.x; }
60
61          if ( last_line ) { new_line=true; }
62          else { p1_old = p1_new; p2_old = p2_new; }
63        }
64        else { new_line = true; }
65
66        if (new_line)
67        {
68          // calculate line
69          l_new[0].x = (p1_min.x+p1_max.x)/2;
70          l_new[0].y = (p1_min.y+p1_max.y)/2;
71          l_new[1].x = (p2_min.x+p2_max.x)/2;
72          l_new[1].y = (p2_min.y+p2_max.y)/2;
73
74          printf("line\t%d\t%d\t%d\t%d\n", l_new[0].x, l_new[0].y, l_new[1].x, l_new[1].y
    ↪ );
75
76          // reset variables
77          p1_min = p1_new;
78          p1_max = p1_new;
79
80          p2_min = p2_new;
81          p2_max = p2_new;
82
83          p1_old = p1_new;
84          p2_old = p2_new;
85
86
87          // Left lane line
88          if ( l_new[0].x > 0 && l_new[0].x < cols/2 && l_new[1].x >= 0 && l_new[1].x <
    ↪ cols/2 )
89          {
90            if ( !inn_ll )
91            {
92              inn_ll = true;
93              ll_inn[0] = l_new[0];
94              ll_inn[1] = l_new[1];
95
96              printf("left inner line 1\n");
97            }
98            else if ( l_new[0].x > ll_inn[0].x && l_new[1].x > ll_inn[1].x )
```

```
99            {
100             if ( hls::abs(l_new[0].x-ll_inn[0].x) > lines_dis )
101             {
102               out_ll = true;
103               ll_out[0] = ll_inn[0];
104               ll_out[1] = ll_inn[1];
105             }
106
107             ll_inn[0] = l_new[0];
108             ll_inn[1] = l_new[1];
109
110             printf("left inner line +\n");
111           }
112         }
113         // Right lane line
114         else if ( l_new[0].x > cols/2 && l_new[0].x < cols && l_new[1].x > cols/2 &&
    ↪ l_new[1].x <= cols )
115         {
116           if (!inn_rl)
117           {
118             inn_rl = true;
119             rl_inn[0] = l_new[0];
120             rl_inn[1] = l_new[1];
121
122             printf("right inner line 1\n");
123           }
124           else if (l_new[0].x < rl_inn[0].x && l_new[1].x < rl_inn[1].x)
125           {
126             if (hls::abs(l_new[0].x-rl_inn[0].x) > lines_dis)
127             {
128               out_rl = true;
129               rl_out[0] = rl_inn[0];
130               rl_out[1] = rl_inn[1];
131             }
132
133             rl_inn[0] = l_new[0];
134             rl_inn[1] = l_new[1];
135
136             printf("right inner line +\n");
137           }
138         }
139         // Left outter line
140         if (l_new[0].x > 0 && l_new[0].x < cols/2 && l_new[1].x < 0 && (ll_inn[0].x ==
    ↪ 0 || hls::abs(l_new[0].x-ll_inn[0].x) > lines_dis))
141         {
142           if (!out_ll)
143           {
144             out_ll = true;
145             ll_out[0] = l_new[0];
146             ll_out[1] = l_new[1];
```

```
147
148             printf("left outer line 1\n");
149           }
150           else if (l_new[0].x > ll_out[0].x && l_new[1].x > ll_out[1].x)
151           {
152             ll_out[0] = l_new[0];
153             ll_out[1] = l_new[1];
154
155             printf("left outer line +\n");
156           }
157         }
158         // Right outter line
159         if (l_new[0].x > cols/2 && l_new[0].x < cols && l_new[1].x > cols && (rl_inn[0].
    ↪ x == 0 || hls::abs(l_new[0].x-rl_inn[0].x) > lines_dis))
160         {
161           if (!out_rl)
162           {
163             out_rl = true;
164             rl_out[0] = l_new[0];
165             rl_out[1] = l_new[1];
166
167             printf("right outer line 1\n");
168           }
169           else if (l_new[0].x < rl_out[0].x && l_new[1].x < rl_out[1].x)
170           {
171             rl_out[0] = l_new[0];
172             rl_out[1] = l_new[1];
173
174             printf("right outer line +\n");
175           }
176         }
177         // Middle line
178         if (l_new[0].x > min_mid && l_new[0].x < max_mid && l_new[1].x > min_mid &&
    ↪ l_new[1].x < max_mid)
179         {
180           if (!mid_l)
181           {
182             mid_l = true;
183             l_mid[0] = l_new[0];
184             l_mid[1] = l_new[1];
185
186             printf("mid line 1\n");
187           }
188           else if (l_new[0].x > l_mid[0].x)
189           {
190             l_mid[0] = l_new[0];
191             l_mid[1] = l_new[1];
192
193             printf("mid line +\n");
194           }
```

```
195           }
196
197           if (last_line) { break; }
198         }
199       }
200     }
201   }
202   // Left and Right lines
203   if (inn_ll || inn_rl)
204   {
205     if (inn_ll && inn_rl)
206     {
207       line_mask(dst1, ll_inn[0].x, ll_inn[0].y, ll_inn[1].x, ll_inn[1].y);
208       line_mask(dst2, rl_inn[0].x, rl_inn[0].y, rl_inn[1].x, rl_inn[1].y);
209     }
210     else if (inn_ll && !inn_rl)
211     {
212       line_mask(dst1, ll_inn[0].x, ll_inn[0].y, ll_inn[1].x, ll_inn[1].y);
213       line_mask(dst2, ll_inn[0].x, ll_inn[0].y, ll_inn[1].x, ll_inn[1].y);
214     }
215     else if (!inn_ll && inn_rl)
216     {
217       line_mask(dst1, rl_inn[0].x, rl_inn[0].y, rl_inn[1].x, rl_inn[1].y);
218       line_mask(dst2, rl_inn[0].x, rl_inn[0].y, rl_inn[1].x, rl_inn[1].y);
219     }
220   }
221   else
222   {
223     hls::Zero(dst1);
224     hls::Zero(dst2);
225   }
226
227   // Left and Right out lines
228   if (out_ll || out_rl)
229   {
230     if (out_ll && out_rl)
231     {
232       line_mask(dst3, ll_out[0].x, ll_out[0].y, ll_out[1].x, ll_out[1].y);
233       line_mask(dst4, rl_out[0].x, rl_out[0].y, rl_out[1].x, rl_out[1].y);
234     }
235     else if (out_ll && !out_rl)
236     {
237       line_mask(dst3, ll_out[0].x, ll_out[0].y, ll_out[1].x, ll_out[1].y);
238       line_mask(dst4, ll_out[0].x, ll_out[0].y, ll_out[1].x, ll_out[1].y);
239     }
240     else if (!out_ll && out_rl)
241     {
242       line_mask(dst3, rl_out[0].x, rl_out[0].y, rl_out[1].x, rl_out[1].y);
243       line_mask(dst4, rl_out[0].x, rl_out[0].y, rl_out[1].x, rl_out[1].y);
244     }
```

```
245   }
246   else
247   {
248     hls::Zero(dst3);
249     hls::Zero(dst4);
250   }
251
252   // Middle line
253   if (mid_l)
254   {
255     line_mask(dst5,  l_mid[0].x, l_mid[0].y, l_mid[1].x, l_mid[1].y);
256   }
257   else
258   {
259     hls::Zero(dst5);
260   }
261 }
```

ANEXO II

ALGORITMO DE INICIALIZAÇÃO DO *hardware* DO FPGA

Listagem II.1: Inicialização do *hardware* e verificação de alterações nos interruptores.

```
#include "xparameters.h"

#include "platform/platform.h"
#include "ov5640/OV5640.h"
#include "ov5640/ScuGicInterruptController.h"
#include "ov5640/PS_GPIO.h"
#include "ov5640/AXI_VDMA.h"
#include "ov5640/PS_IIC.h"
#include "axis_switch/SWITCH_CTL.h"

#include "MIPI_D_PHY_RX.h"
#include "MIPI_CSI_2_RX.h"


#define IRPT_CTL_DEVID    XPAR_PS7_SCUGIC_0_DEVICE_ID
#define GPIO_DEVID      XPAR_PS7_GPIO_0_DEVICE_ID
#define GPIO_IRPT_ID    XPAR_PS7_GPIO_0_INTR
#define CAM_I2C_DEVID   XPAR_PS7_I2C_0_DEVICE_ID
#define CAM_I2C_IRPT_ID   XPAR_PS7_I2C_0_INTR
#define VDMA_DEVID      XPAR_AXIVDMA_0_DEVICE_ID
#define VDMA_MM2S_IRPT_ID XPAR_FABRIC_AXI_VDMA_0_MM2S_INTROUT_INTR
#define VDMA_S2MM_IRPT_ID XPAR_FABRIC_AXI_VDMA_0_S2MM_INTROUT_INTR
#define CAM_I2C_SCLK_RATE 100000
#define SRC_AXIS_SW_DEVID XPAR_AXIS_SWITCH_0_DEVICE_ID
#define DST_AXIS_SW_DEVID XPAR_AXIS_SWITCH_1_DEVICE_ID
#define GPIO_SW_DEVID     XPAR_SW_GPIO_DEVICE_ID

```

```
28 #define DDR_BASE_ADDR   XPAR_DDR_MEM_BASEADDR
29 #define MEM_BASE_ADDR   (DDR_BASE_ADDR + 0x0A000000)
30
31 #define GAMMA_BASE_ADDR     XPAR_AXI_GAMMACORRECTION_0_BASEADDR
32
33 using namespace digilent;
34
35 void pipeline_mode_change(AXI_VDMA<ScuGicInterruptController>& vdma_driver, OV5640& cam,
       ↪ VideoOutput& vid, Resolution res, OV5640_cfg::mode_t mode)
36 {
37   //Bring up input pipeline back-to-front
38   {
39     vdma_driver.resetWrite();
40     MIPI_CSI_2_RX_mWriteReg(XPAR_MIPI_CSI_2_RX_0_S_AXI_LITE_BASEADDR, CR_OFFSET, (
       ↪ CR_RESET_MASK & ~CR_ENABLE_MASK));
41     MIPI_D_PHY_RX_mWriteReg(XPAR_MIPI_D_PHY_RX_0_S_AXI_LITE_BASEADDR, CR_OFFSET, (
       ↪ CR_RESET_MASK & ~CR_ENABLE_MASK));
42     cam.reset();
43   }
44
45   {
46     vdma_driver.configureWrite(timing[static_cast<int>(res)].h_active, timing[static_cast
       ↪ <int>(res)].v_active);
47     Xil_Out32(GAMMA_BASE_ADDR, 3); // Set Gamma correction factor to 1/1.8
48     //TODO CSI-2, D-PHY config here
49     cam.init();
50   }
51
52   {
53     vdma_driver.enableWrite();
54     MIPI_CSI_2_RX_mWriteReg(XPAR_MIPI_CSI_2_RX_0_S_AXI_LITE_BASEADDR, CR_OFFSET,
       ↪ CR_ENABLE_MASK);
55     MIPI_D_PHY_RX_mWriteReg(XPAR_MIPI_D_PHY_RX_0_S_AXI_LITE_BASEADDR, CR_OFFSET,
       ↪ CR_ENABLE_MASK);
56     cam.set_mode(mode);
57     cam.set_awb(OV5640_cfg::awb_t::AWB_ADVANCED);
58   }
59
60   //Bring up output pipeline back-to-front
61   {
62     vid.reset();
63     vdma_driver.resetRead();
64   }
65
66   {
67     vid.configure(res);
68     vdma_driver.configureRead(timing[static_cast<int>(res)].h_active, timing[static_cast<
       ↪ int>(res)].v_active);
69   }
70
```

```
  {
    vid.enable();
    vdma_driver.enableRead();
  }
}

int main()
{
  init_platform();

  ScuGicInterruptController irpt_ctl(IRPT_CTL_DEVID);
  PS_GPIO<ScuGicInterruptController> gpio_driver(GPIO_DEVID, irpt_ctl, GPIO_IRPT_ID);
  PS_IIC<ScuGicInterruptController> iic_driver(CAM_I2C_DEVID, irpt_ctl, CAM_I2C_IRPT_ID,
    ↪ 100000);
  SWITCH_GPIO sw_gpio(GPIO_SW_DEVID);
  AXI_SWITCH src_switch(SRC_AXIS_SW_DEVID);
  AXI_SWITCH dst_switch(DST_AXIS_SW_DEVID);

  SWITCH_CTL axis_switch_ctl(src_switch, dst_switch, sw_gpio, XPAR_AXIS_SWITCH_0_NUM_MI,
    ↪ 0);

  OV5640 cam(iic_driver, gpio_driver);
  AXI_VDMA<ScuGicInterruptController> vdma_driver(VDMA_DEVID, MEM_BASE_ADDR, irpt_ctl,
      VDMA_MM2S_IRPT_ID,
      VDMA_S2MM_IRPT_ID);
  VideoOutput vid(XPAR_VTC_0_DEVICE_ID, XPAR_VIDEO_DYNCLK_DEVICE_ID);

  pipeline_mode_change(vdma_driver, cam, vid, Resolution::R1280_720_60_PP, OV5640_cfg::
    ↪ mode_t::MODE_720P_1280_720_60fps);


  xil_printf("Video init done.\r\n");


  // Liquid lens control
  uint8_t read_char0 = 0;
  uint8_t read_char1 = 0;
  uint8_t read_char2 = 0;
  uint8_t read_char4 = 0;
  uint8_t read_char5 = 0;
  uint16_t reg_addr;
  uint8_t reg_value;
  uint32_t color_values;

  while (1) {

    axis_switch_ctl.updateChannel();

  }

```

```
118   cleanup_platform();
119
120   return 0;
121 }
```