Universidade do Minho
Escola de Engenharia

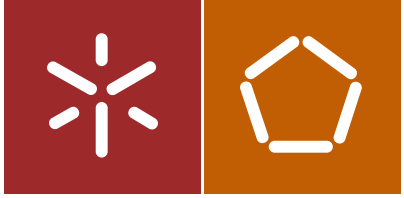
Joaquim Leal Rangel Fonseca

Segmentação de matrícula em CUDA e FPGA

Joaquim Leal Rangel Fonseca Segmentação de matrícula em CUDA e FPGA

UMinho | 2015

novembro de 2015



Universidade do Minho
Escola de Engenharia

Joaquim Leal Rangel Fonseca

Segmentação de matrícula em CUDA e FPGA

Dissertação de Mestrado
Ciclo de Estudos Integrados Conducentes ao Grau de
Mestre em Engenharia Eletrónica Industrial e Computadores

Trabalho efectuado sob a orientação de
Professor Doutor Nuno Filipe Gomes Cardoso
Professor Doutor Nuno Pedro Rodrigues Peixoto

novembro de 2015

DECLARAÇÃO

Nome: Joaquim Leal Rangel Fonseca

Endereço eletrónico: joaquimLrangel@gmail.com Telefone: 915555670

Número do Bilhete de Identidade: 13575013

Título dissertação:

Segmentação de matrícula em CUDA e FPGA

Orientadores:

Professor Doutor Nuno Filipe Gomes Cardoso

Professor Doutor Nuno Pedro Rodrigues Peixoto

Ano de conclusão: 2015

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Mestrado Integrado em Engenharia Eletrónica Industrial e de Computadores

Nos exemplares das teses de doutoramento ou de mestrado ou de outros trabalhos entregues para prestação de provas públicas nas universidades ou outros estabelecimentos de ensino, e dos quais é obrigatoriamente enviado um exemplar para depósito legal na Biblioteca Nacional e, pelo menos outro para a biblioteca da universidade respetiva, deve constar uma das seguintes declarações:

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, ___/___/_____

Assinatura: _____

Agradecimentos

Em primeiro lugar, gostaria de agradecer aos meus orientadores, o Doutor Nuno Cardoso e o Doutor Nuno Peixoto, pelo apoio, aconselhamento e disponibilidade que me proporcionaram ao longo do ano, que foram, sem dúvida, necessários para a conclusão deste projeto.

Agradeço também ao Doutor Adriano Tavares e ao Doutor Jorge Cabral, pela ajuda que proporcionaram no decorrer do processo de dissertação.

Gostaria também de deixar uma nota de agradecimento muito especial aos meus pais, Joaquim Maria de Araújo Rangel Fonseca e Maria de Fátima Veríssimo Leal Rangel Fonseca, que sempre me deram a possibilidade e apoio necessários para alcançar esta etapa. Infelizmente as circunstâncias da vida nem sempre são as ideais, pelo que dedico este trabalho à minha mãe, que de certeza estará feliz por me ver concluir mais um objetivo.

Por último gostava de agradecer aos meus amigos e família, pela sua ajuda e compreensão e aos meus colegas de curso, pelos momentos passados ao longo do curso.

Resumo

A sociedade enfrenta atualmente problemas de mobilidade rodoviária em ambientes urbanos. Com o aumento verificado da complexidade na gestão das redes de transporte, apareceu a necessidade de sistemas inteligentes de auxílio. A deteção e identificação da matrícula dos veículos apresenta-se como uma das componentes mais importantes, possibilitando tarefas mais complexas como o seguimento de veículos, ou mais simples como o auxílio ao parqueamento em regiões controladas.

Neste documento é proposta uma nova abordagem no processo de segmentação e deteção da região da matrícula, que permitirá o aparecimento de uma nova gama de aplicações da tecnologia. Recorrendo a uma sequência de fotogramas obtidos através de uma câmara de videovigilância e um sistema heterogéneo, composto por CPU e GPU ou FPGA, com migração das rotinas de processamento de imagem mais intensivas para *hardware,* será possível realizar o processamento no local e em tempo real, de uma forma não intrusiva. O sistema é composto por quatro principais módulos de processamento: captura e pré-processamento, segmentação, identificação e seguimento.

O sistema proposto apresenta a capacidade de deteção da região da matrícula de mais do que um veículo, a distâncias variadas, mantendo os requisitos de tempo real. A diminuição do detalhe do *background* e o destaque dos caracteres da matrícula pela metodologia de *Threshold* local, em conjunto com a de reconstrução morfológica, permite a deteção em ambientes mais complexos como o urbano e rural.

Durante a realização da dissertação foi criada uma *framework* em OpenGL, com aceleração das funções de processamento de imagem altamente paralelizáveis em CUDA, que permite uma rápida depuração da metodologia proposta, simplificação do processo de calibração do sistema e a validação dos resultados obtidos.

Finalmente, é proposta uma implementação do algoritmo de reconstrução morfológica em FPGA, uma vez que o módulo é considerado crítico para o bom desempenho do sistema. A comparação dos resultados obtidos com a implementação em CPU demonstra a viabilidade da implementação do sistema em FPGA, especialmente em ambiente embebidos.

Abstract

In the current days, the society faces a road mobility problem in urban environments. With the increase in the complexity in the management of public transportation networks, appeared a necessity for auxiliary intelligent transportation systems. The detection and identification of the vehicle registration is presented as one of the most important of its components, enabling more complex tasks such as vehicle tracking, or simply parking assist in controlled areas.

In this paper a new approach is proposed in the segmentation of the license plate region, which will allow a new range of applications for the technology. Using a sequence of frames taken by a surveillance camera and a heterogeneous computing system, consisting of a CPU and a FPGA or GPU, will be capable of processing the information on site and on real-time, in a non-intrusive way, by migrating the most intensive image processing routines to hardware. The system consists of four main processing modules: frame capture and pre-processing, segmentation, license plate detection and vehicle tracking.

The proposed system has the capability of detecting the license plate region of more than one vehicle per frame, at varying distances, while keeping the real-time requirements. The decreased detail in the background and the highlight of the license plate characters, provided by the Local Threshold in conjunction with the Morphological Reconstruction technique, will enable the detection on more complex environments such as urban or rural ones.

During the course of this dissertation a framework based on OpenGL, with acceleration of highly parallel image processing algorithms CUDA, was created in order to allow for faster and more precise debugging of the application and its results.

Finally, an implementation of the Morphological Reconstruction algorithm in FPGA is proposed, since this module is deemed critical for the system performance. The comparison with the results from the CPU implementation demonstrate the feasibility of the implementation of the system in FPGA, especially in embedded environments.

Conteúdo

Índice de figuras

Índice de tabelas

Índice de equações

Acrónimos e abreviaturas

2D	Duas Dimensões
ANPR	Automatic Number Plate Recognition
ANSI	American National Standards Institute
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BMP	Bitmap image file
Cloud	Cloud computing
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDR3	Double Data Rate type three
DMA	Direct Memory Access
Dpi	Dots per inch
DSP	Digital Signal Processor
E/S	Entrada / Saída
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
FPS	Frames per Second
GPGPU	General Purpose Graphics Processing Unit
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HD	High Definition
HDMI	High-Definition Multimedia Interface
HDL	Hardware Description Language
HLS	High-Level Synthesis
HPS	Hard Processor System
IDE	Integrated Development Environment
ITS	Intelligent Transportation Systems
OCR	Optical Character Recognition
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OS	Operating System
PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect express
RAM	Random-Access Memory
RGB	Red, Green, Blue
ROM	Read-Only Memory
SD	Simple Definition
SM	Streaming Multiprocessor
SoC	System on Chip
VGA	Video Graphics Array

1. Introdução

Automatic Number Plate Recognition (ANPR) é uma metodologia de vigilância em massa que utiliza *Optical Character Recognition* (OCR) para ler a matrícula dos veículos, efetuando assim o seu reconhecimento. Existem, neste momento, duas possíveis metodologias na conceção de sistemas ANPR [1].

A primeira, e mais usual, consiste na utilização de um sensor de imagem direcionado a uma área restrita, onde só atravessa um veículo (como portagens, entradas, saídas, etc.). Nesse espaço está presente um sensor, normalmente de posição, que quando ativado envia os fotogramas capturados pela câmara para um computador central (servidor). As imagens capturadas entram no sistema de espera (*queue*) do servidor, e são processadas logo que possível. Esta abordagem apresenta bastantes limitações, pois os recursos do servidor são partilhados por todos os sensores de imagem da infraestrutura. De modo a não sobrecarregar a *queue* do servidor, é necessário enviar o menor número possível de fotogramas e com baixa resolução, pois o número de operações sobre pixéis cresce com o quadrado das dimensões da imagem numa razão n^2. Estas limitações impossibilitam a expansão da área de aplicação da tecnologia, ficando assim restringidas ao controlo de entrada/saída em parques de estacionamento e portagens.

A segunda metodologia consiste no processamento e identificação no local e em tempo real, porém este é um processo computacionalmente muito intensivo. A arquitetura consiste num sistema embebido, composto por um sensor de imagem e um processador, que deteta e identifica a matrícula do veículo. O avanço da tecnologia permite melhorar o tempo de resposta de sistemas anteriormente propostos ou o aparecimento de novas metodologias com taxas de sucesso superior.

Nesta dissertação aborda-se a segmentação da região da matrícula, em tempo real, com o intuito de proporcionar um seguimento de veículos robusto. O processamento é acelerado em *hardware*, de forma a cumprir os requisitos de tempo real e expandir a gama de aplicações da tecnologia.

1.1. Enquadramento

Um dos maiores problemas relativamente ao seguimento de objetos, encontra-se na complexidade associada ao processo de relacionar e identificar objetos em diferentes fotogramas [2]. Utilizando a matrícula como identificador comum entre veículos, torna-se possível ultrapassar esta limitação. O seguimento de veículos em tempo real é uma tarefa computacionalmente muito intensiva. Atualmente, a maioria dos sistemas ANPR utiliza *general purpose* CPUs para resolver algoritmos de processamento de imagem complexos e intensivos [3].

Com o aumento da resolução na aquisição de imagens, o processador, mesmo que possua vários núcleos, deixa de ser capaz de responder aos requisitos de tempo real do sistema, pelo que se torna necessária a aceleração por *hardware* (GPU, DSP, FPGA, etc.). O aumento da qualidade de imagem, apesar de requerer maior capacidade de processamento no processo de localização da matrícula, é essencial para o aumento da taxa de sucesso do *Optical Character Recognition* (OCR), na casa dos 300 dpi [4].

Neste momento, uma das aplicações mais importantes da metodologia é a sua integração num sistema inteligente de transportação, uma área em constante desenvolvimento que integra diversas tecnologias. O serviço procura fornecer informação aos diversos meios de transporte, de forma a melhorar a gestão de tráfego e permitir ao utilizador tomar melhores decisões sobre como utilizar os sistemas disponíveis. Como os sistemas ANPR permitem o reconhecimento da matrícula, sem posicionar nenhum componente diretamente na estrada ou na berma, são considerados sistemas de deteção de trânsito não-intrusivos.

1.2. Motivação

A sociedade sofre atualmente de problemas de mobilidade rodoviária introduzidos pela vida citadina e pelas urbanizações, que causam problemas económicos e sociais [5]. O aumento da complexidade da gestão das redes de transporte, que se vem sentindo há mais de uma década, proporcionou o aparecimento de novos sistemas de auxílio na sua gestão. O desenvolvimento de novos sistemas inteligentes de decisão e controlo é visto como uma necessidade no controlo de

serviços públicos de transporte, no apoio a autoridades e no melhoramento da qualidade de vida dos cidadãos em geral [5].

Assim, os *Intelligent Transportation Systems* (ITS) foram introduzidos de forma a tirar partido das infraestruturas já disponíveis, melhorando a sua eficiência e atratividade. São definidos como um conjunto de sistemas avançados de comunicação, processamento e controlo com o intuito de melhorar a qualidade dos sistemas de transporte, salvando vidas (através de sinalização adequada), poupando tempo e dinheiro [6].

Os sistemas de ANPR são parte integrante dos ITS, uma vez que apresentam uma forma de deteção de veículos não intrusiva, através da análise de um *stream* de vídeo proveniente de uma câmara de videovigilância. Com o processamento dos dados a ser realizado no local, a gama de aplicações da metodologia é extensa e apenas é limitada pela qualidade do algoritmo de deteção ou capacidade de processamento da tecnologia.

A junção do seguimento de veículos com um sistema ANPR proporcionará o aparecimento de novas aplicações da tecnologia, como a deteção de veículos em contra mão. Um dos objetivos atuais da engenharia rodoviária é diminuir o número de veículos que circulam em sentido contrário, em especial nas autoestradas e vias-rápidas, onde a velocidade elevada a que os veículos circulam acaba por provocar acidentes fatais, causados maioritariamente por choques frontais [7]. Nos Estados Unidos morrem por ano, em média, 350 pessoas em acidentes causados por veículos que circulam em contramão nas autoestradas [7].

Outra possível aplicação do sistema será a deteção de acidentes em autoestradas, através de uma rede de câmaras conectadas entre si e estrategicamente posicionadas, será possível prever se houve um acidente ou avaria num troço. Utilizando um mecanismo de *timeout* que regista a última posição conhecida e inicia uma contagem decrescente. Quando o veículo é detetado na próxima etapa, o contador recomeça e o processo continua até acabar o percurso. No entanto, se o contador terminar sem que o veículo volte a ser identificado, pode ser emitido um alerta contendo a última posição conhecida do veículo.

A interação do sistema com um serviço de *cloud* permite uma fácil integração numa rede de ITS, fornecendo dados em tempo real, aliviando o problema da complexidade de gestão dos sistemas de transporte atuais. Um sistema ativo de localização poderá ser criado, dependendo da taxa de sucesso do módulo de deteção e do posicionamento estratégico de diversos módulos.

1.3. Desafios

O maior desafio deste trabalho de investigação está em conciliar o tempo de resposta do sistema com o aumento de resolução das imagens adquiridas, proporcionando assim, melhores resultados de OCR, tornando o sistema mais robusto e fiável.

Para que seja possível criar um sistema de seguimento robusto, é necessário expandir a área em que a deteção da matrícula é efetuada. Para que tal seja possível, a implementação de um algoritmo de segmentação robusto, capaz de segmentar a matrícula dos veículos (*foreground*) do ambiente que os rodeia (*background*), é considerado um grande desafio ao sucesso da metodologia. Para além de ser obrigatório ter em consideração o pavimento, vegetação ou ambiente que irá rodear os veículos, a matrícula apresentará dimensões/proporções significativamente inferiores em relação ao resto da imagem, contrariamente às técnicas normalmente propostas na literatura [8].

O interface com a memória é uma limitação que se deve ter em conta, durante todas as fases de desenvolvimento, de forma a não se tornar um *bottleneck* do sistema. Em arquiteturas SoC, onde o CPU e o GPU partilham o mesmo circuito integrado, a memória (RAM) é unificada e partilhada por ambos, o que pode causar alguma lentidão de acesso em aplicações intensivas do ponto de vista de acesso à memória. Quando o sistema apresenta uma GPU discreta, a latência do barramento *PCIExpress*, que realiza a transferência de dados entre o CPU e o GPU, deve ser analisada para que se possa aferir a viabilidade da aceleração.

Finalmente, a comunicação entre o CPU e o FPGA é um processo delicado, pois o sistema operativo utilizado não é de tempo real (*Linux* embebido), o que pode levar a algumas falhas no processo de OCR.

1.4. Objetivos

Esta dissertação tem como objetivo o levantamento do estado da arte, idealização, especificação, desenvolvimento e implementação de um sistema de seguimento de veículos motorizados, através da identificação e deteção da posição da sua matrícula, em tempo real. A arquitetura do sistema é composta por quatro etapas principais de processamento: captura e pré-processamento da imagem, segmentação, identificação e seguimento.

Devido aos enormes recursos computacionais exigidos, é realizada a migração para *hardware* de etapas de processamento de imagem altamente paralelizáveis, de forma a responder aos requisitos de tempo real.

A memória assume um papel fundamental no sistema, devido à necessidade de armazenamento de fotogramas antes do processamento. É assim necessário identificar qual o impacto da memória no desempenho do sistema e explorar o paralelismo da arquitetura no acesso à memória.

O sistema apresenta duas implementações distintas: (i) analisar uma sequência de fotogramas (sem compressão), segmentar e identificar as regiões da matrícula, acelerando o processo recorrendo à *framework* CUDA, migrando as funções altamente paralelas do CPU para o GPU e (ii) implementar um sistema embebido, composto por um SoC (FPGA e CPU) e um sensor de imagem. Pretende-se que esta abordagem estude a viabilidade da implementação do algoritmo de segmentação num sistema embebido.

1.4.1. Requisitos funcionais

- Adquirir e processar imagens de elevada resolução;
- Segmentar e identificar possíveis regiões da matrícula;
- Efetuar o seguimento de veículos.

1.4.2. Requisitos não funcionais

- Ser capaz de processar pelo menos 20 fotogramas por segundo;
- Necessitar de pouca calibração no algoritmo;
- Garantir que a qualidade da imagem original é mantida (OCR);
- Ter capacidade de comunicação com uma plataforma de *cloud*;
- Ser capaz de identificar mais do que uma matrícula em simultâneo;
- Ser possível adaptar a metodologia a um sistema embebido.

1.5. Contributos

Durante as etapas efetuadas ao longo do trabalho foi gerado novo conhecimento em diversas áreas, nomeadamente no processamento de imagens e aceleração em GPU. Uma nova abordagem na conceção de sistemas ANPR é proposta, com foco no módulo de pré-processamento e segmentação.

O módulo de pré-processamento apresenta uma nova abordagem, que procura manter a definição da imagem original perante o módulo de reconhecimento e aumentar o desempenho do processo de segmentação. É proposta uma variação do algoritmo de *Threshold* local, com o intuíto de realçar o contraste entre o *foreground* e *background* da região da matrícula. Por fim, o algoritmo de reconstrução morfológica, similar em alguns aspetos às técnicas de *labeling* utilizadas na literatura, é introduzido nos sistemas de ANPR. Assim, para a implementação do sistema é proposta a utilização da *framework* CUDA.

1.6. Estrutura analítica do projeto

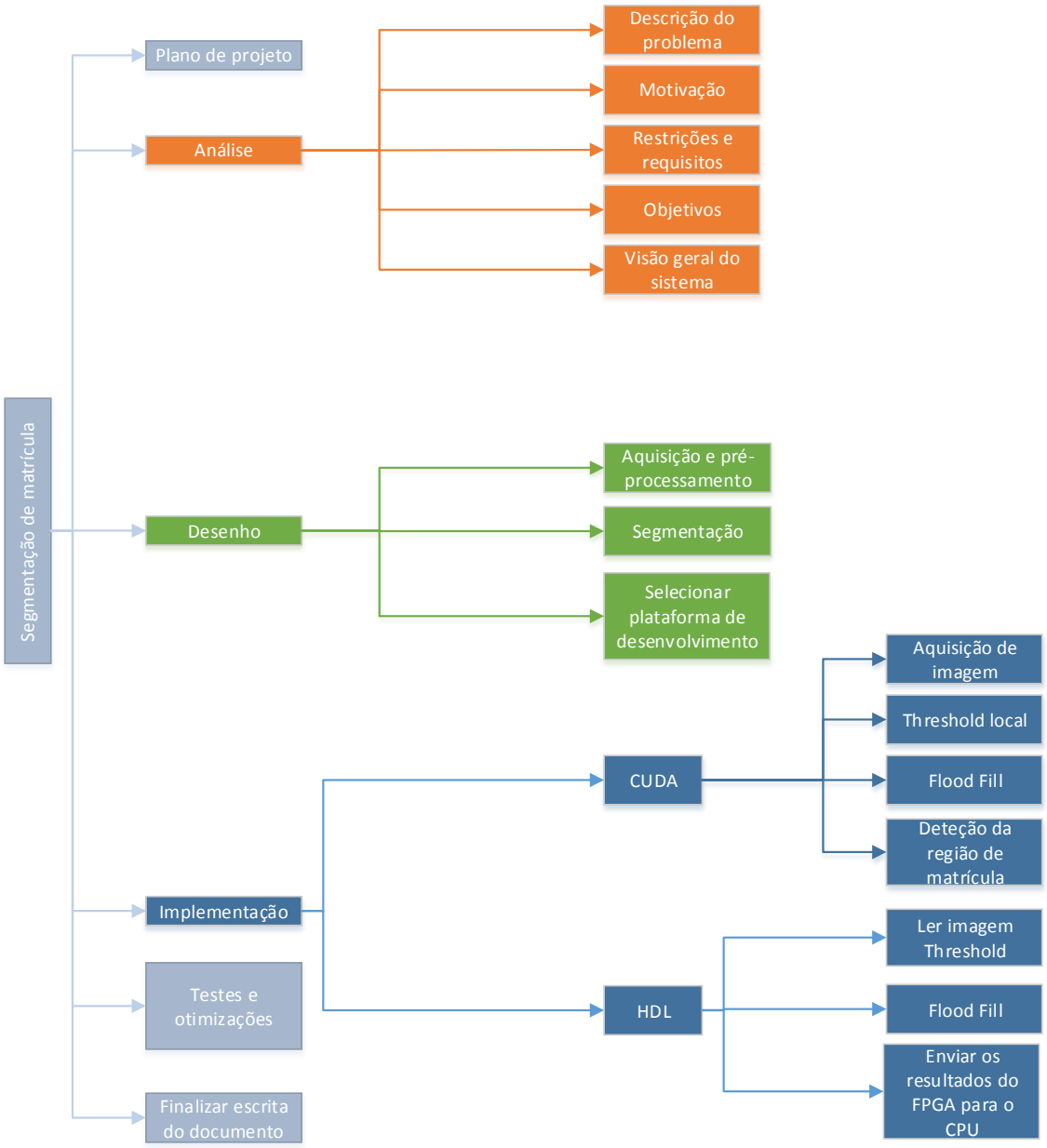


Figura 1.6.1 – Estrutura analítica do projeto

1.7. Estrutura analítica dos recursos

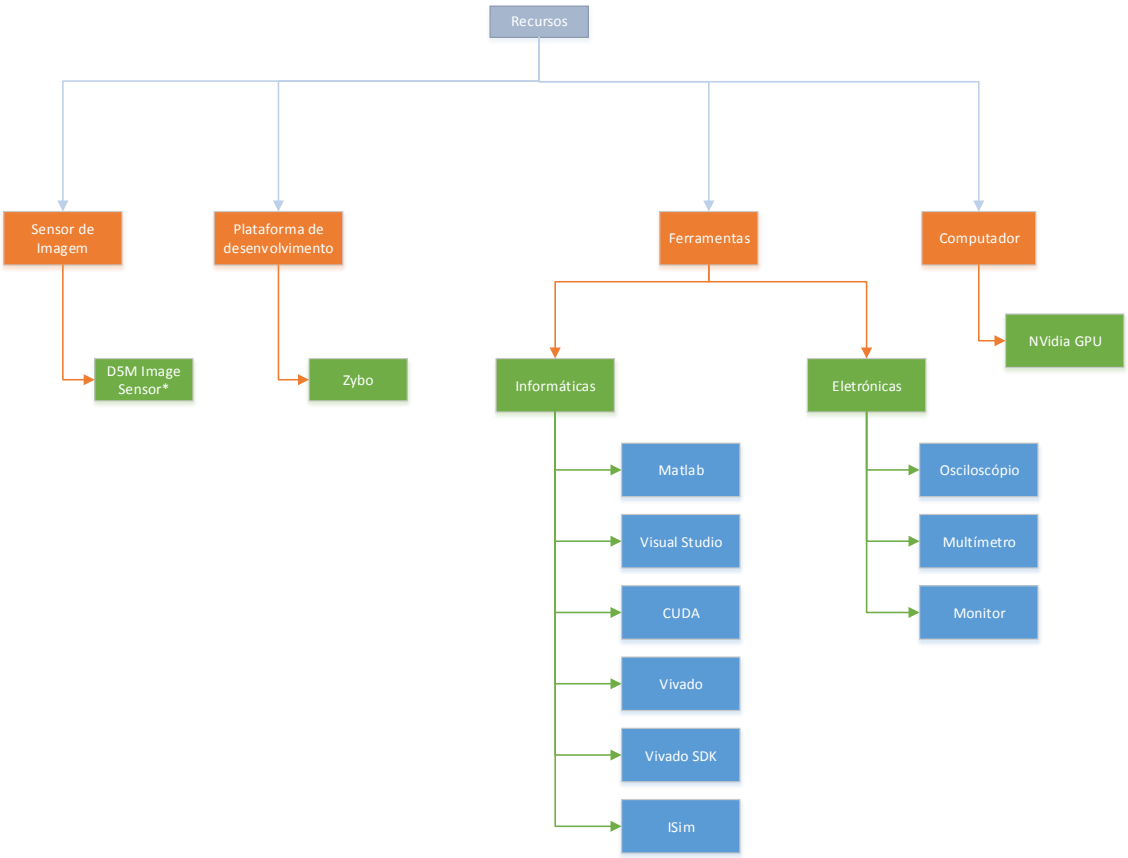


Figura 1.7.1 – Estrutura analítica dos recursos

1.8. Riscos

1.8.1. Estrutura analítica dos riscos

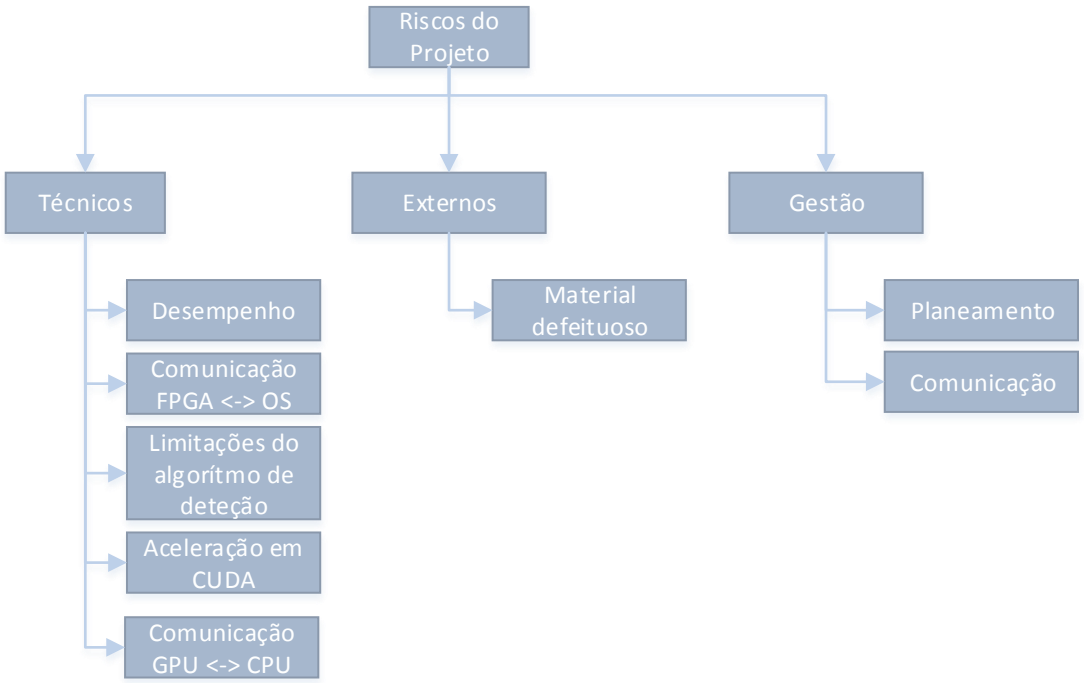


Figura 1.8.1.1 – Estrutura analítica dos riscos

1.8.2. Riscos técnicos

- Não atingir uma velocidade de processamento adequada no seguimento de veículos a velocidades elevadas;
- Baixa qualidade na aquisição de imagens, que pode levar à troca de caracteres pelo módulo de OCR;
- Não se adaptar corretamente às alterações climatéricas (como mudanças de luminosidade, precipitação, nevoeiro, etc.).
- Não ser capaz de segmentar a matrícula do *background* em ambientes complexos, como rurais (vegetação, pavimento) ou urbanos (pavimento, sinalização, etc.).

1.9. Âmbito e limitações

Um dos objetivos deste trabalho de dissertação é a conceção de um novo algoritmo de segmentação da região da matrícula. Para que tal seja possível o algoritmo deve ser trabalhado, numa fase inicial, em alto nível recorrendo à ferramenta matlab, onde estão disponíveis os resultados numéricos e visuais para comparação. O foco está na análise de imagens estáticas em tons de cinzento (*grayscale*), com o objetivo de segmentar as possíveis regiões da matrícula.

O passo seguinte é a implementação do algoritmo em C/C++ com aceleração em GPU através da *framework* CUDA. O recurso ao OpenGL para o *rendering* das várias imagens produzidas pelo algoritmo é importante para a depuração e confirmação de resultados, acelerando o processo de desenvolvimento.

Devido às restrições impostas de tempo (duração do período de dissertação) e do espaço disponível na placa de desenvolvimento, a implementação em FPGA incide sobre o módulo considerado mais crítico após o *profiling* da aplicação.

1.10. Organização da dissertação

Este documento encontra-se dividido em cinco capítulos que abordam a análise, o desenvolvimento e as diferentes implementações do sistema proposto. Assim, o capítulo 2 começa por levantar o estado da arte das técnicas de segmentação que podem ser aplicadas na deteção da região da matrícula. Posteriormente, são analisados os sistemas ANPR propostos na literatura e as suas limitações. O terceiro capítulo propõe uma nova abordagem na segmentação da região da matrícula, com o intuito de aplicar a metodologia a um sistema de seguimento em tempo real. O quarto capítulo descreve a implementação do sistema proposto em C/C++, a respetiva aceleração em CUDA e do *rendering* das imagens processadas recorrendo à *framework OpenGL,* com o intuito de depurar e assim melhorar os resultados obtidos. O quinto capítulo analisa a viabilidade e propõe uma implementação do algoritmo de reconstrução morfológica em FPGA. Finalmente, no sexto capítulo reúnem-se as conclusões finais e levantam-se algumas ideias pertinentes para trabalho futuro.

2. Estado da Arte

2.1. Introdução

Os sistemas de ANPR, podem ser divididos em três etapas de processamento, a deteção da região da matrícula, a segmentação e reconhecimento de caracteres. A diversidade no formato da matrícula, e a falta de uniformidade da iluminação presentes em ambientes rodoviários condicionam o sucesso dos módulos de segmentação e deteção. O seguimento de objetos é um processo complexo, sendo difícil relacionar um objeto em dois fotogramas consecutivos, se este não apresentar uma característica ou identificador único. No seguimento de veículos, a sua matrícula pode ser utilizada, para reconhecer o objeto no processo de seguimento. No entanto, o sucesso fica condicionado pela velocidade de processamento e qualidade da deteção da região da matrícula.

Neste segundo capítulo, é levantado o estado da arte referente à segmentação de imagem e de movimento, e a sua aplicabilidade na deteção da região da matrícula. Posteriormente, são abordados os problemas subadjacentes às metodologias de deteção, propostas na literatura, para sistemas de ANPR e as suas limitações. Finalmente, são analisados os sistemas ANPR com aceleração em CUDA e FPGA, propostos na literatura.

No capítulo seguinte, é proposta uma nova abordagem no processo de segmentação da região da matrícula, que devido às suas características, proporcionará melhores resultados no seguimento de veículos.

2.2. Segmentação

O processo de segmentação subdivide uma determinada imagem, em diversas regiões ou objetos com atributos similares [9], sendo que o nível pela qual a subdivisão é realizada depende do problema a solucionar, ou seja quando os objetos de interesse forem isolados [10]. A segmentação de imagens é uma das tarefas mais difíceis no processamento de imagem, e a sua exatidão determina o eventual sucesso ou falha do processo de análise computacional [10].

É por isso importante, em alguns casos, como o de aplicações de inspeção industrial, estudar o ambiente que irá rodear a aplicação e se possível tentar controlar alguns dos fatores. O caso dos sistemas ANPR, pertence à gama de aplicações denominadas como *autonomous target aquisition,* onde o *designer,* não tem controlo sobre o ambiente que rodeia a sua aplicação. A abordagem a seguir deve então, passar pela seleção de sensores capazes de realçar os objetos de interesse e diminuir as regiões irrelevantes da imagem [10].

Os algoritmos de segmentação de imagem baseiam-se em duas propriedades básicas dos valores de intensidade: as descontinuidades, onde a imagem é particionada com base em mudanças acentuadas de intensidade, e as semelhanças, onde a imagem é dividida em regiões semelhantes, de acordo com um critério predefinido [10].

As diversas técnicas de segmentação, analisadas neste documento, podem ser divididas em duas categorias. A primeira categoria, segmentação de imagem, engloba todas as metodologias de segmentação, que sejam aplicadas a uma imagem estática, sem necessitar de informação dos fotogramas anteriores ou posteriores. A segunda categoria, segmentação de movimento, inclui todas as técnicas de segmentação que necessitem de diversos (mais do que um) fotogramas. Em seguida, é efetuado o levantamento do estado da arte, de algumas metodologias, bem como as suas vantagens e desvantagens, as quais podem ser utilizadas em aplicações de seguimento de veículos, em tempo real.

2.2.1. Segmentação de imagem

2.2.1.1. Region Growing

Region Growing é uma técnica de segmentação por região, agrupando pixéis da mesma intensidade (ou outra propriedade) em aglomerados cada vez maiores, até que toda a imagem seja segmentada. Esta técnica deve também suportar e facilitar a junção de regiões, bem como a sua separação quando se tornam demasiado extensas e perdem a sua homogeneidade [11]. Além de ser uma metodologia computacionalmente intensiva, e que não deve ser utilizada em sistemas de tempo real, também apresenta problemas no critério de atribuição de região, quando estão presentes: ruído, bordas afiadas e linhas isoladas (que não formem fronteiras) [12].

2.2.1.2. Thresholding

Quando a iluminação de *background* é uniforme, o objeto que se procura é liso e contrasta com o *background* da imagem, a segmentação pode ser alcançada através do *thresholding* num determinado nível de intensidade [13]. Aplicado a imagens em tons de cinzento, as técnicas de *threshold* analisam o valor de cada pixel (ou vizinhança de pixéis), presente no fotograma, e atribuem o valor zero quando o resultado fica abaixo do valor de *threshold* ou um quando fica acima, criando assim uma imagem binária.

Em aplicações, como a de reconhecimento de caracteres de um livro, o contraste entre as letras (*foreground*) e o *background* é bastante demarcado, dando a ilusão que a escolha de um valor de *threshold* possa ser ajustado "a olho". No entanto, alterações no nível de ruído, após a medição inicial podem comprometer os resultados, uma vez que o equilíbrio entre escuro e claro na imagem será alterado [13]. Esta técnica, apesar da sua limitação, é útil em aplicações industriais, onde o ambiente é parcialmente controlado.

A escolha de um valor adequado de *threshold* é então essencial para o bom funcionamento da metodologia, e esta depende em muito do ambiente esperado pela aplicação. Existem várias técnicas que tentam diminuir o erro na escolha, em seguida são presentes alguns exemplos.

Uma técnica muito utilizada na determinação do valor de *threshold,* envolve a análise do histograma, que representa os níveis de intensidade de uma imagem digital. Como descrito por [14], se um mínimo for encontrado, ele é interpretado como o valor de *threshold* da imagem. No entanto, esta metodologia apresenta diversos problemas, como o tamanho ou ruído inerente apresentado pelos vales do histograma, a escolha de um valor do mínimo indicado, valores de pico demasiado elevados, devido ao contraste com o *background*, que poderão introduzir erro na escolha, etc. O ponto mais importante a considerar, é que o histograma poderá ser inerentemente multimodal (Figura 2.2.1.2.3), quando se está apenas a tentar aplicar um valor único de *threshold* [13]. Nestes casos, é normal que o valor escolhido não seja o indicado, e uma análise completa da imagem é necessária antes de declarar os resultados válidos.

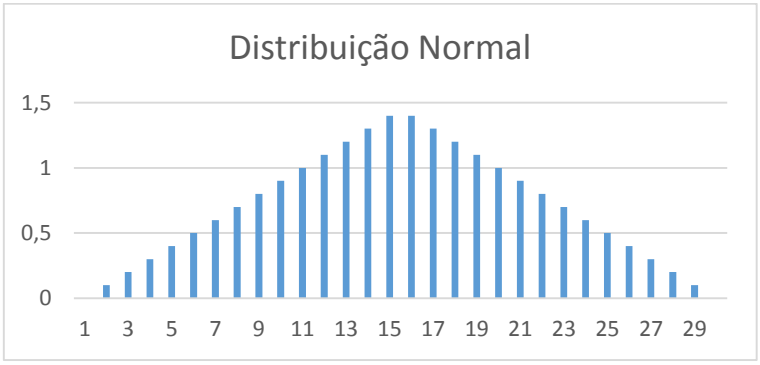


Figura 2.2.1.2.1 – Distribuição Normal de um Histograma

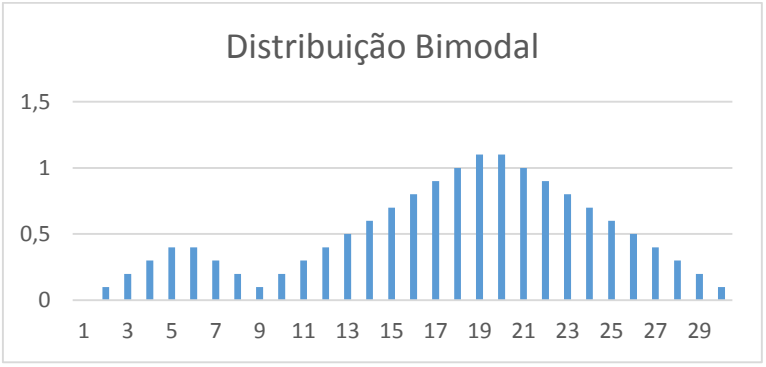


Figura 2.2.1.2.2 – Distribuição Bimodal de um Histograma

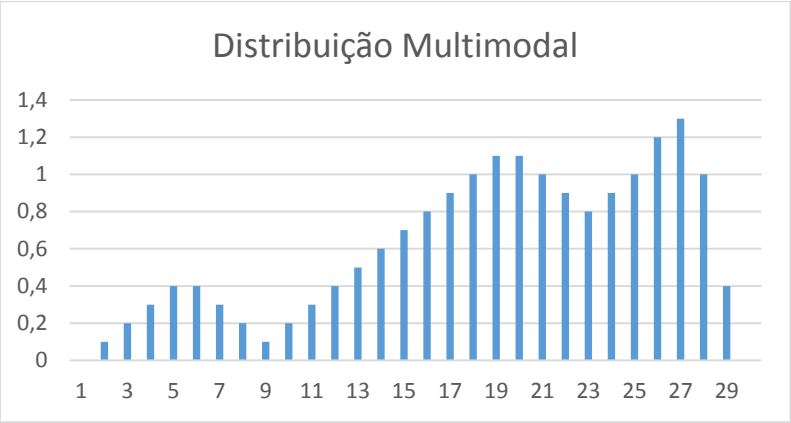


Figura 2.2.1.2.3 – Distribuição Multimodal de um Histograma

O *Otsu's method*, também conhecido por *Clustering Thresholding* [15], sugere minimizar a soma ponderada dentro das classes de variância dos pixéis, de primeiro e segundo plano, estabelecendo um valor ideal de *threshold*. Esta metodologia obtém bons resultados, quando o número de pixéis em cada classe são próximos entre eles. Devido a ser um método, que espera um histograma bimodal (Figura 2.2.1.2.2), onde apenas existem duas categorias de pixéis, o *background* e o *foreground,* não pode ser considerado o mais indicado para segmentação da matrícula. A sua extensão, a aplicações de *multithreshold,* propostas pelo autor [16] foi utilizada com algum sucesso em alguns sistemas de ANPR [17].

É então possível concluir, que as técnicas que analisam a imagem e atribuem um valor global de *threshold,* não resolvem os problemas de iluminação uniforme, reagindo mal ao reflexo, sombras e aglomerar de objetos em imagem [13]. Em situações reais, estes problemas de iluminação estão bem presentes e são bastante notórios, especialmente em ambientes rodoviários, onde cada veículo possuí iluminação própria. Pelo que foi descrito anteriormente, todas as metodologias que utilizem um valor de *threshold* global na binarização da imagem não são consideradas.

Uma abordagem baseada em *threshold* adaptativo é necessária, de forma a responder às dificuldades, impostas pela falta de uniformidade na iluminação em sistemas ANPR. O processo de segmentação por, *Threshold* local é proposto, no próximo capítulo, e a sua viabilidade é analisada em relação aos métodos tradicionais.

2.2.1.3. Edge Detection

O *Edge Detection (*deteção de contornos) é uma metodologia alternativa na segmentação de imagem, reduzindo consideravelmente a informação redundante presente na imagem, diminuindo o espaço necessário para armazenar informação, e o tempo de processamento dos módulos que o sucedem [18]. Os dois principais métodos de deteção de contornos são: o *template matching* e o *differential gradient*. Ambos os operadores estimam a intensidade local do gradiente, com o auxílio de máscaras de convolução adequadas [18]. O intuito da metodologia é analisar os pontos, onde os valores de intensidade da imagem, em tons de cinzento, mudam drasticamente, criando uma imagem binária contendo o conjunto de curvas de contorno da imagem.

Existem vários operadores propostos na literatura, no entanto, devido à carga computacional apresentada por alguns deles, é importante considerar o operador *Sobel*, pois apresenta um bom compromisso, entre a precisão do algoritmo e o tempo de processamento. Este operador, em conjunto com um gradiente vertical, é utilizado em alguns sistemas de ANPR [17], com o intuito de extrair todas as estruturas verticais, presentes na imagem. Esta técnica apresenta bons resultados em diversas implementações, onde a matrícula ocupa parte significativa do fotograma a analisar. No entanto, fatores como a inclinação e *backgrounds* mais complexos diminuem o sucesso da metodologia.

2.2.1.4. Histogramas

A segmentação de matrícula baseada em histogramas, utilizada em sistemas ANPR por [19], [20], [21] e [22], calcula a concentração de cor ou intensidade numa região. Relacionando os picos dos histogramas verticais e horizontais de uma imagem é possível obter as coordenadas x_0,y_0 (canto superior esquerdo) e x_1,y_1 (canto inferior direito). Não é uma técnica computacionalmente intensiva, pois não se trata de uma segmentação de movimento mas de uma forma de definir um objeto através da análise de uma imagem estática. Uma dificuldade desta implementação é, em alguns casos, a separação entre os picos e os vales [23].

2.2.1.5. Transformada de *Hough*

A deteção de linhas retas, em imagens é uma componente importante na análise de imagem e no reconhecimento de padrões visuais. A transformada de *Hough* apresenta resultados altamente robustos, na deteção de linhas retas em imagens digitais. Pode ser aplicado um detetor de contornos, para melhorar os resultados da transformada.

É uma metodologia particularmente tolerante ao ruído e oclusão parcial, porém a sua complexidade computacional e elevados requisitos de memória criam problemas na sua utilização em sistemas de tempo real [24].

Segundo *Duda e Hart* [25], uma linha pode ser representada pela Equação 2.2.1.5.1, em que ρ representa a distância da linha à origem da imagem, e θ o ângulo entre ρ e o eixo das abcissas (x). Para cada par de coordenadas (x_i, y_i), a equação descreve uma curva sinusoidal no espaço de parâmetros ρ θ, e cada ponto (ρ_i, θ_i) da curva representa uma linha no plano xy que passa por (x_i, y_i). É possível então concluir, que pontos colineares no plano xy partilham as mesmas coordenadas (ρ, θ). O algoritmo itera sobre cada par de coordenadas e um voto é adicionado sempre que um par específico (ρ, θ) é encontrado [26].

$$\rho = x \cdot cos\theta + y \cdot sin\theta$$

Equação 2.2.1.5.1 – Representação de uma linha

Quando o processo de votação termina, são selecionados os pontos (ρ, θ), superiores ao valor de *threshold* mínimo, para determinar se o par constitui uma linha, destacando as ocorrências mais proeminentes. É necessário que as regiões de interesse estejam suficientemente realçadas, de forma a obter uma votação elevada e ser possível distinguir do ruído de *background*. A eficiência desta metodologia está muito dependente, da qualidade do algoritmo de deteção de contornos utilizado [27].

A transformada de *Hough,* pode ser utilizada em sistemas ANPR na segmentação da região da matrícula. Através da deteção das linhas na imagem, é possível descobrir padrões que se assemelhem a uma matrícula, segmentando a região para confirmação posterior. No entanto, a sua implementação apresenta algumas limitações, como a deteção em ambientes urbanos, onde as linhas do *background* obtêm maior votação, do que as da matrícula, devido ao seu

comprimento. A análise em imagens que sofreram compressão também apresenta limitações, pois o processo pode deformar as linhas retas, diminuindo assim a taxa de sucesso da metodologia. Algumas regiões de matrícula podem ainda apresentar um padrão não reconhecido, seja através da sua forma, frisos ou contornos do veículo.

2.2.2. Segmentação de movimento

A Subtração entre *frames,* é um processo de segmentação por movimento. A sua implementação é rudimentar, suscetível a ruído e existe muita insegurança face aos resultados apresentados. Quando aplicado entre fotogramas adjacentes, apenas localiza, de uma forma muito limitada, os contornos do objeto em movimento. A modelação de *background*, descrita em seguida, iterou sobre os problemas desta metodologia e resolve algumas das dificuldades apresentadas [28].

A subtração de fundo, utilizada em [3], é um processo de segmentação que pretende separar o *foreground* do *background*. Utilizando os princípios de *background modeling*, tenta-se obter uma imagem de fundo ideal, ou seja, um fotograma onde se saiba que não existem objetos em movimento. Esta metodologia levanta duas grandes questões: como é possível saber que não existem objetos presentes que comprometem uma representação ideal do *background,* e como deve o sistema reagir às mudanças naturais da iluminação, que variam com o tempo e a hora do dia [28]. A vantagem desta técnica é a redução do nível de detalhe a analisar, uma vez que apenas o *foreground* é processado.

A mistura de gaussianos, utilizada na remoção de ruído de *background*, verifica a oscilação dos valores de intensidade de cada pixel no tempo e elimina pequenas variações, como por exemplo o abanar da vegetação. É uma técnica computacionalmente muito intensiva, pois obriga a que todos os pixéis sejam analisados individualmente, de forma a ser possível determinar o modelo da mistura de gaussianos [28]. Esta abordagem falha, quando há variações de elevada frequência na imagem [28].

O *optical flow* é uma metodologia de segmentação por movimento. Aplica um operador local a todos os pixéis da imagem, cria um campo de vetores de movimento que varia suavemente

ao longo da imagem [29]. Ao estimar o *optical flow* entre diferentes fotogramas, é possível caracterizar e quantificar o movimento descrito pelos objetos, obtendo a velocidade e sentido a que se deslocam [30]. Apresenta um *overhead* comparável ao de um detetor de contornos (quando aplicado a uma imagem em tons de cinzento). No entanto, tem que ser aplicado a uma sequência de imagens [29], o que aumenta o processamento, a ocupação e o respetivo acesso à memória. O principal problema desta metodologia, é a suscetibilidade apresentada à variação de luminosidade, devido ao cálculo do movimento através da variação de intensidade, numa sequência de imagens.

2.3. Morfologia

A morfologia matemática, é composta por um conjunto de algoritmos e métodos, que analisam e processam figuras geométricas, normalmente presentes numa imagem digital. É especialmente importante, pois fornece o *backbone* para o estudo da forma, sendo capaz de unificar técnicas tão díspares como a supressão de ruído, análise de forma*, feature recognition*, *skeletonization*, *convex hull formation*, entre outros [31]. Desenvolvida inicialmente para o processamento de imagens binárias e posteriormente estendida a imagens em tons de cinzento, a matemática morfológica é bastante utilizada em sistemas ANPR [17].

A sua aplicação em imagens binárias apresenta quatro operadores básicos: erosão, dilatação, *opening* e *close*. Em seguida, é apresentado um resumo dos operadores básicos e suas aplicações. O operador de reconstrução morfológica é abordado mais à frente, no próximo capítulo.

O primeiro passo é a escolha de um elemento estruturador (B), que determina a máscara que é aplicada a toda a imagem (A), e influencia o resultado das operações morfologias. As suas principais características são: a sua forma e o seu tamanho, sendo que a sua escolha é feita, com conhecimento prévio sobre a forma do objeto que se pretende encontrar na imagem.

2.3.1. Dilatação

Expande os objetos pelo *background*, sendo capaz de eliminar ruído ou preencher falhas inferiores ao elemento estruturador utilizado [31]. A escolha da máscara de dilatação (elemento estruturador) define a vizinhança, que é considerada pelo operador, sendo esta geralmente de 3x3. A janela é centrada no pixel que se pretende analisar, e é dilatada pela máscara escolhida como se pode confirmar na Equação 2.3.1.1.

$$A \oplus B = \bigcup A_B$$

Equação 2.3.1.1 – Dilatação da imagem A pelo elemento estruturador B

Na deteção da matrícula, a dilatação poderá ser utilizada após um processo de deteção de contornos, para realçar as linhas e letras da imagem. No entanto, pode afetar também outros objetos presentes na imagem e juntar regiões de interesse, dificultando o processo de deteção.

2.3.2. Erosão

Em contraste com o operador anterior, o processo de erosão diminui os objetos em imagens binárias, removendo algum ruído "picotado" e objetos finos, com largura inferior ao elemento estruturador [31]. O processo de seleção da máscara de erosão é igual ao de dilatação. A operação de erosão pode ser descrita pela Equação 2.3.2.1.

$$A \ominus B = \bigcap A_{-B}$$

Equação 2.3.2.1 – Erosão da imagem A pelo elemento estruturador B

A dilatação e a erosão são transformadas duplas, no que diz respeito ao complementar, ou seja, o processo de erosão de uma imagem é o complementar do processo de dilatação, quando o mesmo elemento estruturador é utilizado. No entanto, elas não processam os objetos e o seu *background* de uma forma simétrica, não sendo possível recuperar a imagem inicial, na sua totalidade, após uma das operações. O processo de erosão diminui os objetos, expandindo o seu *background*, enquanto o de dilatação expande os objetos, diminuindo o *background* [32]. Em

sistemas de ANPR, o operador de erosão pode ser utilizado para diminuir o detalhe da imagem binária, quando a região da matrícula ocupa uma porção significativa da imagem.

2.3.3. Opening e Close

A dilatação e erosão são operadores básicos, sobre os quais outros podem ser derivados [31]. O processo de dilatação, seguido por uma erosão, gera o operador *Closing* (•), utilizado para eliminar fissuras entre objetos, remover *salt noise*, pequenos buracos ou concavidades [31]. O processo de erosão, seguido por uma dilatação, gera o operador *Opening* (o), utilizado para abrir fissuras em objetos, remover *pepper noise* e pequenas saliências [31]. Estes novos operadores podem ser definidos pela Equação 2.3.3.1 e Equação 2.3.3.2, respetivamente.

$$A \cdot B = (A \oplus B) \ominus B$$

Equação 2.3.3.1 – Closing

$$A \circ B = (A \ominus B) \oplus B$$

Equação 2.3.3.2 – Opening

Ao subtrair o resultado da operação morfológica, de *opening* ou *close,* pela imagem original, é possível detetar vários tipos de defeitos em objetos, tornando-os dois operadores muito importantes para tarefas como: localizar curto-circuitos na solda e falhas na impressão de pistas de PCBs (*Printed Circuit Board*) [31].

Em [33], o autor propõe aplicar o operador de *sobel* vertical, sobre uma imagem em tons de cinzento, seguido da operação de *closing,* com um elemento estruturador retangular de 3x22, para unir os pixéis, e da operação de *opening* com um elemento estruturador de 3x5, com o intuito de filtrar informação não desejada. Esta implementação falha pelo facto de a distância entre a matrícula e a câmara de videovigilância, estar limitada pelas dimensões do elemento estruturador. O sistema pode também confundir um objeto retangular com uma região da matrícula.

2.4. Problemas Associados à Segmentação de matrícula

O reconhecimento automático de matrícula (ANPR), em imagens ou vídeo, é composto geralmente, por três etapas de processamento: a extração da região da matrícula, a segmentação e o reconhecimento dos caracteres. É um processo complexo, devido às condições não uniformes apresentadas pela iluminação exterior, e pela diversidade das matrículas, pelo que grande parte das aplicações funciona apenas sobre condições restritas de iluminação, velocidade, trajetória ou que apresentem um *background* estacionário [8].

Uma comparação direta, entre sistemas de ANPR, não pode ser estabelecida, devido à falta de uniformidade pela qual as metodologias são avaliadas, e às diferentes condições de funcionamento [8]. No entanto, pode ser efetuado um estudo sobre as limitações, que os diversos sistemas apresentam.

As técnicas que recorrem ao processamento binário de imagem, a junção entre a metodologia de deteção de contornos com operações morfológicas apresentam bons resultados [8]. Esta metodologia assenta no princípio que, as mudanças de intensidade são mais acentuadas, e frequentes na região da matrícula. No entanto, a deteção de contornos não pode ser aplicada a imagens complexas, uma vez que se mostra demasiado sensível a contornos indesejáveis, sendo a morfologia utilizada na redução ou eliminação de contornos indesejados [8].

O operador de *sobel* vertical é bastante citado na literatura [17], e apresenta resultados satisfatórios. Após o operador, são empregues diversas técnicas, dependendo do autor, para a remoção de ruído de *background,* seguido por uma pesquisa da região da matrícula através de uma janela retangular [8]. O processo não é tolerante a inclinações quer da região da matrícula, quer do pavimento.

A *Connected Component labeling* analisa e extrai os componentes ligados*,* de uma imagem binária. O sucesso desta metodologia, depende em grande parte do processo utilizado na segmentação de imagem, em tons de cinzento.

A transformada de *Hough* é aplicada a uma imagem binária, que contém os contornos da imagem, em tons de cinzento, na deteção da matrícula. Devido a ser uma técnica computacionalmente intensiva, na literatura é proposta uma combinação da transformada com

um algoritmo de deteção de contornos [8]. A limitação do anglo e da região de deteção, bem como a implementação de uma *lookup table,* são necessários para obter um bom tempo de resposta do sistema. No entanto, a transformada de *Hough* é bastante sensível à deformação de contornos, pelo que apenas apresenta bons resultados, quando aplicada a imagens adquiridas próximas do veículo.

No conjunto das técnicas de deteção da região da matrícula, através da análise de imagens em tons de cinzento*,* sobressaem as metodologias que utilizam a transformação de imagem. Os filtros de *Gabor*, uma das ferramentas mais utilizadas na análise de textura, apresentam uma boa taxa de deteção (98%), no entanto o método foi apenas testado em pequenas amostras de imagens uma vez que o processo é computacionalmente muito intensivo [17]. A *wavelet transform* é utilizada com o intuito de extrair caraterísticas importantes sobre o contraste da imagem. No entanto esta metodologia é pouco fiável quando a câmara está muito perto, ou longe, do veículo ou quando o anglo de visão não é o melhor [17].

As técnicas que utilizam imagens a cores para a deteção da região da matrícula não são consideradas, pois requerem requisitos muito superiores de memória e processamento por parte do sistema. Para além de não haver bons resultados na deteção noturna, a cor das matrículas é diferente de país para país e até entre veículos de categorias diferente.

Através da revisão do estado da arte, de sistemas ANPR, é possível concluir que estes são restringidos a condições de funcionamento como a perspetiva, distância, *background*, iluminação e posição do veículo [17]. Módulos de infravermelhos são utilizados, com sucesso, para solucionar o problema de falta de iluminação. O foco, neste momento, dos sistemas de ANPR propostos é a deteção de um veículo, mesmo que mais estejam presentes na imagem a analisar [17]. É por isso importante estudar e propor uma nova metodologia que não seja tão dependente destas restrições, com um algoritmo que necessite de pouca calibração, sendo assim possível adaptar esta solução a novas aplicações.

O avanço da tecnologia permite utilizar algoritmos de deteção mais robustos, mantendo as restrições de resposta em tempo real, considerado na literatura como 20 fotogramas por segundo [17]. Um sistema configurável é importante, pois o utilizador pode escolher qual a

prioridade, entre a taxa de sucesso na deteção, a qualidade da imagem ou a velocidade de processamento, através de módulos de pré e pós-processamento.

2.5. Sistemas ANPR em CUDA

Em [34], uma técnica de OCR com base em redes neuronais, com aceleração em CUDA é proposta. O recurso ao GPU e à sua arquitetura inerentemente paralela permitiu, acelerar o processo de teste e aprendizagem, mantendo baixos custos de desenvolvimento. As matrículas persas foram utilizadas, como caso de teste para o sistema. No entanto, a informação referente aos resultados obtidos é muito vaga. No momento da escrita deste documento, este foi o único artigo relacionado com sistemas ANPR em CUDA encontrado na literatura.

2.6. Sistemas ANPR em FPGA

Recorrendo à aceleração por *hardware,* torna-se possível estender os sistemas ANPR de *Standard Definition* (SD) para *High Definition* (HD) sem que haja um aumento significativo do tempo de processamento, mantendo assim as características de tempo real [35] [36]. Este aumento na qualidade, dos fotogramas processados, proporciona melhores resultados de OCR [4], aumentado assim a fiabilidade do sistema.

Em [33], é proposta a utilização do operador de *sobel* vertical, para a deteção de contornos na imagem, em conjunto com um valor específico de *threshold,* que elimina grande parte da informação não desejada. Uma vez que a região da matrícula apresenta valores de intensidade mais elevados, é possível realizar a segmentação entre regiões recorrendo a operações morfológicas.

Finalmente, em [3], é proposto um sistema de ANPR utilizando a subtração de *background* e operações morfológicas, sobre imagens estáticas (de uma base de dados) com uma resolução de 640x480, e obtendo um tempo de resposta de cerca de 11ms, na localização da matrícula e segmentação de caracteres. O sistema foi implementado em *hardware*, na sua totalidade.

3. Pré-Processamento e Segmentação

3.1. Introdução

O sistema recebe uma sequência de imagens em tons de cinzento, provenientes de uma câmara de videovigilância, sem compressão, e tem como objetivo segmentar as regiões da matrícula presentes em cada fotograma. As regiões selecionadas são posteriormente identificadas (OCR), atribuindo um identificador único, de forma a ser possível obter um processo de seguimento robusto.

As técnicas propostas na literatura, de uma forma geral, apenas garantem bons resultados sob condições restritas. O intuito das metodologias é detetar a região da matrícula numa sequência de fotogramas, pelo menos uma vez, para a deteção ser considerada válida. A câmara de videovigilância apresenta relativa proximidade ao veículo, pelo que a região da matrícula ocupa uma proporção considerável da imagem e o ambiente rodoviário (*background*) não é destacado, pois pode interferir com o processo de deteção.

Neste terceiro capítulo, é proposta uma nova abordagem no processo de segmentação da região da matrícula, com o intuito de conseguir a deteção da região da matrícula, no maior número de fotogramas consecutivos possíveis. Para que tal seja possível, o módulo de deteção deve ser capaz de responder às diferentes posições, e proporções que a região da matrícula apresentará numa sequência de fotogramas. Esta distanciação, entre a câmara de videovigilância e a região da matrícula, implica que o sistema deve ser tolerante às diferenças de luminosidade e alterações de *background*, no entanto tornará possível a deteção de mais do que um veículo por fotograma. A aceleração do sistema será também um ponto importante, pois quanto maior o número de FPS analisado, melhor será o resultado do processo de seguimento.

Para responder a alguns dos requisitos levantados, um sistema de deteção, baseado em *Threshold* local e *Flood Fill* é apresentado, onde a primeira metodologia torna o sistema mais impermeável às variações de luminosidade, diminuí o nível de detalhe e realça o contraste entre os caracteres e o *background* da região da matrícula, a segunda procura preencher os caracteres de forma a segmentá-los do resto da imagem. Um módulo de pré-processamento e outro de pós-

processamento são propostos, de forma a melhorar a aceleração e resultados do sistema. Finalmente, são analisados e discutidos os resultados experimentais.

3.2. Local Threshold

Quando a iluminação não é suficientemente uniforme, surgem problemas na segmentação (por cor ou intensidade) e identificação de imagens. Uma possível abordagem, para a resolução deste tipo de problemas, passa por permitir a variação adaptativa do *threshold* ao longo de toda a imagem. A análise dos valores de intensidade da vizinhança em cada pixel, com o intuito de determinar o melhor valor de *threshold* local, é uma tarefa computacionalmente muito intensiva, sendo necessário recorrer a um mecanismo de amostragem eficiente [37]. A média da distribuição de intensidade local proporciona bons resultados. Esta técnica é particularmente eficiente, quando o tamanho da vizinhança escolhida, para estimar o nível de *threshold,* é suficientemente grande para abranger uma área significativa do *background* e *foreground*. Um dos casos notáveis de aplicação é a segmentação de caracteres (aplicando posteriormente OCR e efetuando o seu reconhecimento), pois o seu comprimento e largura é conhecido, permitindo assim a escolha de uma vizinhança apropriada na segmentação dos caracteres do *background* [37].

A metodologia apresenta características interessantes para a segmentação de matrículas, devido às variações de luminosidade verificadas no ambiente rodoviário, demonstrando bons resultados na segmentação de caracteres e reduzindo a carga computacional através do recurso a um mecanismo de amostragem.

O intuito desta técnica é diminuir, o nível de detalhe presente nas imagens, e fazer sobressair as formas geométricas dos objetos de interesse. No caso de aplicação em sistemas de ANPR, o seu principal objetivo é realçar o formato dos caracteres e o seu contraste com a chapa da matrícula dos veículos. O tamanho da janela da vizinhança pode ser definido através de calibração prévia, uma vez que os caracteres presentes, nas matrículas dos veículos, apresentam dimensões regularizadas. Assim, apenas a distância entre o local onde é colocada a câmara de videovigilância e a resolução da imagem devem ser consideradas.

O intuído do algoritmo é calcular a média dos valores de intensidade, na vizinhança do pixel atual. O processo não é aplicado às margens da imagem, uma vez que o valor atual necessita de estar centrado numa vizinhança, sendo possível a inclusão das margens recorrendo à expansão por repetição. No entanto, o processo não é considerado devido à carga computacional exigida. Após o cálculo da média, cada pixel é submetido ao processo de *threshold*, como descrito pela Equação 3.2.1.

$$Threshold = Mean - (Maximum - Mean)$$

Equação 3.2.1- Formula para calcular o valor de Threshold Local

Esta metodologia é uma excelente candidata para aceleração em *hardware*, pois apenas necessita de informação relativamente aos valores da vizinhança do pixel a analisar. É facilmente adaptada a arquiteturas paralelas, onde apenas o número de *threads* disponíveis e o acesso à memória, condiciona o desempenho do sistema. No entanto, é necessário adaptar alguns parâmetros para que seja possível aplicá-la como primeira etapa na segmentação de matrícula.

3.3. Flood Fill

O *Flood Fill* é uma técnica de preenchimento, que propaga uma determinada cor sobre os componentes ligados, de uma região selecionada. Utilizada em ferramentas como o "*bucket fill*", presente em diversos *softwares* de edição de imagem, ou em jogos como o "*Minesweeper*" para determinar que peças foram limpas [38].

O algoritmo apresenta três parâmetros de entrada: o ponto inicial (*node*), a cor alvo (*target color*) e a cor de substituição (*replacement color*). Sempre que a cor do pixel analisado for igual à cor alvo, o valor do pixel é alterado para o valor RGB da cor de substituição.

Os pixéis na vizinhança próxima, do ponto atual, são então adicionados a uma lista, para serem processados nas próximas iterações do algoritmo. A vizinhança pode ser *4-connected* (Norte, Sul, Oeste, Este) ou *8-connected* (Norte, Sul, Oeste, Este, NO, NE, SO, SE), como representado na Figura 3.3.1. A primeira apresenta um processo menos intensivo computacionalmente. No entanto, não proporciona a qualidade de resultados da região *8-connected*.

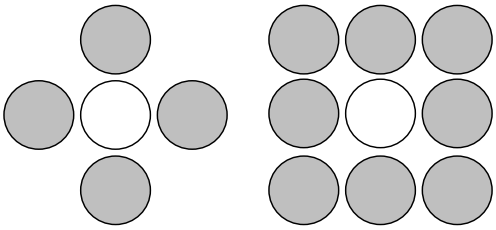


Figura 3.3.1 – Vizinhança 4-connected e 8-connected

3.3.1. Stack based recursive (4-connected)

Stack based recursive é uma simples implementação do algoritmo de *Flood Fill,* que recorre à recursividade, como demonstra a Figura 3.3.1.2, para percorrer a imagem e preencher os componentes ligados. Por cada pixel alterado, são feitas quatro chamadas à função de *FloodFill*, representando as quatro coordenadas da vizinhança a analisar. Quando a primeira sub-rotina é invocada, uma *stack frame* nova é criada por cima da anterior, guardando os parâmetros e variáveis locais da sub-rotina e o endereço de retorno do *Caller Function,* como representado na Figura 3.3.1.1.

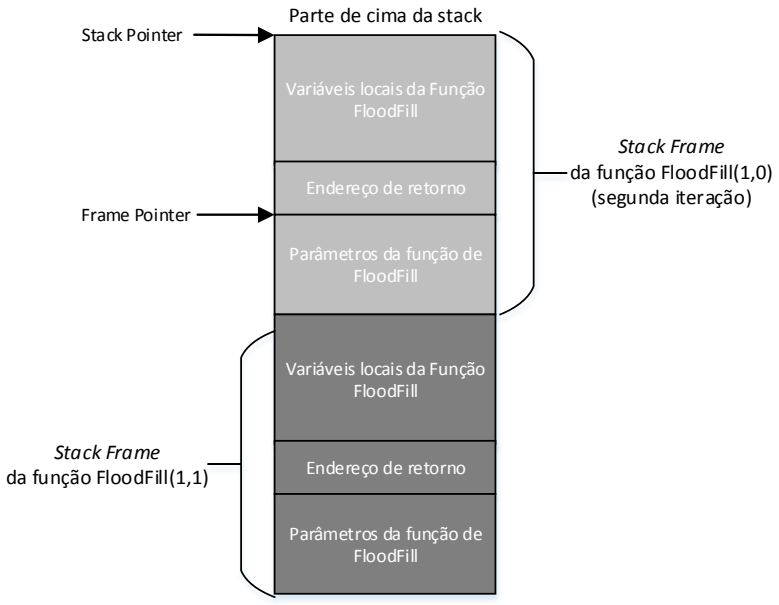


Figura 3.3.1.1 – Representação da Call stack

Em casos de aplicação, como em processamento de imagem, que apresentam uma elevada resolução de imagem, as sucessivas chamadas recursivas vão ultrapassar o espaço limite para reserva de espaço na pilha *(stack overflow)*. Pode-se então concluir que esta implementação não pode ser considerada, devido às restrições de memória da *Call stack* impostas por diversas linguagens e ambientes.

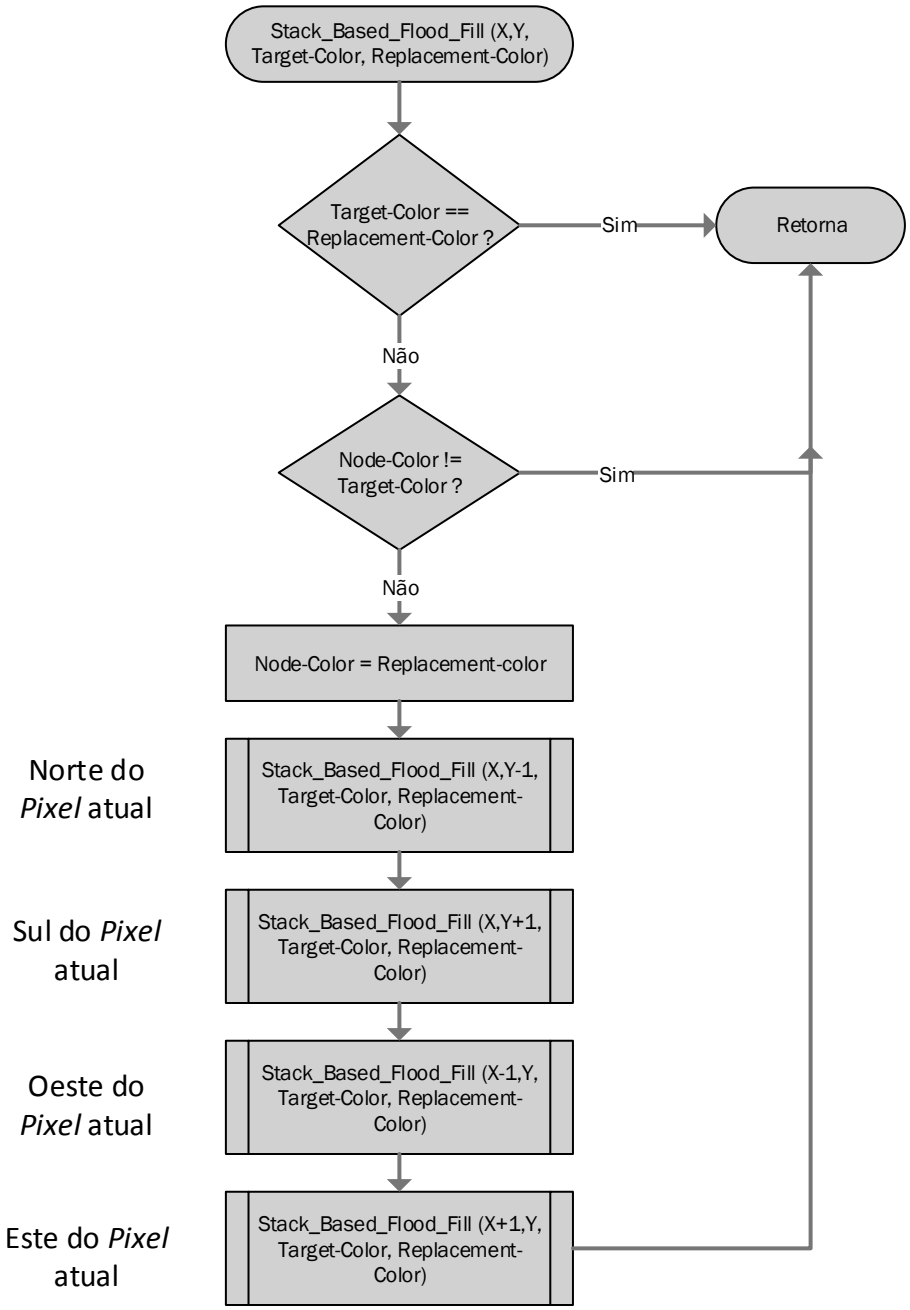


Figura 3.3.1.2 – Stack based Flood Fill

3.3.2. *Queue based*

Alternativamente pode ser considerada uma solução, como a descrita na Figura 3.3.2.1, baseada num sistema que guarda os valores numa *queue* (*heap memory*) e utiliza um ciclo que só termina quando consumir todos os valores na *queue*.

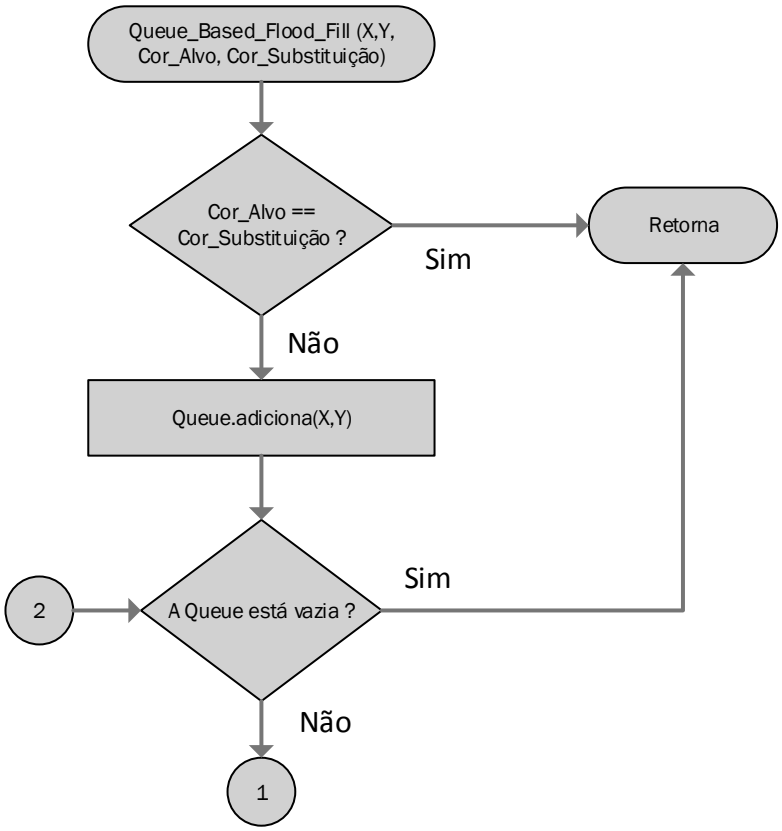


Figura 3.3.2.1 – Queue Based Flood Fill Part I

Esta implementação já não apresenta as limitações de memória da anterior, o "tamanho" das variáveis pode ser alterado e estas podem ser acedidas globalmente. No entanto, a memória *heap* apresenta um acesso relativamente mais lento (face à *stack*), não garante uma ocupação de espaço eficiente (a memoria pode ficar fragmentada ao longo do tempo), e o programador fica responsável pela gestão de memória da aplicação [39].

Cada iteração do ciclo do algoritmo começa por consumir o primeiro valor, retirando-o da *queue*, verifica as características do *node,* e se a sua cor for substituída adiciona os valores da vizinhança (*4-connected*) ao final da *queue*. Este processo é repetido, até não existir nenhum valor na fila de espera, como é possível verificar na Figura 3.3.2.2.

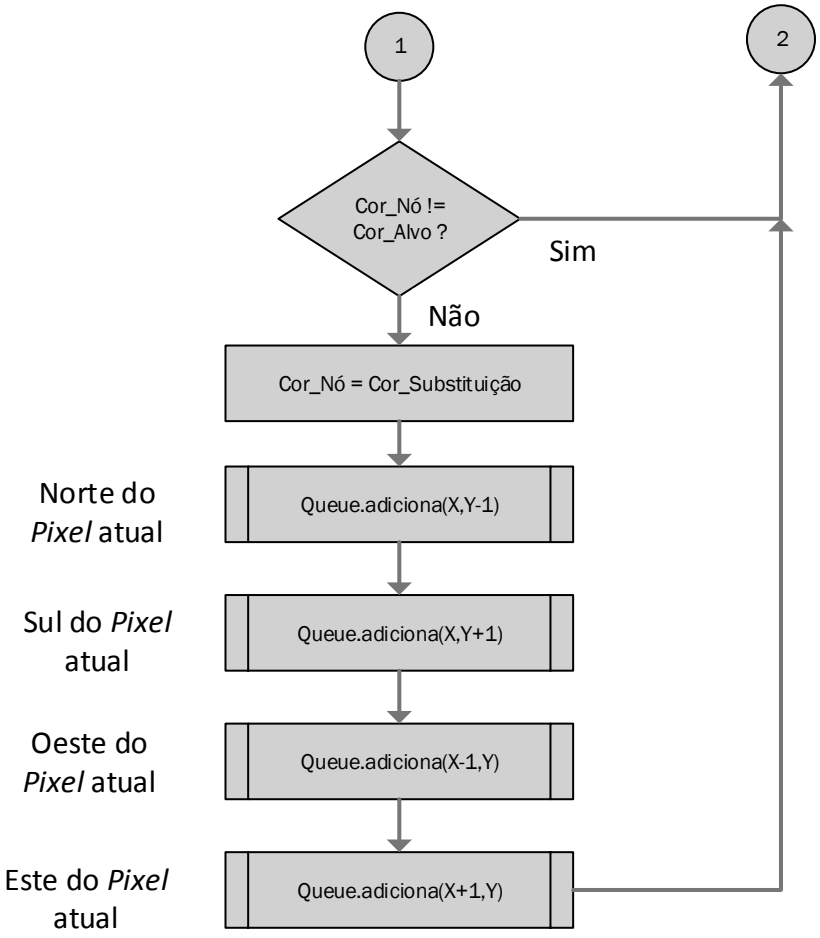


Figura 3.3.2.2 – Queue Based Flood Fill Part II

O problema desta implementação advém, da repetição de pixéis com as mesmas coordenadas na *queue*, algo que torna o processo moroso, especialmente no processamento de imagens de elevada resolução, onde são analisados cerca de dois milhões de pixéis.

3.3.3. Scanline Fill

É uma metodologia de *Flood Fill,* que utiliza preenchimento por linhas, ou seja, em vez de enviar para a *stack/queue* cada possível coordenada de pixéis a analisar, o algoritmo procura segmentos de reta adjacentes e envia apenas o ponto inicial e final, para serem preenchidas posteriormente. Contrariamente às implementações anteriores, cada pixel é verificado apenas uma vez, acelerando assim o processo. No entanto, esta não é uma boa abordagem em aplicações de processamento de imagem, onde o pavimento (Figura 3.3.3.1), a vegetação ou mesmo ruído inerentes à imagem afetam a qualidade do algoritmo.

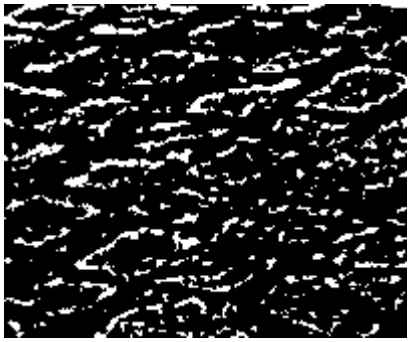


Figura 3.3.3.1 – Imagem binária de um pavimento

3.3.4. Reconstrução Morfológica

É um operador importante, que é providenciado pela matemática morfológica e insere-se num conjunto de operações sobre imagens, normalmente conhecidos como transformações geodésicas. Contrariamente às restantes operações morfológicas, que apenas utilizam uma imagem de entrada e um elemento estruturador, esta abordagem passa por aplicar uma transformação morfológica à primeira imagem e forçar os seus valores abaixo ou acima dos valores da segunda [40].

A escolha de um elemento estruturador deixa então de ser necessária pois, na prática, as transformações geodésicas são iteradas até que a estabilidade seja atingida, tornado a escolha do tamanho do elemento estruturador desnecessária. As primitivas morfológicas são criadas através

da combinação apropriada, do par formado pelas duas imagens de entrada, marcador e máscara, criando uma base importante na definição de diversas estruturas para imagens binárias e em tons de cinzento [40]. A escolha de um bom par de imagens é importante, sendo que a imagem sobre estudo é definida como máscara. A imagem marcador pode ser determinada, utilizando alguns conceitos como o conhecimento prévio do resultado esperado, factos consumados sobre a imagem, propriedades físicas do objeto que representa, marcadores manualmente definidos, entre outros [40]. Uma aplicação muito importante desta metodologia é a extração de componentes ligados, de uma imagem binária g (máscara), marcados em f (marcador) e contida em g [41].

Como descrito por [42] em 2010, a reconstrução morfológica é um método importante, mas pouco conhecido, para extração de informação importante, sobre formas presentes em imagens, como letras num documento digitalizado, núcleos de células realçados, imagens de galáxias distantes captadas por telescópios com infravermelhos, etc. A reconstrução morfológica pode ser utilizada para extrair objetos marcados, encontrar regiões brilhantes cercadas por pixéis escuros, detetar ou remover objetos que intersectem as margens da imagem, detetar ou preencher buracos, filtrar pontos altos ou depressões, entre outras operações [42].

Em [41], o autor apresenta aplicações e algoritmos eficientes para efetuar a reconstrução morfológica em imagens binárias, e a sua extensão a imagens em tons de cinzento. Sendo um artigo publicado 1993, a sua informação ainda é relevante atualmente, estando os seus algoritmos presentes em produtos comerciais como o matlab. A otimização deve passar por adaptar as soluções apresentadas, às tecnologias presentes nos dias correntes, retirando maior partido das arquiteturas disponíveis. Nos próximos capítulos deste documento são apresentadas duas possíveis implementações, em CUDA e FPGA, do algoritmo de reconstrução morfológica.

Em [43], é apresentado um algoritmo para a decomposição de imagem, recorrendo a um filtro de reconstrução morfológica. O seu objetivo consistiu em, realçar o contraste em aplicações de imagens médicas. O algoritmo foi implementado em FPGA, de forma a responder aos requisitos de tempo real impostos.

Em [44], é proposta a utilização da metodologia na análise e processamento de estruturas espaciais de largas proporções e em três dimensões. A implementação recorreu ao processamento do algoritmo em GPU (CUDA) para aceleração.

O processo de *Fill Hole* é utilizado por [45], após aplicar um *threshold* global na fase de pré-processamento, tentando identificar na ressonância magnética, possíveis regiões semelhares a um tumor no cérebro humano.

A característica iterativa desta metodologia torna-a computacionalmente muito intensiva e dificulta a previsão da condição de paragem. Apenas quando não existir alteração nos valores dos pixéis da imagem marcador, em duas iterações consecutivas, se pode considerar atingida a estabilidade.

3.3.5. Reconstrução por Dilatação

Como referido anteriormente, o processo utiliza duas imagens de entrada, máscara e marcador, que por definição pertencem ao mesmo domínio, sendo que os valores da imagem máscara devem ser sempre superiores ou iguais aos valores do marcador. O marcador é dilatado uma vez pelo elemento estruturador isotrópico (B), e os seus valores são posteriormente forçados a manterem-se iguais, ou abaixo, dos valores da máscara, funcionando essencialmente como um limite à propagação da dilatação da imagem marcador [40].

Seja f a imagem marcador, g a imagem máscara com Df = Dg e $f \leq g$. A dilatação geodésica de tamanho um pode ser representada pela Equação 3.3.5.1.

$$\delta_g^{(1)}(f) = \delta^{(1)}(f) \cap g$$

$$(=)\ \delta_g^{(1)}(f) = (f \oplus B) \cap g$$

Equação 3.3.5.1 – Dilatação geodésica

A transformação geodésica de imagens converge sempre, após um número finito de iterações. A reconstrução por dilatação pode então ser alcançada quando a dilatação geodésica de f por g é iterada até a estabilidade, e pode ser definida pela Equação 3.3.5.2 [40]. A dilatação geodésica de uma imagem binária pode ser ilustrada pela Figura 3.3.5.1.

$$R_g^{\delta}(f) = \delta_g^{(n)}(f), \text{ onde n é tal que: } \delta_g^{(n)}(f) = \delta_g^{(n+1)}(f)$$

Equação 3.3.5.2 – Reconstrução morfológica por dilatação

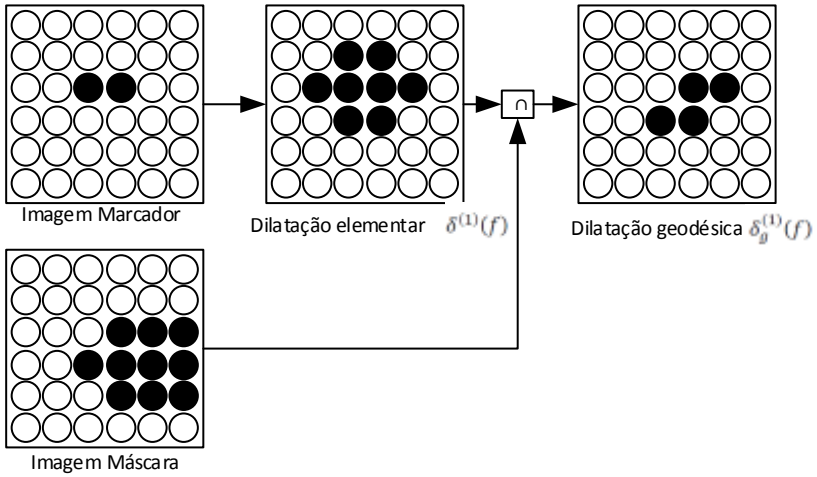


Figura 3.3.5.1 – Dilatação geodésica

É possível então concluir, que em sucessivas dilatações geodésicas elementares do marcador f contidas na máscara g, os componentes ligados de f, cuja interseção com g não representam o conjunto vazio, são progressivamente inundados (*flooded*) [41], como representado pela Equação 3.3.5.3.

$$\rho g(f) = \bigcup_{n \geq 1} \delta_g^{(n)}(f)$$

Equação 3.3.5.3 – Sucessivas dilatações geodésicas

3.3.6. Reconstrução por Erosão

A operação de erosão geodésica, representada por $\varepsilon_g^1(f)$, pode ser definida como o par complementar da transformada dupla da dilatação geodésica, como demonstrado na Equação 3.3.6.1.

$$\begin{aligned}\varepsilon_g^1(f) &= \overline{\delta^{(1)}(\bar{f}) \cap \overline{g}} \\ &= \overline{\varepsilon^{(1)}(f) \cap \overline{g}} \\ &= \varepsilon^{(1)}(f) \cup g,\end{aligned}$$

Equação 3.3.6.1 – Definição de erosão geodésica, partindo do processo de dilatação.

onde $f \geq g \; e \; \varepsilon^{(1)}$ representa a operação elementar de erosão. A imagem marcador é então erodida, atuando a imagem máscara como limite para o encolhimento, sendo que a imagem resultante é sempre maior ou igual a esta.

A erosão geodésica, de ordem n de uma imagem com o marcador f pela máscara g, pode ser obtida após n sucessivas erosões geodésicas de f em relação a g, como demonstrado na Equação 3.3.6.2. Quando a equação anterior é iterada até à estabilidade, obtemos a Equação 3.3.6.3 atingindo a reconstrução morfológica por erosão.

$$\varepsilon_g^{\mathrm{n}}(f) = \varepsilon_g^1\left[\varepsilon_g^{\mathrm{n}-1}(f)\right],\ \varepsilon_g^0(f) = f$$

Equação 3.3.6.2 – Sucessivas erosões geodésicas

$$R_g^{\varepsilon}(f) = \varepsilon_g^{\mathrm{n}}(f)$$

Equação 3.3.6.3 – Condição de estabilidade, Reconstrução morfológica por erosão.

3.3.7. Fill Hole

Em diversos casos de aplicação, é necessário remover os componentes ligados às margens da imagem, uma vez que podem influenciar o estudo estatístico sobre as propriedades da imagem a analisar. As regiões diretamente ligadas às margens podem ser removidos, utilizando a imagem de entrada como máscara e a interceção entre a imagem de entrada e a sua margem

como marcador. Consequentemente, a reconstrução morfológica contem todos os componentes ligados das margens da imagem.

No caso das imagens binárias, pode-se definir os buracos (*holes*) da imagem, como o conjunto de componentes do *background* que não estão ligados às margens da imagem. A subtração da imagem resultante com a original retorna os buracos da imagem. É possível estender esta metodologia a imagens em tons de cinzento, definindo os buracos como o conjunto de valores da *regional minima* que não estão diretamente ligados às margens da imagem [40]. Esta é uma técnica importante na segmentação de caracteres e figuras geométricas, *feature extraction,* entre outros.

3.3.8. Algoritmos propostos na literatura

Na literatura estão identificadas quatro possíveis abordagens, na implementação do algoritmo de reconstrução morfológica em imagens binárias e em tons de cinzento. O documento mais proeminente [41], publicado em 1993 por *Luc Vicent* na revista *IEEE Transactions on Image Processing*, faz um levantamento das técnicas utilizadas previamente (*Parallel* e *Sequential Reconstruction*) e introduz duas novas abordagens (*Reconstruction using a queue of pixels* e *A Fast Hybrid Grayscale Reconstruction*). Posteriormente, diversos autores [46], [47] e [48] propuseram melhorias ao seu algoritmo, utilizando um sistema de *queues* prioritárias mais otimizadas. Atualmente, o estudo científico [44] e [43], tem recaído sobre como melhor adaptar as metodologias propostas às arquiteturas disponíveis.

3.3.9. Parallel Reconstruction Algorithm

Reconhecida na literatura [41] como a implementação *standard*, a reconstrução por dilatação (Figura 3.3.9.1) consiste na iteração de dilatações geodésicas elementares, da imagem máscara g pelo marcador f, até que a estabilidade seja alcançada.

- I: Imagem máscara (binária ou tons de cinzento)
- J: Imagem marcador, definida no domínio D_1, $J \leq I$.
  - A reconstrução é determinada diretamente em J
- Reservar espaço para a imagem K definida em D_1
- Repetir até atingir a estabilidade:
  - Processo de dilatação: por cada pixel $p \in D_1$
    - $K(p) \leftarrow \max \{J(q), q \in N_g(p) \cup \{p\}\}$
  - *Pointwise minimum*: Por cada pixel $p \in D_1$
    - $J(p) \leftarrow \min(K(p), I(p))$

Figura 3.3.9.1 – Parallel Reconstruction Algorithm [41]

Em cada um dos passos da etapa anterior, a imagem pode ser percorrida numa ordem arbitrária, tornando a implementação deste algoritmo numa arquitetura paralela extremamente fácil e eficiente [41]. Devido ao elevado número de iterações necessárias para atingir a estabilidade, esta abordagem não pode ser considerada em processadores sequenciais e mesmo em arquiteturas paralelas é necessário estudar a sua viabilidade, face às outras implementações propostas.

3.3.10.Sequential Reconstruction Algorithm

Numa tentativa, de diminuir o número de interações necessárias para concluir o processo de reconstrução morfológica, foi proposto a utilização de um algoritmo sequencial ou recursivo que assenta sobre dois importantes princípios [41]. O primeiro impõe que a imagem deve ser percorrida numa ordem pré-estabelecida, normalmente *Raster* (Figura 3.3.10.1) ou *Anti-Raster Order* (Figura 3.3.10.2). Contrariamente à implementação anterior, a ordem de pesquisa é essencial para o funcionamento do algoritmo.

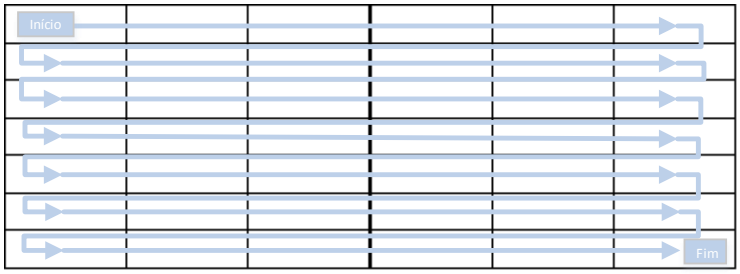


Figura 3.3.10.1 – Raster Order

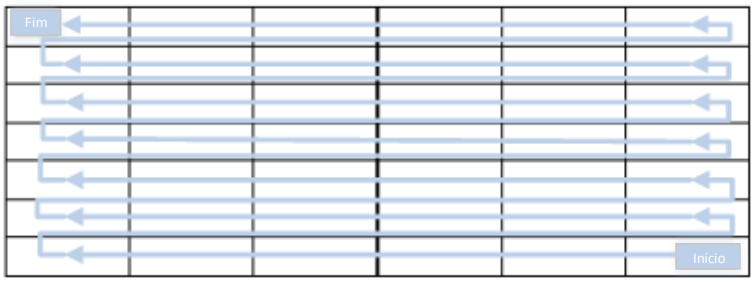


Figura 3.3.10.2 – Anti-Raster Order

O segundo, Figura 3.3.10.4, determina que o valor do pixel atual, depende do valor dos pixéis na sua vizinhança, compara com o valor da imagem a analisar (máscara) e escreve o valor diretamente na imagem que está a ser processada (marcador), para que o seu resultado possa ser tomado em conta na iteração seguinte. A vizinhança analisada depende da ordem da pesquisa, tendo como base a divisão da região *8-connected* em dois operadores, o elemento estruturador é composto pela metade superior (N_g^+) durante o *raster scan* e pela metade inferior (N_g^-) no decorrer do *anti-raster scan*. O número de iterações que o algoritmo necessita até atingir a estabilidade, é muito inferior ao apresentado anteriormente, no entanto, não é tão eficiente em alguns padrões. Os elementos estruturadores utilizados podem ser consultados na Figura 3.3.10.3.

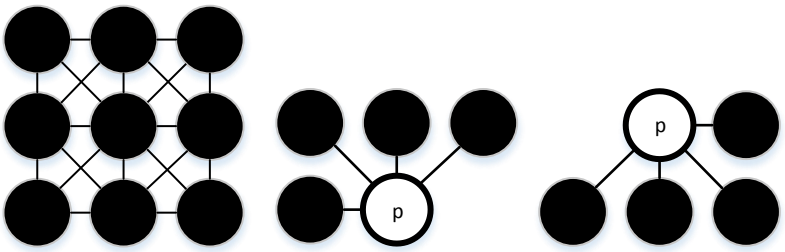


Figura 3.3.10.3 – Elemento estruturador 8-connected, N_g^+ e N_g^-

- I: Imagem máscara (binária ou em tons de cinzento)
- J: Imagem marcador, definida no domínio D_1, $J \leq I$.
  - A Reconstrução é determinada diretamente em J

- Repetir até atingir a estabilidade:
  - Percorrer D_1 em *Raster Order*:
    - Seja p o pixel atual;
    - $J(p) \leftarrow (\max \{J(q), q \in N_g^+(p) \cup \{p\}\})$
  - Percorrer D_1 em *Anti-Raster Order*:
    - Seja p o pixel atual;
    - $J(p) \leftarrow (\max \{J(q), q \in N_g^-(p) \cup \{p\}\})$

Figura 3.3.10.4 – Sequential Reconstruction Algorithm [41]

Com o avanço atual das GPGPU, no processamento paralelo e o elevado número de pixéis presentes numa imagem em alta definição, não se pode descartar, à partida, nenhuma das metodologias anteriores sem qualquer comparação recente, entre ambas as metodologias. No capítulo 3, desta dissertação, é feito um estudo mais aprofundado sobre ambas as implementações.

3.3.11.Binary Reconstruction using a queue of pixels

Um dos avanços propostos [41], para a implementação de um sistema de reconstrução morfológica mais eficiente, consiste em considerar apenas os pixéis cujos valores podem ser alterados. A primeira pesquisa tem como principal objetivo detetar os pixéis que dão início ao processo, normalmente situados nas margens da imagem. Esta pesquisa inicial possibilita que, a informação seja apenas propagada nas regiões de interesse da imagem. Os valores da pesquisa são guardados numa queue de pixéis, recorrendo a estruturas de dados como a FIFO, onde os pixéis são adicionados ao final da queue e consumidos do topo. Enquanto a queue não estiver vazia o algoritmo deve continuar. No caso de aplicação de reconstrução de imagens binárias, a implementação de um algoritmo com FIFO é simples. As margens da imagem marcador são adicionadas à queue e o seu valor é propagado ao longo dos componentes ligados da imagem máscara. O algoritmo (Figura 3.3.11.1) é extremamente eficiente, após a inicialização da *queue* de pixéis [41].

- I: Imagem máscara (binária)
- J: Imagem marcador (binária), definida no domínio D_1, J $\subseteq$ I.
  - A Reconstrução é determinada diretamente em J
- Inicialização da *queue* com os pixéis de contorno da imagem marcador:
  - Por cada pixel p $\in D_1$:
    - Se J(p) = 1 e $\exists q \in N_g(p)$, J(q) = 0 e I(p) = 1:
      - fifo_add(p)
- Propagação: Enquanto fifo_empty() = false
  - P $\leftarrow$ fifo_first()
  - Por cada q $\in N_g(p)$ (vizinhança de p):
    - Se J(q) = 0 e I(q) = 1
      - J(q) $\leftarrow$ 1
      - fifo_add(q)

Figura 3.3.11.1 – Binary Reconstruction using a queue of pixels [41]

Na literatura, em [46] , [48], entre outros, têm sido propostos diversos sistemas de *queues* prioritárias com o intuito de melhorar os resultados, ou seja o tempo de processamento, da reconstrução. A vantagem da utilização desta técnica em relação à anterior não é clara, dependendo do número de iterações, que o algoritmo necessita até atingir a estabilidade.

3.3.12.A Fast Grayscale Reconstruction Algorithm

Geralmente mais eficiente que as técnicas previamente propostas na literatura [41], na reconstrução de imagens em tons de cinzento, o algoritmo é atrasado pela determinação inicial dos máximos regionais da imagem marcador. Esta abordagem não é desenvolvida ao longo desta dissertação, pois a sua aplicação é indicada no processamento de imagens em tons de cinzento.

3.4. Sistema Proposto

O sistema é composto por quatro distintos módulos: pré-processamento, segmentação, identificação e seguimento. O trabalho realizado nesta dissertação foca-se, essencialmente na conceção de uma nova abordagem e implementação do processo de pré-processamento e segmentação de matrícula, sendo que a sua aplicabilidade na identificação (OCR) e seguimento foi apenas trabalhada ao nível funcional, recorrendo ao matlab. A Figura 3.4.1 representa o diagrama de blocos do sistema proposto, bem como os módulos opcionais e fora do âmbito do trabalho de dissertação.

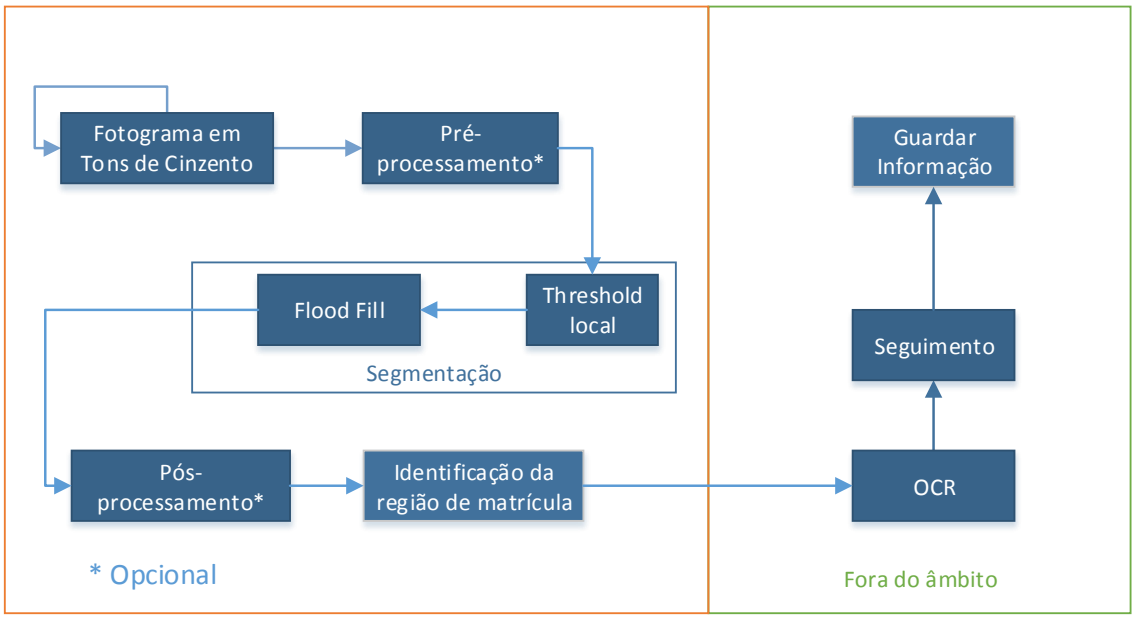


Figura 3.4.1 – Diagrama de blocos do sistema proposto

3.4.1. Pré-processamento

O módulo de pré-processamento recebe uma imagem em tons de cinzento, e apresenta como principal objetivo melhorar os resultados da segmentação, realçando as regiões de interesse na imagem. Pode também reduzir o tamanho da imagem a analisar, recorrendo ao *downscaling*, sem compressão, diminuindo o tempo de processamento dos blocos que o sucedem, guardando a imagem original inalterada em memória, para que o processo de OCR continue a ser executado sobre imagens de elevada resolução.

Este princípio é similar ao *Supersampling*, uma metodologia de *anti-aliasing* espacial utilizado para remover bordas irregulares e pixelizadas em imagens renderizadas. O problema de *aliasing* surge, pois contrariamente aos objetos presentes no mundo real que apresentam curvas e linhas suaves, as imagens digitais são compostas de pequenos quadrados chamados pixéis, todos do mesmo tamanho e cada um com a sua cor [49]. Quando um pixel fica entre dois (ou mais) valores de cor, a média entre eles é calculada. O *Supersampling* propõe que o *rendering* inicial da imagem seja realizada a uma resolução mais elevada, reduzindo-a posteriormente para o tamanho desejado. O resultado é uma imagem *down-sampled* com transições mais suaves entre os contornos dos objetos [49].

No sistema proposto é apresentada uma metodologia similar, onde para obter um bom resultado de OCR a imagem captada deve ser de resolução elevada, apresentando contornos entre objetos bem definidos. No entanto, o processo de segmentação e deteção da região da matrícula demonstra-se moroso, em imagens de alta resolução. O que o módulo de pré-processamento propõe é manter a imagem original, sem alterações, para que o módulo de OCR possa apresentar bons resultados, e uma imagem reduzida de forma a aliviar a carga de processamento do módulo de segmentação e deteção. Como descrito na metodologia de *Supersampling*, reduzir o tamanho de uma imagem de resolução superior apresentará contornos mais suaves entre objetos, contrariamente a uma imagem adquirida a resoluções inferiores. Diferentes técnicas de redimensionamento podem ser utilizadas, conforme o compromisso escolhido entre qualidade e tempo de processamento.

O algoritmo da vizinhança mais próxima (*Nearest Neighbor Algorithm)* é proposto para o redimensionamento da imagem, mantendo a definição de contornos. É uma técnica pouco intensiva computacionalmente, sendo a melhor opção em sistemas de tempo real [50] [51], no entanto, aumenta o *aliasing* (em especial em linhas e curvas diagonais). É por isso necessário atingir um compromisso, entre a redução de tamanho da imagem e a capacidade de deteção do sistema, que varia dependendo do ambiente que se está a analisar.

Ao diminuir o tamanho da imagem existe uma perda de informação relativa aos pixéis que foram postos de parte, pelo que o processo é irreversível e a imagem original deve ser guardada para efeitos de OCR. Neste caso, o algoritmo procura os pixéis mais indicados para descartar. O algoritmo apresenta um elevado grau de paralelismo, não depende do resultado do cálculo da

imagem ou pixéis anteriores e não recorre a *branch conditions* [50]. O rácio da dimensão pode ser calculado pelas Equação 3.4.1.1 e Equação 3.4.1.2. O algoritmo de redimensionamento pode ser expresso pelo fluxograma representado na Figura 3.4.1.1, onde W1 corresponde ao *Width* original.

$$x_{ratio} = \frac{Width_1}{Width_2}$$

Equação 3.4.1.1 – Rácio de compressão em X

$$Y_{ratio} = \frac{Height_1}{Height_2}$$

Equação 3.4.1.2 – Rácio de compressão em Y

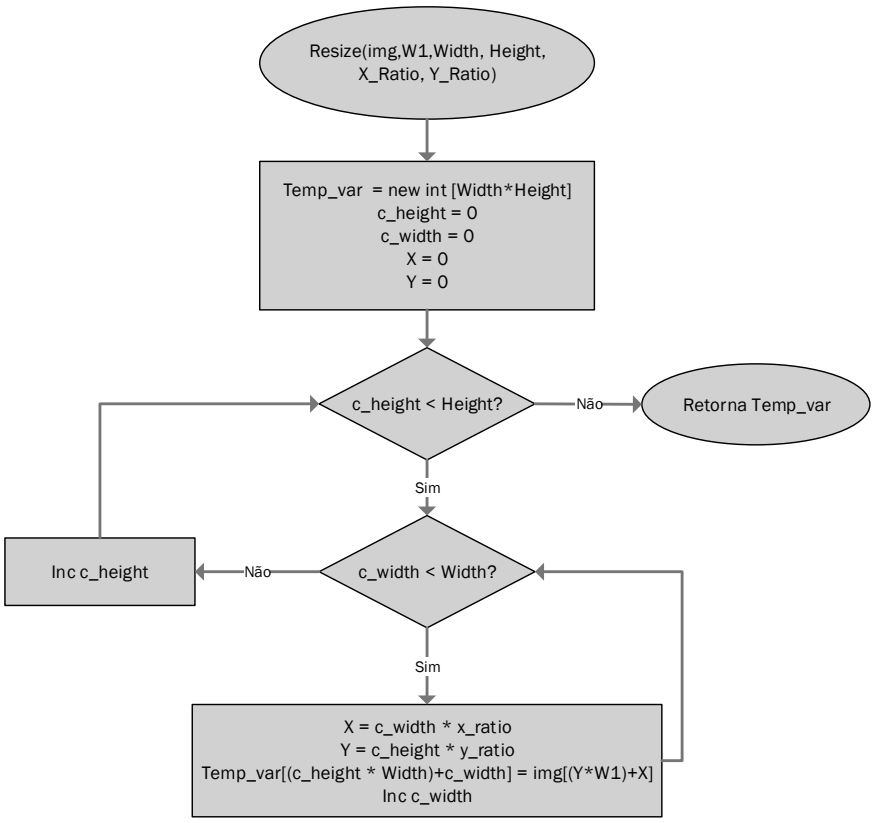


Figura 3.4.1.1 – Algoritmo de redução do tamanho da imagem, recorrendo aos valores da vizinhança mais próxima

3.4.2. Segmentação

O módulo está dividido em duas partes distintas, primeiro o processo de *Threshold* local, que cria uma imagem binária, e posteriormente a reconstrução morfológica, que preenche os buracos presentes na imagem binária. O intuito do processo de segmentação é isolar possíveis regiões da matrícula. Quanto mais preciso for o algoritmo, maior será a carga computacional exercida sobre o sistema, pelo que um compromisso deve ser estabelecido sobre a taxa de sucesso na deteção e o tempo de execução do algoritmo.

O módulo de *Threshold* local é utilizado com o intuito de diminuir o nível de detalhe presente no fotograma, cria uma imagem binária, enquanto realça o contraste entre as letras e a chapa da matrícula. Esta metodologia de *Threshold* adaptativo foi escolhida, pois num ambiente rodoviário existem várias fontes independentes de iluminação (como candeeiros, luz natural, iluminação dos veículos, etc.). A análise por regiões dos valores de intensidade demonstra-se então mais eficaz.

As abordagens que procuram apenas um valor global de *Threshold* para as imagens, apesar de menos intensivas computacionalmente, não obtêm bons resultados, pois mesmo que consigam realçar uma região da matrícula, dificilmente conseguem identificar mais. A própria posição dos veículos na imagem afeta a iluminação que incide sobre a matrícula, tornando o seguimento, mesmo que apenas de um veículo, uma tarefa complexa.

As metodologias de deteção de contornos, em especial quando o gradiente vertical é utilizado, apresentam bons resultados na literatura. No entanto, a proximidade entre o veículo e a câmara de videovigilância e a inclinação são bastante restritos. Em aplicações de seguimento de veículos, é importante haver algum distanciamento entre a câmara e o veículo. O algoritmo deve ser capaz de identificar a região da matrícula no maior número de fotogramas possíveis, tornando as abordagens que recorrem a detetores de contornos menos eficazes, pois fazem sobressair outros elementos do *background* da imagem (como passeios, paredes, candeeiros, etc.) sobre a região da matrícula.

Figura 3.4.2.1 – Imagem em tons de cinzento

Figura 3.4.2.2 – Deteção de contornos Sobel

O algoritmo de *Threshold* local proposto difere da sua definição em [37], na medida em que o objetivo é comparar o pixel atual com a média da região, contrariamente à comparação do valor máximo da região com a sua média, descrita pela literatura. Como é possível observar na Figura 3.4.2.3, a região da matrícula não fica devidamente evidenciada e a qualidade dos resultados piora, com o aumento do tamanho da janela. Quando o valor máximo da região é utilizado, não é possível identificar os pixéis nos quais ocorre uma mudança acentuada de luminosidade. Contrariamente, ao utilizar o valor do pixel atual, é possível estabelecer se o seu valor se encontra acima ou abaixo, da média da região, permitindo assim realçar o contraste entre os caracteres e a chapa da matrícula, quando o tamanho da janela utilizado é adequado. É introduzida uma nova variável (*smoothing factor*), com o intuito de suavizar alguns dos detalhes da imagem (pontos de menor intensidade) e algum ruido, nomeadamente o *pepper noise*. A equação proposta para o cálculo do *Threshold* local pode ser definida pela Equação 3.2.1.

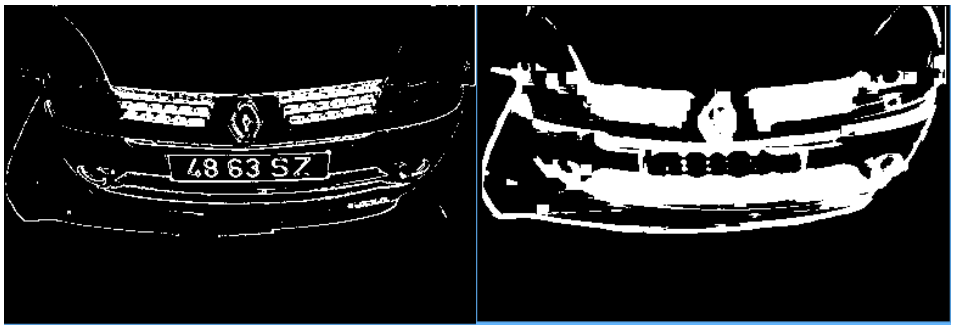


Figura 3.4.2.3 – Threshold local, utilizando o valor máximo da região no cálculo.

$$\text{Binary}_{frame} = mean_{frame} - (Grayscale_{frame} - Smoothing_{factor})$$

Equação 3.4.2.1 – Formula proposta para o calculo do valor do Threshold local

O tamanho da janela (*Wsize*), é um parâmetro importante na calibração do algoritmo, corresponde ao tamanho da vizinhança a analisar e é centrada no pixel a processar. O seu tamanho está relacionado com as dimensões que objeto a segmentar apresenta em proporção à imagem. Com o aumento do *Wsize*, também aumenta a carga computacional que o algoritmo exerce sobre o sistema, pelo que um compromisso deve ser alcançado, entre a qualidade dos resultados da segmentação e o tempo de processamento. De uma forma geral, uma *Wsize* de tamanho 9 demonstrou ser a mais indicada ao longo dos testes realizados.

Figura 3.4.2.4 – Imagem em tons de cinzento e Imagem Threshold (Wsize 3)

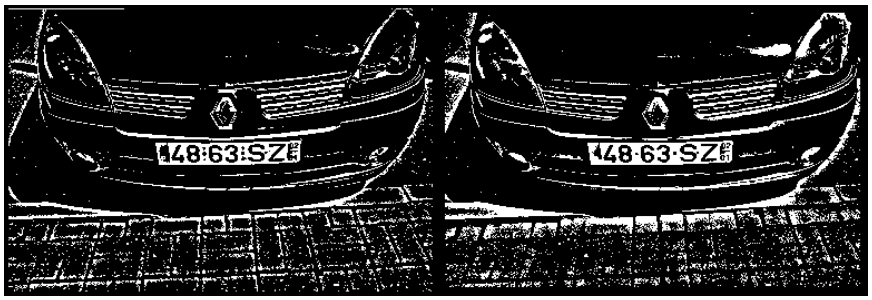


Figura 3.4.2.5 – Imagem Threshold (Wsize 9) e Imagem Threshold (Wsize 17)

O *smoothing factor* representa outro dos parâmetros configuráveis no sistema. No entanto, a sua calibração não apresenta grande impacto no resultado esperado, uma vez que nos testes realizados um fator de 20 se demonstrou adequado a praticamente todas as imagens analisadas. Na Figura 3.4.2.5 é utilizado um valor de *smoothing factor* de 20, na Figura 3.4.2.6 são utilizados os valores 5 e 30, respetivamente.

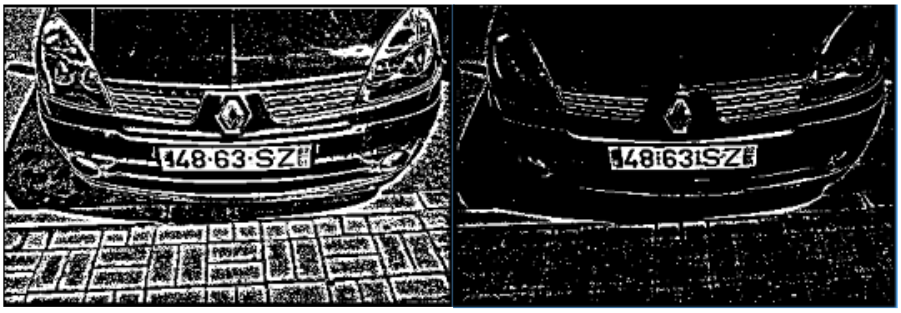


Figura 3.4.2.6 – Smoothing Factor 5 e 30, WSize 9

O algoritmo de *Threshold* local pode ser expresso em dois blocos de processamento distintos, o primeiro percorre a imagem, atribui o valor zero às margens e o valor zero ou um, dependendo do resultado da equação nos restantes pixéis. Assim, o fluxograma de *Threshold* local é representado na Figura 3.4.2.7. A Equação 3.4.2.2 determina se o pixel atual (*pos*) pertence às margens da imagem ou se é necessário calcular a média dos valores na sua vizinhança.

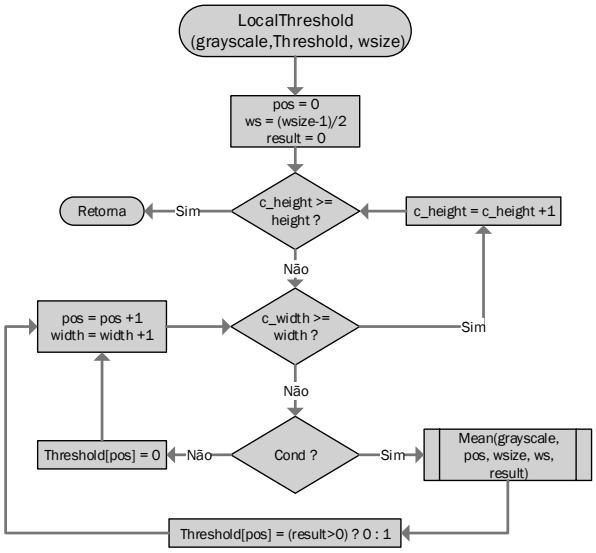


Figura 3.4.2.7 – Fluxograma Threshold Local

Cond = ((c_width < ws) || (c_width > width – ws) || (c_height < ws*width) || (c_height > size - (ws*width)))

Equação 3.4.2.2 – Condição que determina se o pixel pertence ou não às margens da imagem

Se o pixel não pertence às margens da imagem, é necessário calcular a média da intensidade dos valores da sua vizinhança, recorrendo à função *Mean* e indicando a imagem (*grayscale*) a analisar, a posição atual (*pos*), o tamanho da vizinhança (*Wsize*) e a variável que vai receber o resultado da operação (*result*). O Fluxograma da Figura 3.4.2.8 representa o processo.

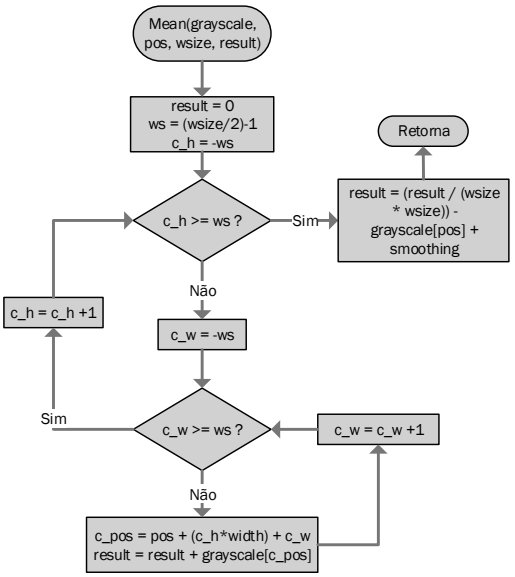


Figura 3.4.2.8 – Fluxograma do cálculo da média da intensidade numa vizinhança

O módulo de reconstrução morfológica, apresenta como intuito o preenchimento de buracos presentes na imagem binária. A técnica de *Threshold* local, quando bem calibrada, apresenta uma separação evidente entre os caracteres e a chapa da matrícula, isolando os caracteres das margens, permitindo assim aplicar a metodologia de *Fill Hole* à imagem. O algoritmo de reconstrução morfológica necessita de duas imagens, o marcador e a máscara. Neste caso a imagem máscara corresponde ao inverso do resultado do *Threshold* local (Figura 3.4.2.9) e a imagem marcador é uma imagem com todos os valores a zero (preto) e as margens a um (branco).

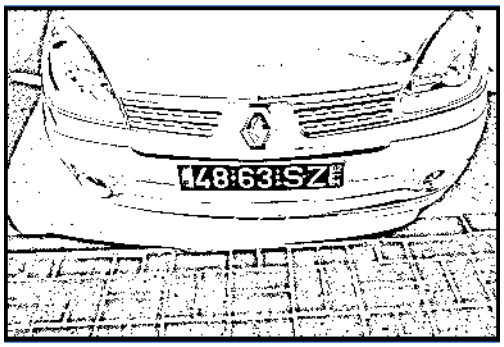


Figura 3.4.2.9 – Imagem máscara

A imagem marcador começa a ser inundada até que o algoritmo esteja terminado. O objetivo da metodologia é preencher todos os pixéis ligados às margens da imagem, ficando os buracos isolados. Através da subtração desta nova imagem (Figura 3.4.2.10) pela imagem máscara (Figura 3.4.2.9) é possível obter os buracos da imagem binária, que neste caso correspondem aos caracteres da matrícula.

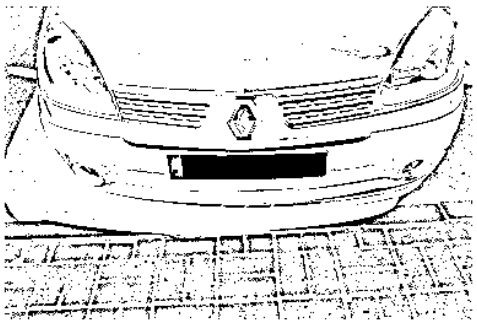


Figura 3.4.2.10 - Imagem preenchida

Outra variável configurável do sistema é o número de iterações, do algoritmo de reconstrução morfológica. Dependendo do tipo e cenário de aplicação, pode não ser necessário atingir a estabilidade para obter uma boa segmentação de matrícula. Um compromisso entre a qualidade de resultados e o tempo de processamento do sistema volta a ser estabelecido neste parâmetro. Na Figura 3.4.2.11 são apresentados, os resultados de sucessivas reconstruções morfológicas até atingir a estabilidade.

Figura 3.4.2.11 – Fill Hole: uma iteração e estabilidade

Como o algoritmo de reconstrução morfológica processa imagens binárias, não necessita de várias iterações, geralmente a primeira já apresenta um resultado suficientemente bom para o processo de segmentação, e a segunda atinge a estabilidade. No entanto, em ambientes mais movimentados, com mais objetos presentes na imagem podem ser necessárias mais iterações.

Como referido anteriormente, o módulo de *Threshold* local apresenta como uma das suas principais funções, realçar o contraste entre os caracteres e a chapa da matrícula, funcionando bem em conjunção com o algoritmo de *Fill Hole* pois, contrariamente aos algoritmos de deteção de contornos, não destaca outros contornos ou formas geométricas, de forma tão demarcada. Como é possível confirmar pelas imagens representadas na Figura 3.4.2.12, a abordagem realça mais os contornos do veículo e não permite isolar os caracteres.

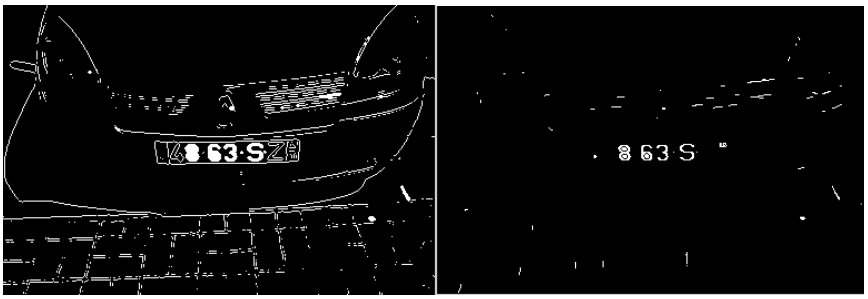


Figura 3.4.2.12 – Flood Fill aplicado sobre Deteção de contornos

3.4.3. Pós-Processamento

Um módulo adicional pode ser utilizado no sistema, entre o processo de segmentação e deteção. A sua principal função é "limpar" a imagem, de modo a que o processo de identificação seja mais robusto. Apresenta um custo considerável no desempenho do sistema, pelo que não é considerado como parte integrante do sistema, ou seja, o seu uso e implementação são opcionais. No entanto, é importante referir a sua utilização uma vez que pode ser importante em algumas aplicações, dependendo do cenário e circunstâncias em que se encontra a câmara de videovigilância.

O algoritmo proposto tem como objetivo, eliminar pequenos objetos da imagem binária, uma vez que, o módulo de *Flood Fill* pode preencher pequenos pontos ou objetos na imagem, que podem condicionar os resultados da deteção. Como se pode verificar, na Figura 3.4.3.1, o resultado da operação de reconstrução morfológica deixa pequenos objetos, que podem interferir na taxa de deteção do algoritmo. Na Figura 3.4.3.2 são removidos todos os objetos de área reduzida, destacando outra vez os caracteres da matrícula, em relação ao resto da imagem. Durante o processo de investigação foi utilizada a função "bwareaopen" do matlab para comprovar o conceito.

Figura 3.4.3.1 – Imagem em tons de cinzento e respetivo Flood Fill

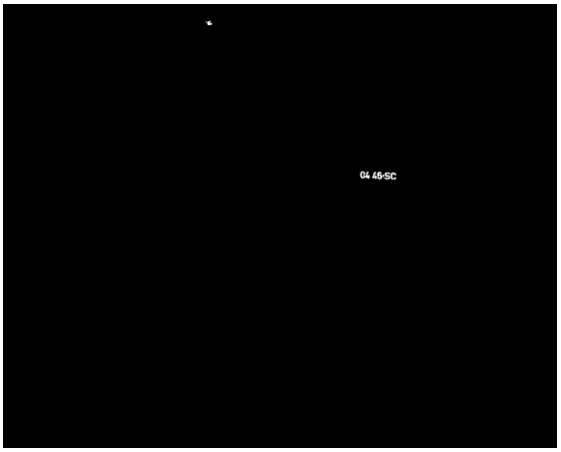


Figura 3.4.3.2 – Pequenos objetos removidos da imagem Flood Fill

3.4.4. Deteção

O módulo de deteção procura zonas de grande concentração de intensidade na imagem segmentada. Recorrendo a histogramas verticais e horizontais é possível estudar as dimensões da possível região da matrícula, no entanto, é necessário suavizar os resultados dos histogramas. Recorrendo a um *Moving Average Filter* (Equação 3.4.4.1) é possível representar a região da matrícula como um todo e não em caracteres separados. O filtro é ser descrito pela Equação 3.4.4.1.

$$y(n) = \frac{1}{Size}\ (x(n) + \ x(n-1) + \cdots + x(n - Size - 1))$$

Equação 3.4.4.1 – Moving Average Filter

A suavização dos resultados (Figura 3.4.4.2), permite identificar a região da matrícula como um todo, uma vez que a segmentação de caracteres apresenta alguma descontinuidade nos valores de intensidade.

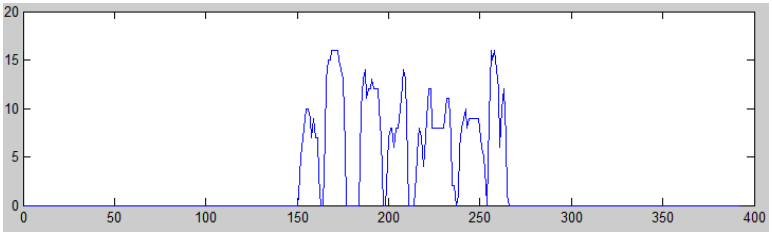


Figura 3.4.4.1 – Histograma vertical da Figura 3.4.2.11

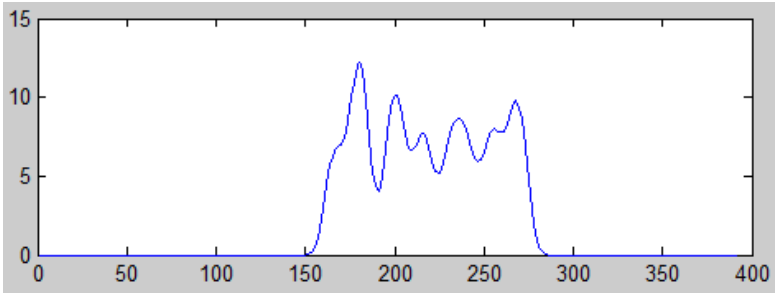


Figura 3.4.4.2 – Histograma vertical da Figura 3.4.2.11, suavizado

No caso da Figura 3.4.2.11, como esta apenas apresenta os caracteres da matrícula após a segmentação de imagem, o resultado do histograma vertical não suavizado (Figura 3.4.4.1), é igual ao resultado da segmentação de caracteres. Nas matrículas portuguesas o histograma deve apresentar dois picos e um espaço, ou um pico maior e um espaço, que representa a junção de picos, podendo identificar a matrícula através do padrão. A Figura 3.4.4.3 demonstra 6 picos com intervalos entre eles, sendo que o 7° corresponde ao ano e mês do veículo, que dependendo da proporção da matrícula em relação à imagem pode também ser representado.

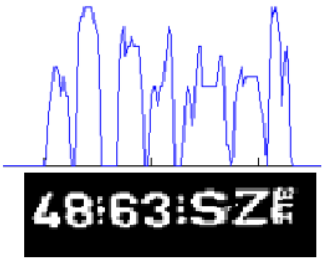


Figura 3.4.4.3 – Relação entre histograma e segmentação de caracteres da matrícula

A metodologia de deteção e suavização apresenta características indicadas para o processamento paralelo, demonstrando-se como fortes candidatas para aceleração em *hardware*, uma vez que vários pixéis podem ser processados em simultâneo.

3.5. Aceleração em *Hardware*

Nos dias que correm o processador (CPU), por si só, não consegue responder aos requisitos impostos pelas aplicações mais intensivas de processamento de imagem, devido à carga computacional exigida.

A solução é recorrer à aceleração por *hardware*, como GPU e FPGA, no entanto, os algoritmos apresentam desempenho diferente, dependendo da arquitetura do acelerador. As FPGAs demonstram um desempenho elevado, na aceleração de algoritmos de processamento de imagem, apesar da sua frequência de operação não ser elevada, através do paralelismo da arquitetura. As GPUs possuem um número muito elevado de núcleos, demonstrando um potencial de alto desempenho em diversas aplicações.

A performance elevada do FPGA advém da sua flexibilidade, que permite criar e otimizar o circuito ideal para cada aplicação, bem como número elevado de bancos de memória *on-chip* que ajudam no paralelismo de acesso à memória [52]. Devido à baixa frequência de operação, é necessário realizar o maior número de operações possíveis, num ciclo de relógio, e limitar as operações de acesso à memória. É por isso vantajoso trabalhar com imagens binárias, que ocupam menos espaço e diminuem o *bottleneck* no acesso à memória. Os sistemas de ANPR, de

uma forma geral, apresentam melhores resultados quando a deteção é feita em imagens binárias, sejam elas o resultado da deteção de contornos ou do *threshold,* pelo que a metodologia proposta nesta dissertação é uma boa candidata para aceleração em FPGA.

Originalmente, a arquitetura dos GPUs era apenas vocacionada para o processamento gráfico. No entanto, a partir do ano 2000 com o aparecimento de *shaders* programáveis e suporte para a realização de cálculos com vírgula flutuante [53], o conceito de GPGPU foi introduzido. Problemas que envolviam o cálculo de matrizes ou vetores de uma até quatro dimensões eram facilmente traduzidas para o GPU. O aparecimento de *frameworks* como o CUDA da *Nvida* e o OpenCL do *Khronos group,* permitem que o programador ignore os conceitos gráficos subadjacentes e se foque na computação de alto desempenho.

As GPUs suportam também um número elevado de núcleos (1664 CUDA *cores* [54]*,* no caso da placa usada na realização desta dissertação) que correm em paralelo. No entanto os diversos núcleos estão agrupados e a transferência de dados entre grupos é bastante limitada [52]. A frequência base de operação é pelo menos duas a cinco vezes superior ao de uma FPGA e a largura de banda da memória das GPUs é muito superior à do CPUs, apresentando valores na ordem dos 224 GB/s. O inerente paralelismo das aplicações de processamento de imagem, torna o GPU um bom candidato para a aceleração, sendo possível mapear cada pixel diretamente a uma GPU *thread*.

O objetivo desta dissertação é o desenvolvimento, implementação e aceleração de um algoritmo de segmentação de matrícula. Como tal a *framework* CUDA é uma ferramenta ideal no desenvolvimento inicial da metodologia proposta, uma vez que o CUDA C é uma extensão ao ANSI-C. O suporte para a interoperabilidade entre CUDA e as APIs gráficas OpenGL/DirectX facilitam o desenvolvimento e depuração de aplicações de processamento de imagem.

Como apresentado no próximo capítulo, o algoritmo de reconstrução morfológica apresenta alguns conceitos interessantes e a sua implementação em GPU é comparada com a sua vertente em CPU. Já os algoritmos de pré-processamento e de *Threshold* local, devido ao paralelismo inerente, demonstram ser claramente superiores em GPU. O desenvolvimento em CUDA, possibilita a gestão de recursos e processamento entre GPU e CPU (tendo em consideração

o tempo de transmissão de dados, em sistemas com GPUs discretas), não limitando o *designer* à partida.

Após a implementação da metodologia em CUDA, é necessário identificar o bloco crítico do sistema, através de ferramentas de *profiling*. O capítulo 5 propõe uma implementação, do algoritmo de reconstrução morfológica em FPGA, e estuda a sua viabilidade face ao algoritmo proposto no capítulo anterior.

3.5.1. Caso de teste, deteção de vários veículos

Um caso de teste importante na conceção da metodologia foi a capacidade de deteção de diversos veículos numa imagem. Como descrito no capítulo anterior, as metodologias utilizadas em sistemas ANPR, geralmente apresentam uma grande proximidade entre a matrícula a segmentar e a camara de videovigilância. Outras alternativas utilizam janelas de pesquisa que procuram localizar mais que um veículo em posições predeterminadas. Na Figura 3.5.1.1 é possível visualizar a imagem utilizada como caso de teste, em tons de cinzento.

Figura 3.5.1.1 – Caso de teste, tons de cinzento

O matlab foi utilizado, na segmentação da Figura 3.5.1.1, recorrendo de todos os módulos propostos neste capítulo, incluindo o de pós-processamento, que se demonstrou essencial para o resultado de segmentação apresentado na Figura 3.5.1.2.

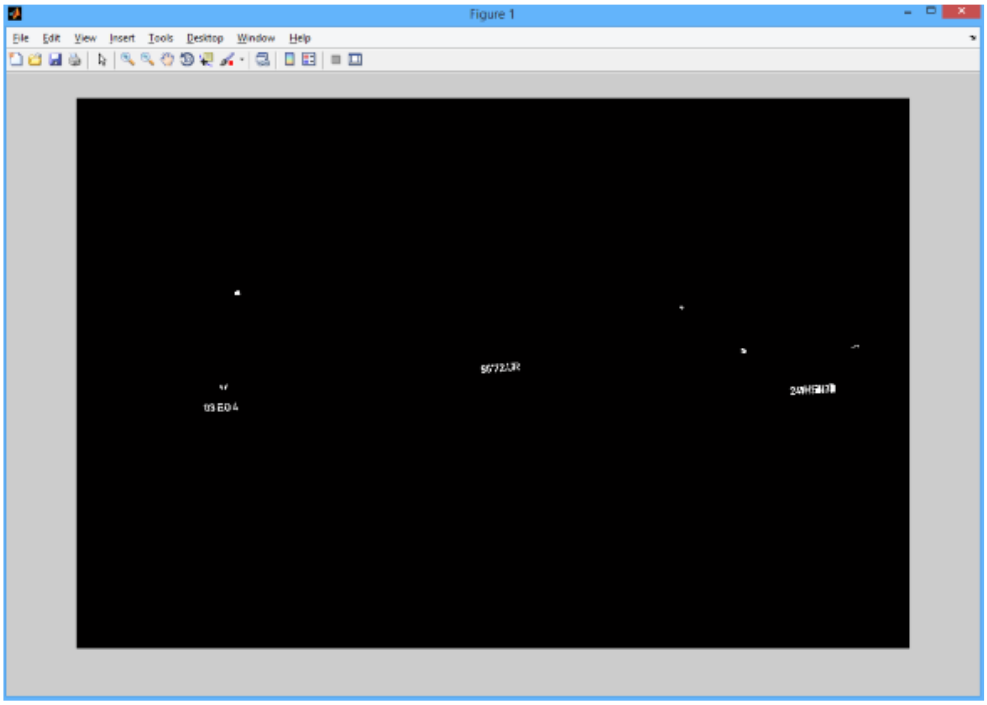


Figura 3.5.1.2 - Caso de teste, resultado do processo de segmentação

A análise detalhada da Figura 3.5.1.2 revela a presença de alguns objetos que não pertencem à região da matrícula. No entanto, os caracteres estão bem evidenciados em comparação. O Histograma vertical presente na Figura 3.5.1.3 apresenta três áreas de maior intensidade e dimensão, pelo que recorrendo a um algoritmo de seleção é possível obter apenas as áreas referentes às regiões da matrícula. O último caractere do primeiro carro, não aparece na imagem segmentada, por consequência do algoritmo de redimensionamento.

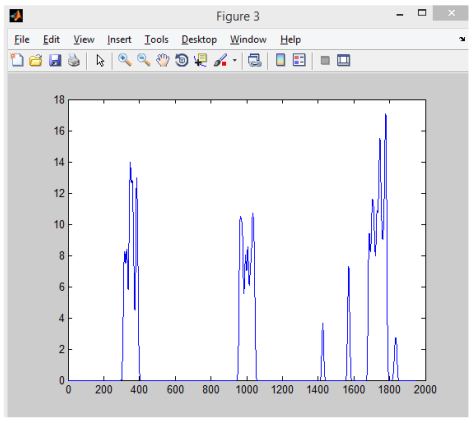


Figura 3.5.1.3 – Caso de teste, Histograma vertical do resultado da segmentação

Finalmente, após o processo de seleção é possível verificar o resultado da segmentação das regiões da matrícula, na Figura 3.5.1.4. A deteção de diversas regiões da matrícula à distância, é essencial para o sucesso de um sistema de seguimento de veículos. A imagem original foi capturada com uma câmara Nikon com 3900x2613 de resolução, sem compressão, que foi reduzida para 1948x1298, de forma a acelerar o processamento do módulo de segmentação.

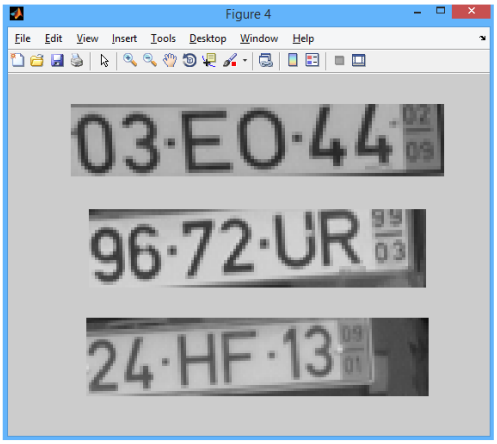


Figura 3.5.1.4 – Caso de teste, resultado do processo de seleção

3.6. Discussão

A metodologia proposta apresenta bons resultados, na segmentação da região da matrícula de veículos a distâncias variadas. Os diferentes contrastes de cor, mais comuns, entre os caracteres da matrícula e a chapa não inviabilizam a metodologia.

Relativamente, às metodologias tradicionalmente propostas na literatura, o sistema proposto apresenta uma capacidade de deteção de diversos veículos, em diversas posições e distâncias na imagem e com *backgrounds* mais complexos. A expansão da área de deteção e o aumento da resolução da imagem permitem a implementação, de um sistema de seguimento de veículos mais robusto.

No entanto, o sistema necessita de alguma calibração, que depende de diversos fatores, como a resolução da imagem, a redução do tamanho da imagem, o tamanho da janela da vizinhança, o *smoothing factor* e o número de iterações do algoritmo de Reconstrução Morfológica. Uma boa calibração do sistema é essencial para o sucesso do módulo de segmentação.

4. CUDA

4.1. Introdução

"Parallel programming is always motivated by performance and driven by profiling." [55]

Nos últimos anos o GPU sofreu uma evolução notória, passando de um dispositivo especializado, utilizado apenas no *rendering* de imagem, para uma tecnologia essencial na realização de tarefas de processamento intensivo. A sua utilização em conjunto com o CPU, permite acelerar uma ampla gama de aplicações de computação, denominada por *heterogeneous computing*. Esta emergência das arquiteturas heterogéneas CPU-GPU, levaram a uma mudança no paradigma da programação paralela. Nesta sua nova aplicação, as GPUs potenciaram novos avanços científicos e tecnológicos e o aparecimento de novas metodologias, que se tornaram possíveis devido à capacidade das GPUs de realizarem processamento em paralelo, nos diversos núcleos apresentados pela sua arquitetura, mantendo um baixo impacto energético.

Neste capítulo é proposta uma aceleração do sistema anteriormente descrito, numa plataforma heterogénea, composta por CPU e GPU, recorrendo à *framework* CUDA. Posteriormente, é estudada a viabilidade da implementação do algoritmo de reconstrução morfológica em GPU e CPU. De seguida, são levantados os resultados obtidos do *profiling* do sistema, recorrendo às ferramentas *Nsight* (GPU) e *gprof* (CPU). O capítulo termina com uma discussão dos resultados obtidos relativamente à aceleração do algoritmo.

4.2. GPGPU

General-purpose GPU recorre à placa gráfica, que normalmente é utilizada no *rendering* de imagens, para realizar tarefas normalmente executadas pelo CPU, tais como: processamento de imagem, visão por computador, inteligência artificial, cálculo numérico, entre outras [56]. Contrariamente à arquitetura do CPU, que está otimizada para obter alto desempenho em código sequencial, os GPU estão otimizados para executar tarefas altamente paralelas [57].

O GPU sempre dispôs de amplos recursos computacionais, tendo evoluído recentemente de um *fixed-function special purpose processor* para um *full-fledged parallel programmable processor*, permitindo assim, a aceleração de algoritmos altamente complexos e computacionalmente intensivos [58]. O OpenCL é, neste momento, a principal *framework open source* para a programação de GPUs, enquanto o CUDA (*Nvidia*) domina o mercado proprietário [56].

4.3. Sistemas Heterogéneos

Numa arquitetura heterogénea composta por um CPU, com vários núcleos, e GPU, a placa gráfica não é uma plataforma independente, mas sim um coprocessador do CPU. O GPU é indicado para acelerar tarefas computacionalmente intensivas com paralelismo de dados, deixando as tarefas de escalonamento, E/S de ficheiros e rede ao cargo do processador [59].

Existem duas arquiteturas possíveis em sistemas heterogéneos de CPU-GPU: arquitetura discreta ou arquitetura integrada. Na arquitetura discreta, o CPU e o GPU são dois componentes independentes e comunicam através do barramento *PCIExpress*. Nesta arquitetura o GPU é considerado discreto, a memória não é partilhada, sendo então necessário contabilizar a latência na transferência de dados. Esta será a abordagem mais comum neste tipo de sistemas. Na arquitetura integrada, o GPU e o CPU partilham o mesmo chip e apresentam uma arquitetura de memória unificada, possibilitando novas aplicações que poderiam ser inviabilizadas pela latência na transferência de dados, como descrito por [59] e [55]. A sua utilização será mais comum em sistemas embebidos.

Uma aplicação heterogénea é composta por duas partes: o código do *Host,* que é executado no CPU, e o código do *Device* que é executado no GPU [55]. Normalmente a aplicação é inicializada pelo CPU, que também é responsável por gerir o ambiente, o código e a informação destinada ao *device*, antes dele executar as tarefas de computação intensiva. Este tipo de aplicações permitem que o programa equilibre a carga de trabalho entre o CPU e o GPU, possibilitando assim, cobrir potenciais diferenças de recursos entre as capacidades de diversas plataformas, de forma autónoma [59].

A capacidade de computação de uma GPU pode ser descrita por duas características: o número de núcleos CUDA e o tamanho da memória [55]. Assim existem duas métricas que permitem descrever o desempenho do GPU, o *Peak computational performance,* que representa o número de cálculos de precisão dupla, ou simples, que o GPU consegue processar por segundo, e a *Memory Bandwidth* que é calculada através da proporção entre a velocidade de escrita e leitura na memória.

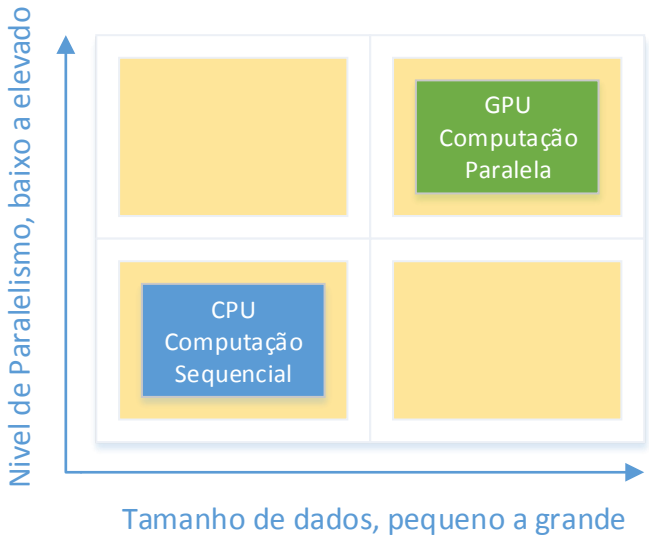


Figura 4.3.1 – Vantagens do processamento em CPU e GPU

Tal como ilustra a Figura 4.3.1, o nível de paralelismo e o tamanho da informação a processar, são dois indicadores que ajudam na decisão sobre qual plataforma utilizar. Se um problema apresentar um tamanho de dados reduzidos, lógica de controlo sofisticada ou um baixo nível de paralelismo, o CPU será a escolha indicada devido à sua capacidade de lidar com lógica

complexa e paralelismo ao nível da instrução. No entanto, se o problema em questão necessitar de processar um número elevado de informação e demonstrar um elevado paralelismo de dados, a escolha indicada será o GPU, devido ao grande número de núcleos programáveis e uma largura de banda de memória superior ao CPU.

4.4. CUDA C

O CUDA apresenta-se como uma das *frameworks* mais populares na aceleração de código no GPU, denominado por *kernel*. Assim, o CUDA C apresenta-se como uma extensão ao ANSI C *standard*, ao adicionar expressões que possibilitam a programação heterogénea, permitindo uma escalabilidade transparente do paralelismo apresentado pelas aplicações, em relação ao GPU e o seu número de núcleos.

A principal diferença entre programação paralela em C e em CUDA C, é o facto da arquitetura CUDA fornecer acesso direto ao seu *memory* e *execution model* [55], permitindo um maior controlo sobre o ambiente massivamente paralelo do GPU.

4.4.1. CUDA *Programming Model*

O CUDA *Programming Model* proporciona uma camada de abstração em relação à arquitetura computacional, atuando como a ponte entre a aplicação e a sua implementação no *hardware* disponível. Assim, é possível a execução de uma aplicação em sistemas de computação heterogéneos, através de um pequeno conjunto de extensões adicionadas ao ANSI C [55].

O *kernel* é uma parte integrante do CUDA *Programming Model* e representa o código destinado a correr no GPU. Do ponto de vista do programador, o *kernel* é expresso como um programa sequencial, ficando o CUDA encarregue de gerir o escalonamento pelos GPU *threads*. O mapeamento do algoritmo ao *device* é definido no *host*, pelo programador, o qual depende dos dados da aplicação e da capacidade do GPU.

Em grande parte das aplicações, o *host* opera independentemente do *device* [55]. Quando um *kernel* é iniciado, o controlo é devolvido ao *host*, libertando o CPU para executar tarefas adicionais que possam ser complementadas pela informação a ser processada no *device,* em paralelo.

4.4.2. Estrutura de aplicação CUDA

A estrutura típica de uma aplicação CUDA assenta em cinco tópicos principais. O primeiro passo, é um processo moroso que passa pela reserva de memória no GPU. Em aplicações de processamento de imagem, a alocação prévia do espaço necessário no início da aplicação será vantajosa para o desempenho da aplicação. Posteriormente, é necessário copiar a informação a processar da memória do CPU para a memória do GPU, invocando de seguida o *kernel*, que realiza o processamento dos dados de forma assíncrona. Após a conclusão do processamento dos dados no GPU, os mesmos são copiados de volta para o CPU, para serem analisados. Quando a aplicação não precisar de processar mais dados no *device,* o *host*, deve libertar o espaço reservado em memória.

Na Tabela 4.4.2.1 são estabelecidas as semelhanças entre as funções para alocar memória utilizadas em ANSI C e CUDA C. De todas as funções apresentadas, a função *cudaMemcpy*, difere mais da sua componente em C, necessitando da direção pela qual é executada a cópia. De seguida, é apresentada a implementação de uma aplicação para converter um ficheiro BMP, no espaço de cores RGB para tons de cinzento, recorrendo à aceleração em CUDA.

Standard C Functions	*CUDA C Functions*
malloc	*cudaMalloc*
memcpy	*cudaMemcpy*
memset	*cudaMemset*
free	*cudaFree*

Tabela 4.4.2.1 – Funções de gestão de memória em ANSI C e CUDA C

4.4.3. *CUDA RGB para tons de cinzento*

Como se pode ver na Figura 4.4.3.1, a aplicação começa pelo código do *host*, que lê um ficheiro no formato BMP e prepara o *device* para a conversão para tons de cinzento. No passo seguinte, é reservada a memória para receber o resultado da conversão e chamada a função *gpu_gray,* que prepara a *framework* CUDA. Quando o *kernel* concluir e o resultado da operação for copiado para o apontador *Grayscale*, a aplicação cria um ficheiro BMP, que possibilita a depuração do algoritmo, liberta a memória reservada e termina a execução.

```
void IMGP(void)
{
	bool img_check = LoadBitmap("Images/001.bmp");
	if (!img_check)
	{
		cout << "Image failed to load!" << endl;
	}
	else{
		cout << "Image loaded!" << endl;
		//cuda startup
		cuda_start();
		//grayscale
		GrayScale = (int *)malloc(Height*Width*sizeof(int));
		gpu_gray(Bitmap, GrayScale, Height, Width);
		CreateBMP("01 - grayscale", GrayScale, Width*Height, false, true);
		//Free Memory
		cuda_end();
		free(Bitmap);
		free(GrayScale);
	}
}
```

Figura 4.4.3.1 – Rotina utilizada para a conversão do espaço de cores RGB para tons de cinzento

Posteriormente, tal como ilustra a Figura 4.4.3.2 prepara-se o *device*, reservando o espaço necessário no GPU para processar os dados, recorrendo à função *cudaMalloc*. Na literatura, é considerada boa prática, identificar as variáveis de *device* por *d_nome*. No passo seguinte, são copiados os dados a processar (a imagem RGB) para a memória do GPU, utilizando a função *cudaMemcpy* e a direção *cudaMemcpyHostToDevice*.

```
void gpu_gray(int *bmp, int *gray, int height, int width)
{
        int *d_bmp, *d_gray;
        //GPU Memory allocation
        cudaMalloc(&d_bmp, 3 * height * width * sizeof(int));
        cudaMalloc(&d_gray, height * width * sizeof(int));
        //CPU -> GPU memory copy
        cudaMemcpy(d_bmp, bmp, 3 * height * width * sizeof(int), cudaMemcpyHostToDevice);
        //Invoke Kernel
        kernel_grayscale <<< height, width >>> (d_bmp, d_gray);
        //GPU -> CPU memory copy
        cudaMemcpy(gray, d_gray, height * width * sizeof(int), cudaMemcpyDeviceToHost);
        //Release GPU allocation
        cudaFree(d_bmp);
        cudaFree(d_gray);
}
```

Figura 4.4.3.2 – Rotina que gere a memória de GPU, cópia de dados e invoca o kernel

Quando a cópia termina é necessário invocar o *kernel* para processar a informação, e definir o mapeamento da imagem nos blocos e *threads* CUDA. A Figura 4.4.3.3, demonstra o sistema de blocos e *threads* da arquitetura CUDA, onde cada *device* possuí vários blocos de processamento, e cada bloco possuí uma ou mais *threads* que podem comunicar entre si através do acesso a uma memória partilhada (*Shared Memory*) de baixa latência. A memória global é partilhada por todas as *threads* e blocos, pelo que o acesso será mais moroso, no entanto apresenta uma capacidade de armazenamento muito superior.

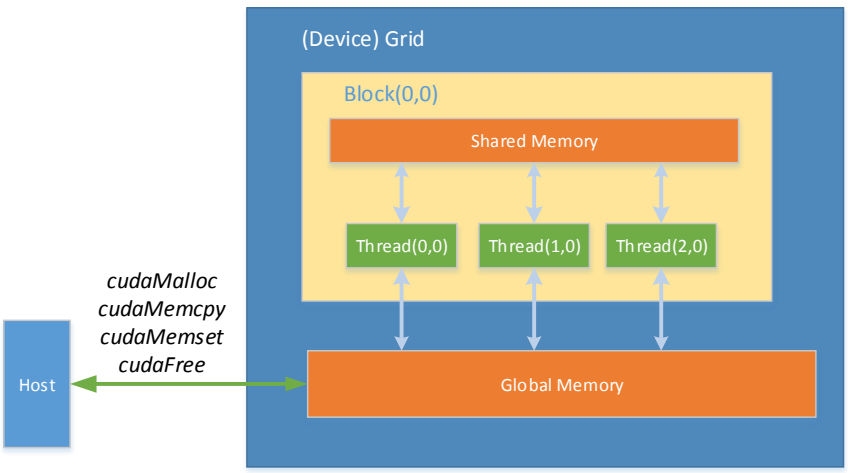


Figura 4.4.3.3 – Arquitetura CUDA

A Equação 4.4.3.1 apresenta a sintaxe utilizada para invocar um *kernel.* No caso da aplicação de conversão do espaço de cores RGB para tons de cinzento, devido às características inerentes do algoritmo, é possível mapear cada pixel a uma GPU *Thread*. O número de pixéis da altura da imagem (*height*) corresponde ao número de blocos e a largura (*width*) corresponde ao número de *threads* por bloco.

$$\text{kernel\_name} <<< \text{block, threads\_per\_block} >>> (\text{argument list})$$

Equação 4.4.3.1 – Sintax utilizada na chamada de um kernel CUDA

Na Figura 4.4.3.4, é representado o *kernel* que faz a conversão do formato de cor RGB para tons de cinzento. Cada pixel foi mapeado a uma GPU *thread,* pelo que o identificador da *thread* atual corresponde à posição do pixel na imagem. Como referido anteriormente, a altura da imagem foi mapeada ao número de blocos e a largura ao número de *threads* por bloco, representado por *blockDim*. As variáveis *blockIdx* e *threadIdx* correspondem à posição atual no *height* e *width* da imagem, respetivamente.

```
__global__ void kernel_grayscale(int *bmp, int *gray)
{
   int pos = (blockDim.x * blockIdx.x + threadIdx.x);
   int pos_Abs = pos * 3;
   gray[pos] = bmp[pos_Abs] * 0.299 + bmp[pos_Abs + 1] * 0.587 + bmp[pos_Abs + 2] * 0.114;
}
```

Figura 4.4.3.4 – Grayscale (tons de cinzento) Kernel

A execução do(s) *kernel*(s) dá-se de forma assíncrona, para que a computação a realizar no GPU possa ser sobreposta com a comunicação entre *host* e *device*. Quando dois *kernels* são invocados consecutivamente, o segundo fica em *queue* e começa apenas o seu processamento quando a execução do primeiro concluir. A função *cudaMemcpy* é utilizada com a direção *cudaMemcpyDeviceToHost,* o CPU fica em modo de espera, até que a execução dos *kernels* seja concluída, copiando posteriormente o resultado obtido no *device* para ser processado e analisado

no *host*. Finalmente, se a rotina de aceleração não for invocada novamente, é necessário libertar a memória reservada em GPU, através da função *cudaMemFree*.

4.5. CUDA em sistemas embebidos

O lançamento do SoC Tegra K1, na primeira metade de 2014, marcou a entrada da *Nvidia* e do CUDA no mercado de sistemas embebidos. Baseado na arquitetura de GPU Kepler (com 192 núcleos CUDA), apresenta um processador quad-core ARM Cortex-A15 (@2.32 GHz) e um consumo energético de 5 *watts*. Com suporte para CUDA 6 e OpenGL 4.4, o Tegra K1 é a primeira solução móvel capaz de suportar todas as APIs de computação por GPU recentes. O desempenho do GPU no cálculo, em paralelo, de operações com vírgula flutuante de 32 bits é capaz de atingir mais de 300 *GFLOPS*. O SoC possuí também um *Image Signal Processor* (ISP), que permite realizar diversas tarefas relacionadas com a aquisição de fotogramas, como *autofocus*, conversão de cor, redução de ruído, etc.

A placa de desenvolvimento embebido *Jetson* TK1, representada na Figura 4.5.1, apresenta um processador Tegra K1, 2GB de memória RAM, 16GB de memória interna e numerosas portas E/S (USB 3.0, HDMI, SATA, mini-PCIE slot) que permite ao programador aceder às capacidades do SoC Tegra e dos seus GPGPU *computing cores.* O kit de desenvolvimento corre sobre o sistema operativo *Linux For Tegra*. Uma versão modificada, pela *Nvidia*, da distribuição *Ubuntu* 14.04.

Figura 4.5.1 – Plataforma de desenvolvimento Jetson TK1

Como referido anteriormente, numa arquitetura SoC, a memória do sistema é partilhada entre CPU e GPU, pelo que a latência de transferência de dados pelo barramento *PCIExpress*, deixa de ser um fator a considerar, possibilitando assim, a aceleração de novas aplicações. O GPU poderá então ser utilizado como um coprocessador, equilibrando a carga de trabalho entre ambos.

4.6. Desenho

Durante o estudo realizado sobre as técnicas de reconstrução morfológica, duas dessas técnicas, mostraram-se mais pertinentes para a utilização no sistema, uma vez que as restantes eram dedicadas à extensão da metodologia a imagens em tons de cinzento. Assim, no capítulo 3 o *Sequential Reconstruction Algorithm* e o *Parallel Reconstruction Algorithm* propostos por [41] e as vantagens que as novas arquiteturas disponíveis podem trazer ao desempenho do sistema, são analisados mais profundamente neste capítulo.

Os módulos de pré-processamento, *Threshold* local, pós-processamento e de cálculo dos valores de intensidade presentes na imagem, através de histogramas verticais e horizontais, são claros candidatos a aceleração em GPU, devido ao paralelismo demonstrado pelos seus algoritmos. O foco deste capítulo é o desenho, implementação e otimização dos módulos de *Threshold Local* e Reconstrução Morfológica, uma vez que a utilização dos módulos de pré-processamento e pós-processamento é opcional e vai depender das condições e ambiente envolvente, onde o sistema estará inserido.

Luc Vincent, referiu no artigo que publicou em 1993, que o processo de reconstrução sequencial era mais rápido que o paralelo [41]. Com a possibilidade de acelerar tarefas altamente paralelas nos GPUs é importante verificar se a afirmação ainda se aplica aos dias de hoje. Na Figura 4.6.1 o algoritmo de reconstrução sequencial (Figura 3.3.10.4) é posicionado face ao esquema de seleção da tecnologia apresentado por [55], que está representado na Figura 4.3.1. O nível de paralelismo apresentado pelo algoritmo é baixo, no entanto, no processamento de imagens o tamanho dos dados a processar será bastante elevado.

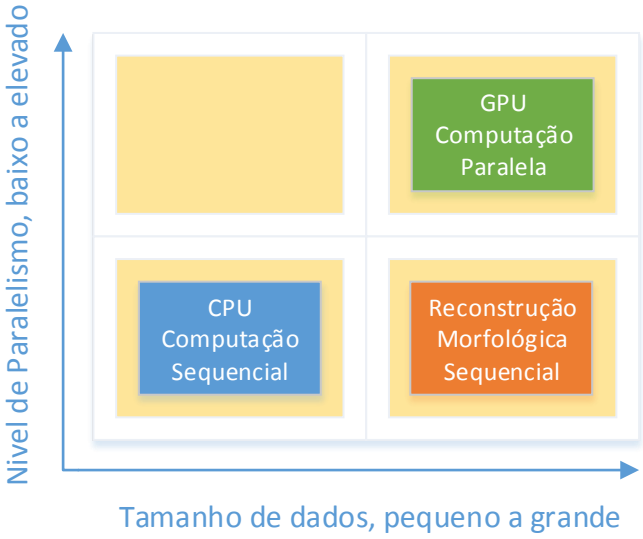


Figura 4.6.1 – Características do algoritmo de Reconstrução Morfológica Sequencial, face às arquiteturas disponíveis

O algoritmo de reconstrução paralelo (Figura 3.3.9.1), baseado em dilatações geodésicas é um candidato claro para aceleração em GPU, como se pode analisar na Figura 4.6.2. No entanto, será interessante estudar se a aceleração proporcionada pelo GPU é suficiente para que a utilização do algoritmo no sistema seja viável. Um dos possíveis problemas deste algoritmo advém do número de iterações e da condição de estabilidade.

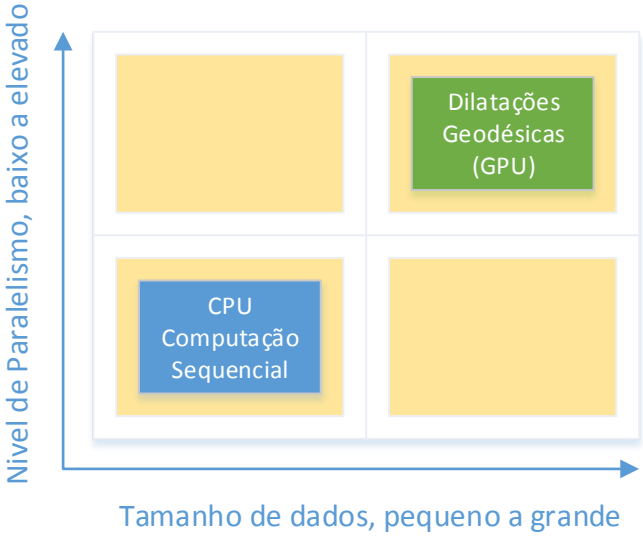


Figura 4.6.2 – Características do algoritmo de Reconstrução Morfológico paralelo, face às arquiteturas disponíveis

4.6.1. Diagrama de Blocos do Sistema

Uma vez que a implementação de ambos os algoritmos de reconstrução morfológica, paralela e sequencial, foi considerada pertinente, especialmente devido há emergência das GPUs na aceleração do metodologias altamente paralelas, duas versões do sistema são apresentadas.

O primeiro, delega ao CPU (*host*) as tarefas de gestão e leitura de ficheiro e resultados, utilizando o GPU (*device*) para acelerar os módulos de *Threshold Local*, *Parallel Flood Fill* e Histogramas. As conclusões mais importantes a retirar deste sistema são: o número de iterações que o módulo de reconstrução morfológica necessita até atingir a estabilidade e a vantagem de ter todos os blocos com grande carga computacional a ser processados sequencialmente no GPU, diminuindo o número de interações com o CPU, excluindo assim (em parte) os problemas de latência da transferência de dados através do barramento *PCIExpress*.

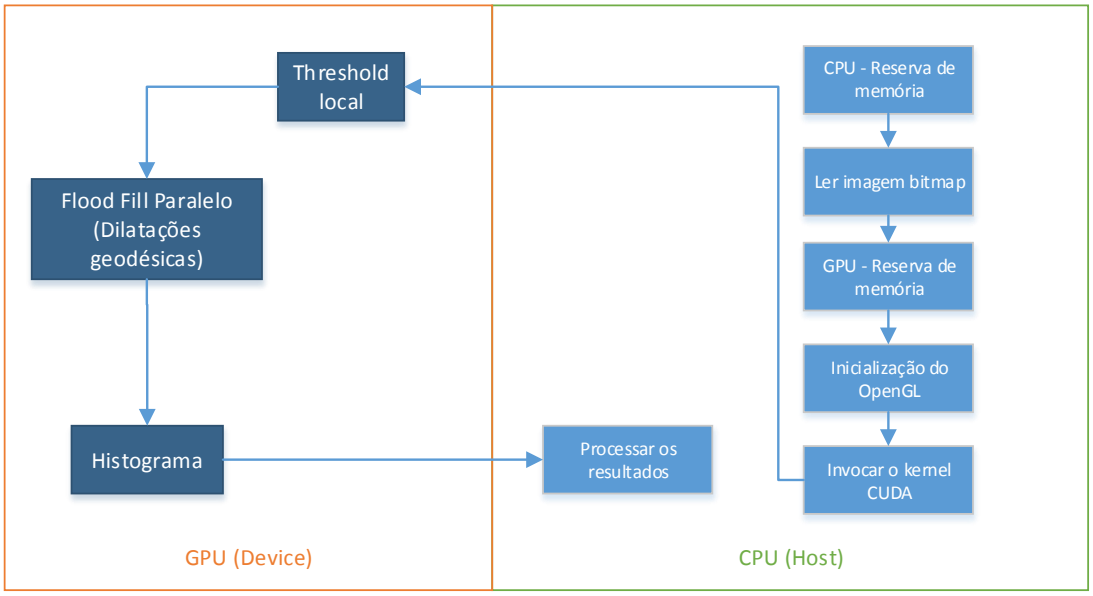


Figura 4.6.1.1 – Diagrama de Blocos (Reconstrução Morfológica Paralela)

A segunda abordagem (Figura 4.6.1.2), realiza o processo de reconstrução morfológica sequencial no CPU, que continua a realizar as tarefas de escalonamento, leitura e análise de resultados. O *Host* está sujeito a uma carga computacional muito superior, no entanto, como a sua frequência de operação é muito superior ao GPU, não faz sentido "acelerar" um algoritmo sequencial no GPU. Os histogramas são realizados também em CPU, uma vez que a latência do barramento *PCIExpress*, inviabiliza a realização de pequenas operações no *device*.

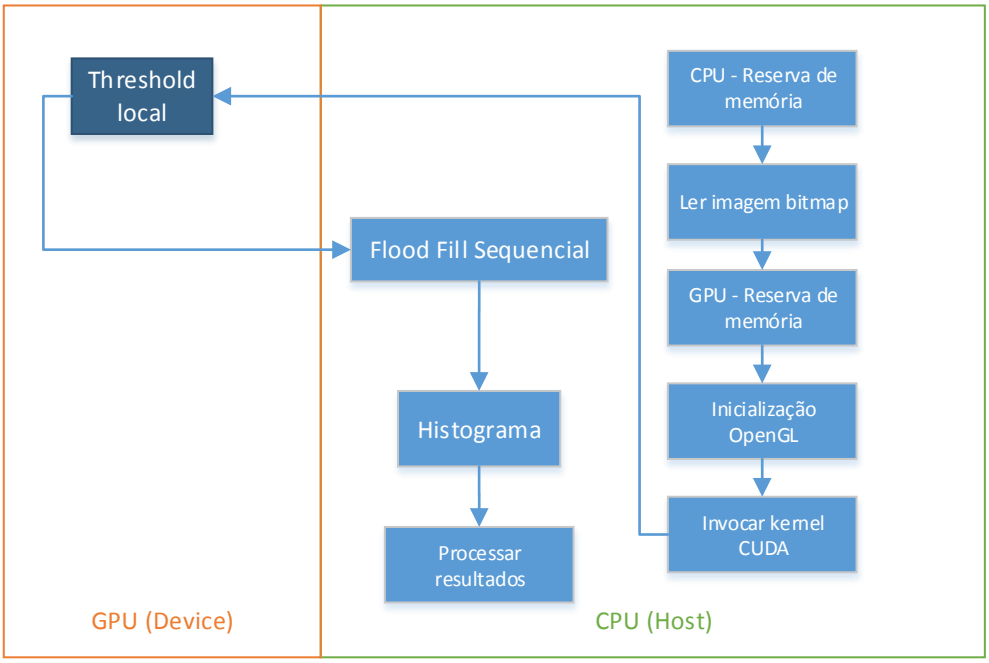


Figura 4.6.1.2 – Diagrama de blocos (Reconstrução Morfológica Sequencial)

4.6.2. Sistema Utilizado

Durante a realização deste trabalho de dissertação, foi utilizado um Desktop PC, para o desenvolvimento e *profiling* da aplicação em CUDA. O Sistema é composto por um processador *quad-core* I7 4790 a 3,6GHz, 16 GB de memória RAM, *Solid State Drive* de 256 GB e uma placa gráfica *Nvidia* GTX 970 com 1664 núcleos CUDA.

4.6.3. *Local Threshold*

Em ambas as abordagens, o algoritmo de *Threshold* local é um claro candidato para aceleração em GPU. Isto deve-se, não só pelo elevado nível de paralelismo e tamanho de dados apresentado, mas também porque prepara a imagem marcador para o módulo seguinte. Na Figura 4.6.3.1, é proposto um algoritmo de *Threshold* Local, com aceleração em GPU (CUDA), que prepara o módulo seguinte (reconstrução morfológica).

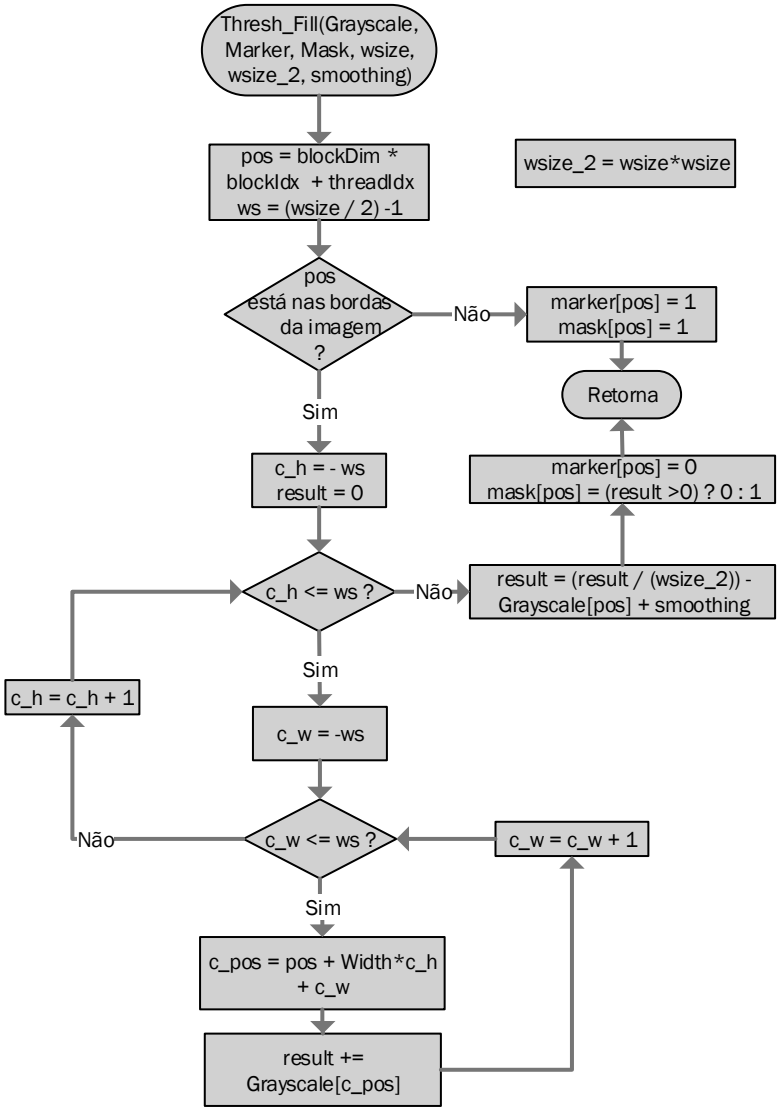


Figura 4.6.3.1 – Fluxograma do módulo de Threshold local

O princípio desta abordagem, é que cada pixel da imagem tons de cinzento pode ser mapeado a uma GPU *thread.* Como a execução do *kernel* é assíncrona, a primeira etapa do algoritmo é identificar a posição da *thread* atual na imagem. Neste contexto, a variável *blockDIM* corresponde ao número de *threads* por bloco, ou seja a largura da imagem (*width*), e a variável *blockIdx* corresponde ao valor do bloco atual (*height*). A variável *threadIdx* corresponde ao número da *thread* atual no bloco, ou seja, o valor de coluna atual. A comparação entre o posicionamento do pixel atual em CUDA, com as metodologias tradicionais, é estabelecida pelas Equação 4.6.3.1 e Equação 4.6.3.2, respetivamente.

$$Pos = blockDim * blockIdx + threadIdx$$

Equação 4.6.3.1 – Equação de posição em CUDA

$$Pos = Width * Current\_Height + Current\_Width$$

Equação 4.6.3.2 – Equação de posição tradicional

O algoritmo proposto apenas apresenta um bloco de decisão, para identificar se o pixel atual faz parte dos valores que são considerados da margem da imagem, uma vez que o algoritmo necessita de informação dos valores que rodeiam o pixel atual. Assim, a condição que determina se o pixel pertence às margens da imagem ou se deve ser processado, pode ser obtida pela Equação 3.4.2.2.

$$blockIdx < ws \;||\; blockIdx > grimDim - ws \;||\; threadIdx < ws \;||\; threadIdx > blockDim - ws$$

Equação 4.6.3.3 – Condição que determina se o pixel pertence ou não às margens da imagem

Após a conclusão do *kernel,* ambas as imagem máscara e marcador estão disponíveis para serem utilizadas pelo módulo de reconstrução morfológica. No sistema proposto na Figura 4.6.1.1, apenas é necessário invocar o *kernel* de reconstrução morfológica paralela para continuar o processo. No entanto, o sistema proposto na Figura 4.6.1.2, necessita de copiar os dados referentes às imagens máscara e marcador da memória do GPU para a memória do CPU, através da chamada da API *cudaMemcpy,* uma vez que o processo de reconstrução é realizado no CPU.

4.6.4. Reconstrução Morfológica Paralela

O algoritmo é composto por diversas dilatações geodésicas, que quando iteradas até à estabilidade, correspondem à reconstrução morfológica da imagem. Visto ser um processo iterativo, é necessário alocar espaço na memória do GPU para uma nova variável temporária, do tamanho da imagem marcador. O fluxograma presente na Figura 4.6.4.1 refere-se às dilatações geodésica em GPU. A condição que determina se a posição do pixel pertence às margens da imagem é igual à presente na Equação 3.4.2.2.

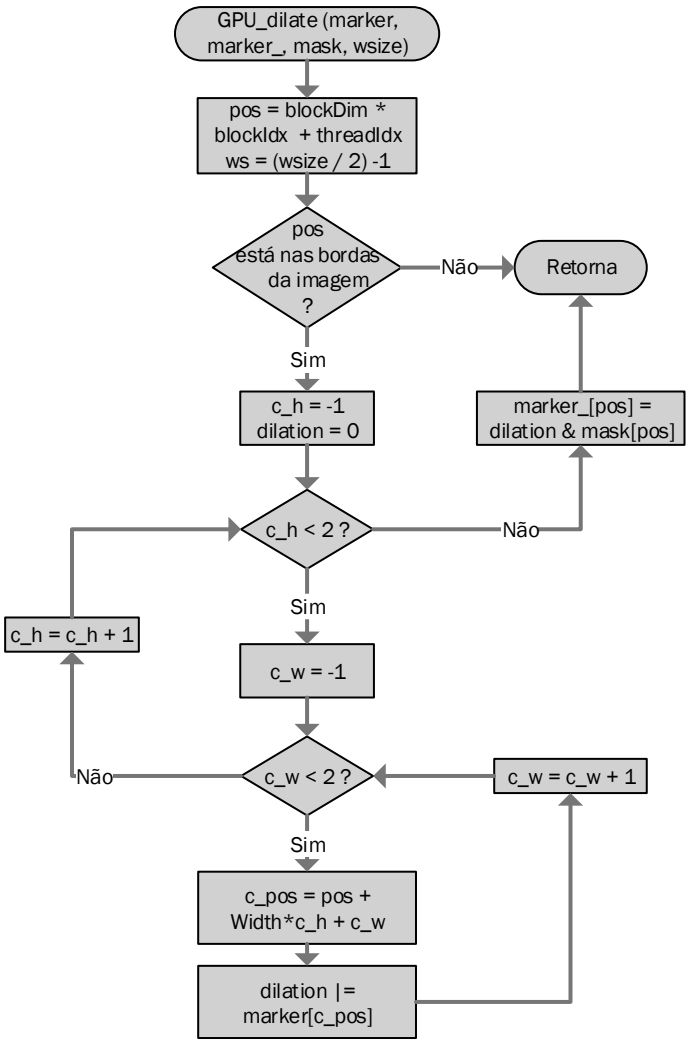


Figura 4.6.4.1 – Fluxograma referente às dilatações geodésicas em GPU

A imagem *marker,* contem informação referente à iteração anterior do algoritmo, e a imagem *marker_* recebe a informação referente à iteração atual. É então necessário alternar a posição dos apontadores das imagens marcador a cada iteração, na chamada do *kernel*, para que o algoritmo consiga avançar o processo de dilatação. A comparação entre os valores dos pixéis das imagens *marker* e *marker_,* facilitam a deteção da condição de paragem, uma vez que um se refere à iteração atual e outro à iteração anterior. Quando ambos apresentarem os mesmos valores, a condição de paragem é atingida.

4.6.5. Reconstrução Morfológica Sequencial

O processo de reconstrução morfológica em CPU é composto por duas iterações principais (Figura 3.3.10.4): na primeira, a imagem é percorrida de cima a baixo em *Raster Order* e na segunda, a imagem é percorrida de baixo para cima em *Anti-Raster Order*. Semelhante aos algoritmos apresentados anteriormente, as margens da imagem não podem ser processadas, uma vez que a metodologia necessita de informação referente ao pixel anterior e posterior ao atual.

O Fluxograma da Figura 4.6.5.1, representa a primeira iteração do processo, onde o valor do pixel atual depende do valor da vizinhança N_g^+ (Figura 3.3.10.3) e do valor do pixel atual da imagem da máscara. De forma a otimizar o processo foram criados três apontadores: o apontador *ptr_cmarker* que aponta para a posição atual na imagem marcador, o apontador *ptr_pmarker* que aponta para a linha anterior, mantendo o mesmo valor de coluna, e o apontador *ptr_mask* que aponta para a posição atual, mas na imagem máscara. A Equação 4.6.5.1 expressa a relação entre o valor atual da imagem marcador com a sua vizinhança e a imagem máscara.

EQ1: *ptr_cmarker = (*ptr_mask == 1) ?

(*(ptr_pmarker - 1) | *(ptr_pmarker) | *(ptr_pmarker + 1) | *(ptr_cmarker - 1) | *ptr_cmarker) : 0

Equação 4.6.5.1 – Equação que determina o valor do pixel atual da imagem marcador

Outra otimização adicionada ao algoritmo, foi a verificação prévia do valor da máscara, no pixel atual, uma vez que, se esta apresentar o valor zero, o resultado do marcador atual será sempre zero, tornando desnecessário o cálculo dos valores da vizinhança.

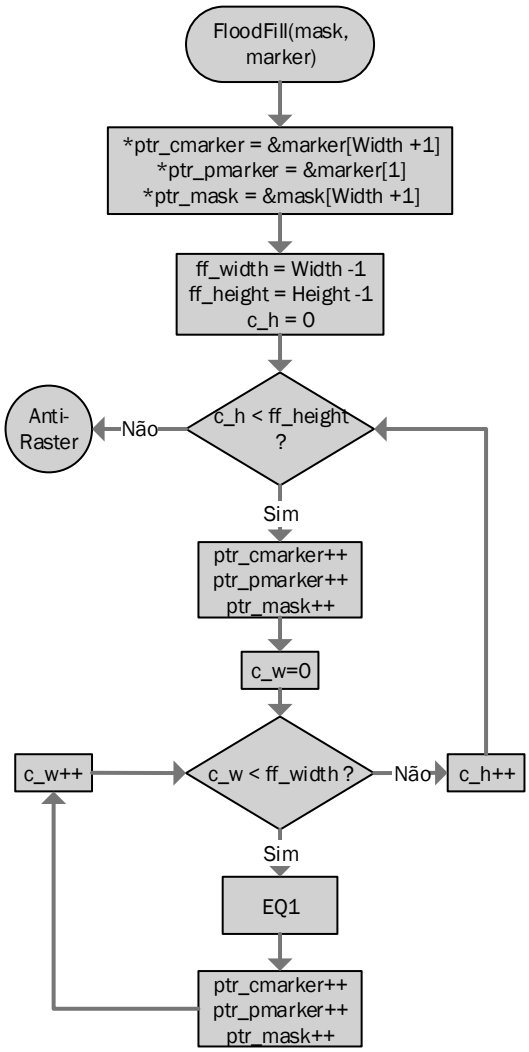


Figura 4.6.5.1 – Fluxograma de Flood Fill em Raster Order

Após a conclusão da pesquisa em *Raster Order*, o algoritmo deve percorrer a imagem de baixo para cima em *Anti-Raster Order*. Essencialmente, a lógica apresentada pelo sistema será a mesma, isto é, apenas os valores e apontadores de posição serão diferentes, pois verão os seus valores decrementados ao invés de incrementados.

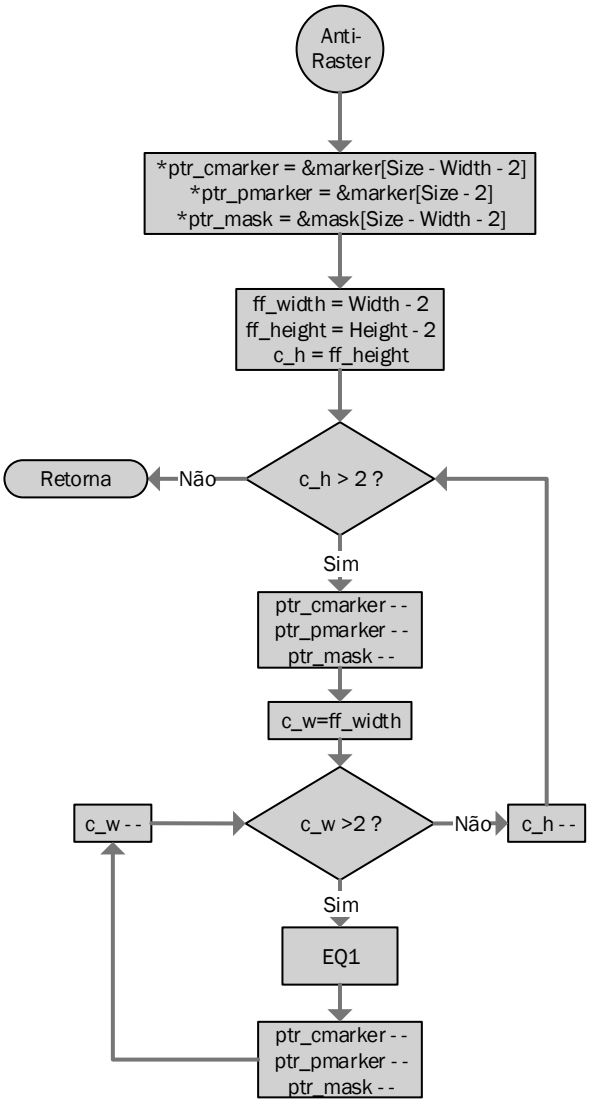


Figura 4.6.5.2 – Fluxograma do Flood Fill em Anti-Raster Order (continuação)

Após o término da ordenação *Anti-Raster*, não é possível concluir se a estabilidade foi alcançada, sendo necessário executar duas vezes a rotina de *Flood Fill* antes de ser possível determinar se a estabilidade foi alcançada. A partir da primeira execução do algoritmo, a determinação do marcador será mais rápida. O princípio apresentado pela Equação 4.6.5.2 é que, se o valor da posição do marcador atual se encontra a 1, o pixel em questão já foi validado por uma iteração anterior do algoritmo. Esta otimização apresentará melhor resultados, quanto maior for o número de iterações realizadas e maior for a resolução da imagem.

EQ2: *ptr_cmarker = (*ptr_cmarker ==1) ? 1 : (*ptr_mask == 0) ?

(*(ptr_pmarker - 1) | *(ptr_pmarker) | *(ptr_pmarker + 1) | *(ptr_cmarker + 1) | *ptr_cmarker) : 0

Equação 4.6.5.2 – Equação que determina o valor do pixel atual da imagem marcador, após a primeira iteração

O algoritmo de reconstrução morfológica proposto é utilizado como parte de um sistema de segmentação da região da matrícula, pelo que em muitos casos de aplicação, não é necessário atingir a estabilidade no processo de reconstrução.

4.6.6. Post Fill

O objetivo do algoritmo de *Post Fill,* representado na Figura 4.6.6.1*,* é simplesmente segmentar as áreas preenchidas pelo algoritmo de reconstrução. Como descrito no terceiro capítulo, é considerado um buraco (*hole*) de uma imagem binária, um objeto que não esteja ligado às margens da imagem. O módulo de *Post Fill* é a última etapa no processo de segmentação, onde a imagem criada apenas contém as regiões preenchidas pelo algoritmo de reconstrução morfológica, que idealmente serão apenas os caracteres da matrícula.

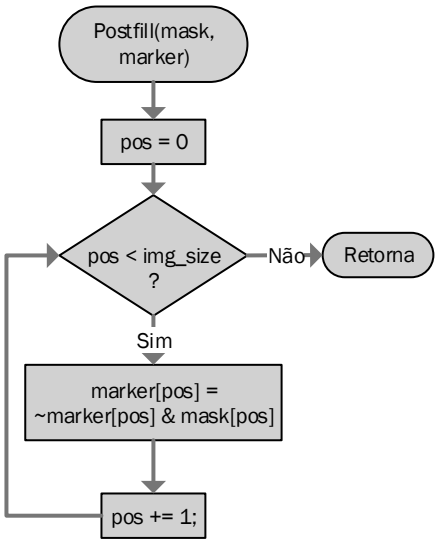


Figura 4.6.6.1 – Fluxograma referente ao algoritmo de PostFill

4.7. Implementação

O processo de implementação apresentou três fases de execução distintas, que proporcionou um sistema de qualidade superior. Inicialmente, o objetivo foi implementar todas as rotinas propostas em CPU, de modo a depurar possíveis erros na lógica dos algoritmos. De seguida foi incluída a API OpenGL na aplicação, facilitando a depuração de resultados e a calibração do sistema. Finalmente, foram implementados os *kernels* referentes aos algoritmos de *Threshold* local e reconstrução morfológica paralela e respetivo código no *host* para suportar a *framework* CUDA.

4.7.1. Sistema de testes inicial

De modo a ser possível realizar testes mais específicos sobre o sistema e em diferentes cenários, foi necessário criar rotinas de leitura e escrita em ficheiros *Bitmap*, uma vez que este tipo de ficheiros, na sua maioria, não apresentam compressão de imagem. Assim, foi possível analisar imagens de 1.2 megapixéis, em tons de cinza provenientes de uma câmara de videovigilância e imagens em RGB provenientes de uma máquina Canon de 12 megapixéis, sem compressão. O módulo de conversão do espaço de cores RGB para tons de cinzento foi implementado em CPU e GPU para efeitos de teste, visto não haver qualquer necessidade da sua inclusão numa versão final do sistema proposto.

A implementação deste sistema foi muito importante na depuração de erros de lógica nos algoritmos, porém a obtenção de resultados ficava muito distante do desejável, ao ponto de atrasar o processo de desenvolvimento. Guardar quatro ficheiros BMP, mesmo utilizando um disco SSD, é um processo demorado, devido à resolução das imagens utilizadas, uma vez que cada ficheiro apresentava, um tamanho médio de 5 MB. Assim, a criação de um interface visual, capaz de realizar o *rendering* das imagens, sem necessitar de guardar a informação em disco, tornou-se a prioridade principal da implementação.

4.7.2. OpenGL

A segunda fase de implementação centrou-se na necessidade da utilização de uma API que permite a depuração visual das várias etapas do algoritmo, através do *rendering* de imagens armazenadas na memória RAM ou na memória do GPU. O OpenGL foi a API escolhida, uma vez que apresenta interoperabilidade com a *framework* CUDA e é suportada por diferentes sistemas operativos, tais como: Linux, Windows, OSX, entre outros.

A integração da API OpenGL com o sistema acelerou o processo de desenvolvimento, especialmente na calibração dos algoritmos. Através de atalhos adicionados ao teclado, a aplicação reage no momento, sem haver a necessidade de executar a aplicação de novo, cada vez que se ajusta uma variável. Além de proporcionar uma rápida mudança entre as diferentes imagens produzidas pelo sistema (RGB, tons de cinzento*, Threshold, Flood Fill*, entre outras), a aplicação permite ajustar valores, como o tamanho da janela de *Threshold* ou o número de iterações do algoritmo de *Flood Fill* e observar as alterações nos resultados obtidos pelo sistema.

A aplicação necessita de sete passos essenciais para a configuração da API OpenGL com o sistema:

1. Inicializa a API, cria a janela e seleção do modo de visualização (Figura 4.7.2.1).
2. Cria a textura 2D retangular, com as dimensões da imagem, alinhada com a janela de visualização (Figura 4.7.2.2).
3. Faz o mapeamento da imagem à textura criada no passo anterior. Para isso, é necessário especificar o tamanho da imagem, formato de cor (*INTENSITY*, RGB) e ao tipo da variável que contém a informação referente à imagem (*unsigned char),* como se pode verificar na Figura 4.7.2.3.

```
void initGL(int *argc, char **argv)
{
	glutInit(argc, argv);
	glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
	glutInitWindowSize(Width, Height);
	glutCreateWindow("ANPR");

	glewInit();
}
```

Figura 4.7.2.1 – Inicialização da API OpenGL

```
void Display(void)
{
	glClear(GL_COLOR_BUFFER_BIT);

	glBindTexture(GL_TEXTURE_2D, texid);
	glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo_buffer);
	glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, Width, Height,
	GL_RGB, GL_UNSIGNED_BYTE, OFFSET(0));
	glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);

	glDisable(GL_DEPTH_TEST);
	glEnable(GL_TEXTURE_2D);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

	glBegin(GL_QUADS);
	glVertex2f(0, 0);
	glTexCoord2f(0, 0);
	glVertex2f(0, 1);
	glTexCoord2f(1, 0);
	glVertex2f(1, 1);
	glTexCoord2f(1, 1);
	glVertex2f(1, 0);
	glTexCoord2f(0, 1);
	glEnd();
	glBindTexture(GL_TEXTURE_2D, 0);
	glutSwapBuffers();

}
```

Figura 4.7.2.2 – OpenGL, criação da textura

```
void init_Data(bool rgb)
{
	GLint bsize;
	long glSize = (rgb) ? (3 * Height * Width) : (Height * Width);

	glGenBuffers(1, &pbo_buffer);
	glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo_buffer);
	glBufferData(GL_PIXEL_UNPACK_BUFFER,
	glSize, glTexture, GL_STREAM_DRAW);
	glGetBufferParameteriv(GL_PIXEL_UNPACK_BUFFER, GL_BUFFER_SIZE, &bsize);
	glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);

	glGenTextures(1, &texid);
	glBindTexture(GL_TEXTURE_2D, texid);

	if (rgb)
	{
		glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, Width, Height, 0, GL_RGB,
GL_UNSIGNED_BYTE, NULL);
	}
	else
	{
		glTexImage2D(GL_TEXTURE_2D, 0, GL_INTENSITY, Width, Height, 0,
GL_INTENSITY, GL_UNSIGNED_BYTE, NULL);
	}
	glBindTexture(GL_TEXTURE_2D, 0);

	glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
	glPixelStorei(GL_PACK_ALIGNMENT, 1);
}
```

Figura 4.7.2.3 – OpenGL, mapeamento da textura

4. Atribuir controlo ao teclado sobre a aplicação, que permite alternar entre os diversos estágios de segmentação e controlar o valor dos parâmetros. Na Figura 4.7.2.4 é possível analisar a seleção de estados referentes à tecla premida no teclado. Como referido anteriormente, este passo é muito importante para uma depuração, calibração e validação de resultados eficiente.

```
void keyboard(unsigned char key, int /*x*/, int /*y*/)
{
	switch (key)
	{
	case '1': //RGB
		to_uchar();
		glutSetWindowTitle("ANPR");
		update_texture();
		break;
	case '2': //grayscale
		bw_to_uchar(GrayScale);
		glutSetWindowTitle("ANPR (Grayscale)");
		update_texture();
		break;
	case '3': //Threshold
		bw_to_uchar(Threshold);
		glutSetWindowTitle("ANPR (Threshold)");
		update_texture();
		break;
	case '4': //Flood Fill
		bw_to_uchar(Dilation);
		glutSetWindowTitle("ANPR (Dilation)");
		update_texture();
		break;
	case '+': //Threshold
		thresh_wsize(true);
		bw_to_uchar(Threshold);
		glutSetWindowTitle("ANPR (Threshold)");
		update_texture();
		break;
	case '-': //Threshold
		thresh_wsize(false);
		bw_to_uchar(Threshold);
		glutSetWindowTitle("ANPR (Threshold)");
		update_texture();
		break;

	}
}
```

Figura 4.7.2.4 – OpenGL, Comandos de teclado

5. Rodar a textura em 180º, uma vez que o armazenamento das imagens em ficheiros BMP é feito através de uma organização *bottom-up*. Ao rodar apenas a textura, não é necessário repetir o processo cada vez que se carregue uma imagem diferente (Figura 4.7.2.5).
6. Redimensionar a textura, de forma a preencher a janela de visualização. Assim, o tamanho da janela pode ser alterado que a imagem acompanhará as alterações, permitindo assim o aumento ou diminuição da imagem visualizada, ajudando na análise visual (Figura 4.7.2.6).

```
void rotation(void) {
	glMatrixMode(GL_TEXTURE);
	glLoadIdentity();
	glRotatef(180, 0.0, 0.0, 1.0);
	glScalef(-1.0f, 1.0f, 1.0f);
	glMatrixMode(GL_MODELVIEW);

}
```

Figura 4.7.2.5 – OpenGL, Rotação de textura

```
void reshape(int x, int y)
{
	glViewport(0, 0, x, y);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	glOrtho(0, 1, 0, 1, 0, 1);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();

}
```

Figura 4.7.2.6 – OpenGL, Redimensionamento de textura

7. Definir a frequência de atualização da janela, ou seja, o número de vezes por segundo que a imagem é atualizada (Figura 4.7.2.7). De seguida, a aplicação entra em modo de espera, até receber algum valor vindo do teclado ou a aplicação for fechada manualmente.

```
void timerEvent(int value)
{
	if (glutGetWindow())
	{
		glutPostRedisplay();
		glutTimerFunc(REFRESH_DELAY, timerEvent, 0);
	}
}
```

Figura 4.7.2.7 – OpenGL, Taxa de atualização da janela

4.7.3. Aplicação CUDA

Após a criação do interface gráfico e da depuração da lógica dos algoritmos utilizados, as rotinas de *Threshold* local e reconstrução morfológica paralela, foram migradas para o GPU. O respetivo código no *host* foi criado para coordenar a execução dos *kernels*.

A aplicação começa por ler o cabeçalho da imagem BMP a processar e reserva o espaço necessário na memória RAM, para armazenar a informação referente à imagem lida e inicializa a API gráfica OpenGL, como demonstra a Figura 4.7.3.1.

```
void IMGP_fast(int argc, char **argv)
{
	bool img_check = LoadBitmap("Images/021.bmp");
	if (!img_check)
	{
		cout << "Image failed to load!" << endl;
		system("pause");
	}
	else{
		to_uchar();
		initGL(&argc, argv);
		glutDisplayFunc(Display);
		init_Data(true);
		glutCloseFunc(cleanup);
		glutKeyboardFunc(keyboard);
		rotation();
		glutReshapeFunc(reshape);
		glutTimerFunc(REFRESH_DELAY, timerEvent, 0);
		cout << "Image Resolution: " << Height << "x" << Width << endl;
		cout << "Press 1 for RGB image!" << endl;
		cout << "Press 2 for Grayscale image!" << endl;
		cout << "Press 3 for Threshold image!" << endl;
		cout << "Press 4 for FloodFill image!" << endl;
```

Figura 4.7.3.1 – Estrutura de uma aplicação CUDA com OpenGL (1)

De seguida, é reservado espaço na memória RAM, para receber os resultados das operações de *Threshold* e *Flood Fill* em GPU, sendo inicializado o *device*, como é possível observar na Figura 4.7.3.2. O processo de alocação de memória no GPU é demorado, por isso, a reserva de espaço deve ser feita durante a inicialização do *device,* como demonstrado na Figura 4.7.3.3, e a sua libertação no final do processamento.

```
    //CPU alloc
    Threshold = (int *)malloc(Height*Width*sizeof(int));
    FFill     = (int *)malloc(Height*Width*sizeof(int));

    //CUDA malloc
    start_cuda(Height, Width);
```

Figura 4.7.3.2 – Reserva de Memória RAM e Inicialização do device (2)

```
void start_cuda(int height, int width)
{
	// set up device
	int dev = 0;
	cudaDeviceProp deviceProp;
	cudaGetDeviceProperties(&deviceProp, dev);
	printf("Using Device %d: %s\n", dev, deviceProp.name);
	cudaSetDevice(dev);
	//GPU Memory allocation
	cudaMalloc(&d_gray, height * width * sizeof(int));
	cudaMalloc(&d_thresh, height * width * sizeof(int));
	cudaMalloc(&d_marker, height * width * sizeof(int));
	cudaMalloc(&d_marker_, height * width * sizeof(int));
}
```

Figura 4.7.3.3 – Inicialização do device

Após a inicialização do *device* é necessário chamar a função, Figura 4.7.3.4, que prepara a execução do *kernel,* Figura 4.7.3.5. O primeiro passo, consiste em copiar a informação a processar da memória do CPU para o GPU, neste caso a imagem em tons de cinzento. Após a conclusão da operação, o *kernel* que realiza a operação de *Threshold* local sobre a imagem (*Grayscale*) e cria as imagens máscara (*Threshold*) e marcador (*FFill*) é chamado, como demonstrado na Figura 4.6.3.1, preparando a operação de *Flood Fill*. A execução do *kernel* dá-se de forma assíncrona, pelo que o pedido de cópia dos resultados obtidos para a memória do CPU, ficam em espera até que a sua execução termine.

```
 //CUDA call
 gpu_anpr(GrayScale, Threshold, FFill, Height, Width, 9);
```

Figura 4.7.3.4 – Função que prepara a execução do kernel (3)

```
void gpu_anpr(int *gray, int *thresh, int *FFill, int height, int width, int wsize)
{
        //CPU -> GPU memory copy
        cudaMemcpy(d_gray, gray, height * width * sizeof(int), cudaMemcpyHostToDevice);

        //Invoke Kernel
        kernel_thresh_fill << <height, width >>> (d_thresh, d_gray, d_marker,d_mask, wsize);

        //GPU -> CPU memory copy
        cudaMemcpy(thresh, d_thresh, height * width * sizeof(int), cudaMemcpyDeviceToHost);
        cudaMemcpy(FFill, d_marker, height * width * sizeof(int), cudaMemcpyDeviceToHost);

}
```

Figura 4.7.3.5 – Função que gere a transferência de dados entre CPU e GPU e invoca o kernel

A aplicação utilizada como exemplo da interação entre o *host* e o *device,* realiza o processo de reconstrução morfológica no CPU, a partir dos valores das imagens máscara (*Threshold*) e marcador (*FFill*) obtidos do GPU, como é possível verificar na Figura 4.7.3.6. Após a conclusão do processo de segmentação a aplicação fica à espera de receber comandos através do teclado, de forma a alterar os parâmetros de configuração do sistema e visualizar o sucedido. Quando a interface gráfica é fechada, a aplicação sai do ciclo de espera, liberta o espaço alocado em memória e termina a sua execução, como ilustrado na Figura 4.7.3.7.

```
//Flood Fill
sequential_fill(Threshold, FFill);
postfill(Threshold, FFill);
```

Figura 4.7.3.6 – Reconstrução Morfológica Sequencial

```
                //OpenGL Loop
                glutMainLoop();
                //END CUDA
                gpu_free();
                cuda_end();
                //
                //Release Alloc
                free(Bitmap);
                free(GrayScale);
                free(Threshold);
                free(Dilation);
        }
}
```

Figura 4.7.3.7 – Estrutura de uma aplicação CUDA com OpenGL (4)

4.8. Resultados práticos

De um modo geral, os resultados obtidos referentes à implementação em CUDA foram bastante positivos. A implementação das duas metodologias, de reconstrução morfológica, mostraram-se essenciais para o sucesso do projeto de investigação. A metodologia proposta na Figura 4.6.4.1 apresenta um nível de paralelismo elevado, no entanto a abordagem proposta em Figura 4.6.5.1 e Figura 4.6.5.2 não necessita de tantas iterações, até atingir a estabilidade. O interface gráfico criado com recurso à API OpenGL demonstrou-se muito útil, especialmente na depuração visual dos resultados obtidos e na calibração dos algoritmos.

Todos os resultados foram obtidos utilizando o sistema descrito em 4.6.2, no entanto o sistema pode ser facilmente portado para um sistema embebido, utilizando uma plataforma de desenvolvimento como a *Jetson TK1* (Figura 4.5.1). A comparação de resultados não será direta, visto que o sistema utilizado apresenta um desempenho muito superior. No entanto, a arquitetura SoC Tegra Tk1 apresenta uma arquitetura de memória unificada, pelo que não existirá a latência na transmissão de dados pelo barramento *PCIExpress*.

4.8.1. Interface gráfica

A interface gráfica da aplicação foi de fácil implementação e ajudou muito na qualidade do trabalho realizado, uma vez que diminuiu muito o tempo necessário entre o teste de parâmetros do sistema e a visualização do resultado. Na Figura 4.8.1.1 é possível observar dois estados do funcionamento da aplicação: à esquerda é apresentada a imagem em tons de cinzento e à direita é apresentada a segmentação da matrícula. Para alternar entre as diversas etapas de segmentação, apenas é necessário pressionar as teclas 1 a 4 do teclado. As teclas '+' e '-' também foram utilizadas para alterar alguns parâmetros do sistema, como o tamanho da vizinhança utilizada pelo algoritmo de *Threshold* local, sem necessitar de executar a aplicação de novo.

Figura 4.8.1.1 – Interface gráfica em OpenGL

A interface é mais importante para a calibração em ambientes mais complexos, como o representado na Figura 4.8.1.2, onde estão presentes 3 veículos, num ambiente rico em informação de *background*. Através da alteração dos parâmetros de configuração e visualização das alterações ao processo de segmentação, no momento, resultados similares aos obtidos não são difíceis de obter.

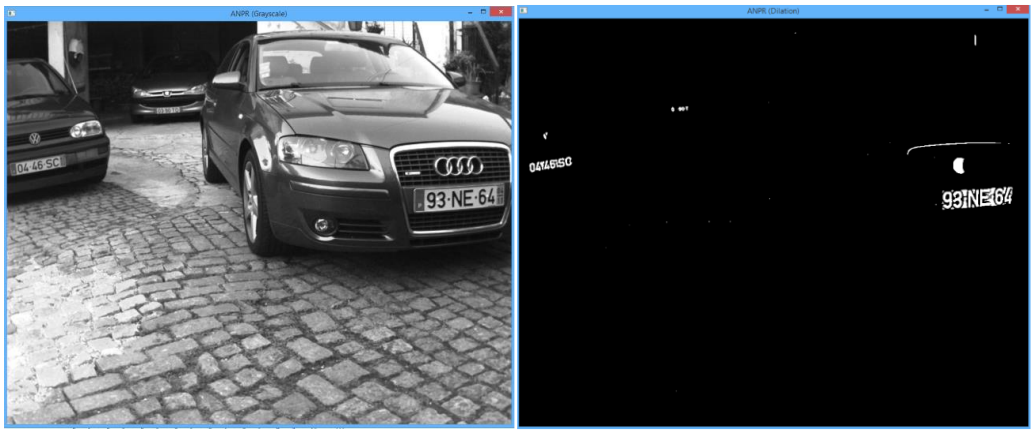


Figura 4.8.1.2 – Interface Gráfica, 3 carros em tons de cinzento e segmentada

4.8.2. *Threshold* local

A implementação do módulo de *Threshold* local, proposto na Figura 4.6.3.1, correu como previsto. O elevado nível de paralelismo do algoritmo, proporcionou bons resultados na aceleração em GPU. Durante o processo de implementação do *kernel*, foi necessário estabelecer um compromisso entre a escalabilidade do tamanho da janela da vizinhança, a otimização do tempo de execução e os recursos do *hardware*. A mesma resolução e tamanho da janela (*Wsize* = 9) foi utilizado no *profiling* dos diversos *kernels* apresentados.

Como é possível observar na Figura 4.8.2.1, para uma imagem com 1024x820 pixéis de resolução, o *kernel* demora 4,6 milissegundos a executar. Este valor pode ser considerado razoável, uma vez que o algoritmo proposto prepara ambas as imagens, marcador e máscara, necessárias pelo módulo de reconstrução morfológica. No entanto, através da análise da informação fornecida pela ferramenta de *profiling* Nsight, é possível verificar que o número de instruções por ciclo de relógio é de 0,74, como representado na Figura 4.8.2.2.

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread
1	kernel_thresh_fill	{820, 1, 1}	{1024, 1, 1}	341,292.685	4,614.368	100.00 %	14

Figura 4.8.2.1 – Profiling do kernel de Threshold Local

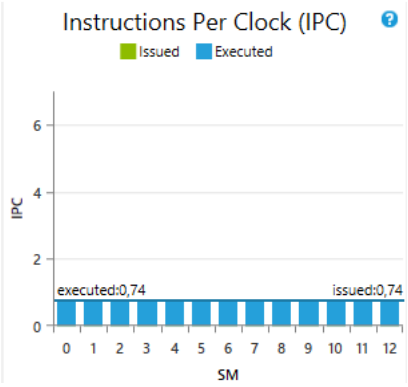


Figura 4.8.2.2 – Instructions Per Clock do kernel de Threshold Local

Na Figura 4.8.2.3 é possível verificar que o acesso à memória não é o *bottleneck* principal do *kernel*, mas sim a dependência de execução. O problema advém da escalabilidade que o valor da janela da vizinhança pode assumir. Ao permitir que a janela tome qualquer valor impar, como representado na Figura 4.8.2.4, quanto maior for o tamanho da vizinhança, maior vai ser a dependência de execução, uma vez que apenas uma instrução é executada por ciclo. Um compromisso deve então ser alcançado entre a escalabilidade da janela e a otimização dos recursos do sistema.

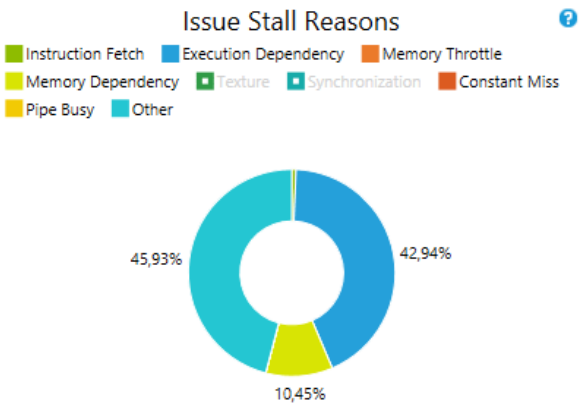


Figura 4.8.2.3 – Stall Reasons do kernel de Threshold local

```
for (int c_h = -ws; c_h <= ws; c_h++)
{
	for (int c_w = -ws; c_w <= ws; c_w++)
	{
		pos = (blockDim.x * (blockIdx.x + c_h) + (threadIdx.x + c_w));
		result = result + Grayscale[pos];
	}
}
```

Figura 4.8.2.4 – Ciclo que calcula o valor da vizinhança

A técnica de desenrolamento de ciclos (*Loop Unrolling*) tenta otimizar a execução dos ciclos, reduzindo a frequência das expressões condicionais (*branches*) e instruções de manutenção de ciclo [55]. Em vez de repetir uma expressão e correr um ciclo várias vezes, no desenrolar de ciclos, o número de operações por ciclo deve ser aumentado e o número de iterações do ciclo reduzido, ou mesmo removido. Esta técnica aumenta o desempenho no processamento sequencial de vetores, quando o número de iterações do ciclo é conhecido antes da execução. Como referido no capítulo 3, o tamanho da janela da vizinhança é um dos parâmetros de calibração do sistema, pelo que o número de ciclos não pode ser um valor fixo. No entanto, é possível criar diversos *kernels*, otimizados para determinados valores de tamanho da janela.

Na Figura 4.8.2.5 é apresentada uma solução que pode ser utilizada para valores ímpares e múltiplos de 3, para a janela de vizinhança. Esta é uma abordagem pertinente, pois demonstra alguma escalabilidade do algoritmo, otimizando o processo de cálculo dos valores da vizinhança, através da diminuição do número de iterações dos ciclos.

```
for (int c_h = -ws; c_h <= ws; y += 3)
{
        for (int c_w = -ws; c_w <= ws; x += 3)
        {
                pos1 = (blockDim.x * (blockIdx.x + c_h) + (threadIdx.x + c_w));
                pos2 = pos + blockDim.x;
                pos3 = pos + 2 * blockDim.x;
                result1 = result1 + Grayscale[pos1] + Grayscale[pos1 + 1] + Grayscale[pos1 + 2];
                result2 = result2 + Grayscale[pos2] + Grayscale[pos2 + 1] + Grayscale[pos2 + 2];
                result3 = result3 + Grayscale[pos3] + Grayscale[pos3 + 1] + Grayscale[pos3 + 2];
        }
}
```

Figura 4.8.2.5 – Desenrolar de ciclo em 3 por 3

Como é possível analisar na Figura 4.8.2.6, a expansão do ciclo em 3 por 3, aumentou muito o desempenho do *kernel*. O tempo de execução apresentado é de aproximadamente 1,7 milissegundos, que representa uma aceleração 2,7 vezes superior em relação à Figura 4.8.2.1. Como era de esperar, a dependência de memória aumenta e a dependência de execução diminuí, repartindo assim melhor os recursos do sistema. Na Figura 4.8.2.8 é possível verificar o aumento do número de instruções por ciclo de relógio para 1,07 e que a atividade dos *Streaming Multiprocessors* (SM) se encontra nos 96% de utilização, demonstrando um bom aproveitamento do *hardware* disponível por parte do *kernel*.

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread
1	unrolled_3x3_kernel_thresh_fill	{820, 1, 1}	{1024, 1, 1}	387,214.157	1,698.240	100.00 %	18

Figura 4.8.2.6 - Profiling do kernel de Threshold local com desenrolamento de ciclos de 3 por 3

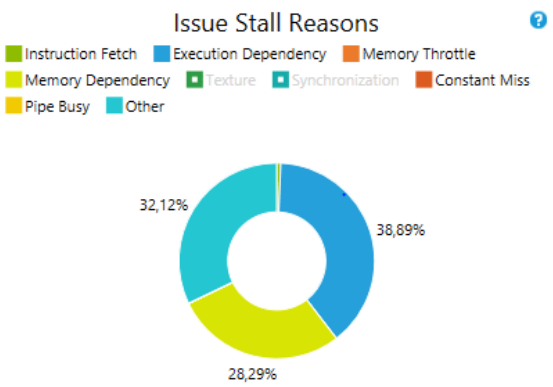


Figura 4.8.2.7 - Stall Reasons do kernel de Threshold local, desenrolado em 3 por 3

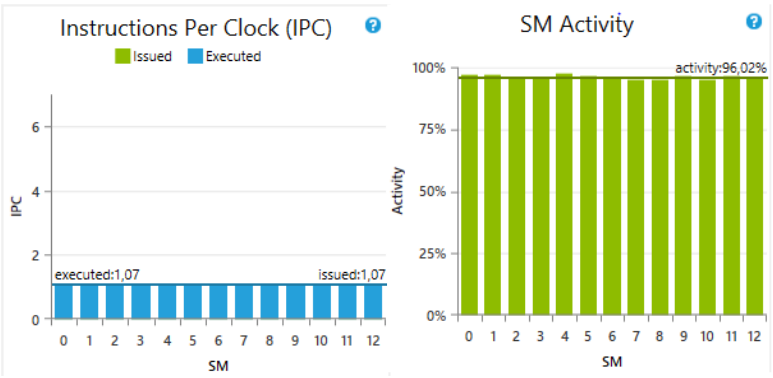


Figura 4.8.2.8 – Instruções por ciclo de relógio e atividade dos SM

Alternativamente, podem ser criados diversos *kernels* para valores da janela da vizinhança específicos, como o algoritmo representado na Figura 4.8.2.10, eliminando completamente os ciclos. Esta abordagem tira maior partido da capacidade do GPU, como pode ser observado na Figura 4.8.2.10, demonstrando um tempo de execução de 1,27 milissegundos, ou seja, 3,64 vezes mais rápido que o *kernel* da Figura 4.8.2.1 e 1,33 que o *kernel* da Figura 4.8.2.7.

```
pos_1 = (blockDim.x * (blockIdx.x - ws) + (threadIdx.x - ws));
.
.
.
pos_n = pos + (n-1) * blockDim.x;   // (n-1) <= ws
result_1 = Grayscale[pos_1] + ... + Grayscale[pos_1 + (n-1)];
.
.
.
result_n = Grayscale[pos_n] + ... + Grayscale[pos_n + (n-1)];
```

Figura 4.8.2.9 – Algoritmo para o cálculo do valor de vizinhança de tamanho n, desenrolado

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread
1	unrolled_kernel_thresh_fill	{820, 1, 1}	{1024, 1, 1}	426,726.509	1,268.064	100.00 %	24

Figura 4.8.2.10 – Profiling do kernel de Threshold Local, desenrolado

Através dos resultados de *profiling* realizados, é possível concluir que para uma melhor otimização do sistema, devem-se construir *kernels* sem ciclos, com os valores da janela da

vizinhança mais comuns. No entanto, a escalabilidade do algoritmo fica prejudicada com a predeterminação do tamanho da janela, pelo que a solução ideal passará pela utilização do *kernel* representado na Figura 4.8.2.5 para valores múltiplos de 3, e do *kernel* da Figura 4.6.3.1 para os restantes valores de janela que não se encontrem abrangidos pelas soluções anteriores.

4.8.3. Reconstrução Morfológica Paralela

A aceleração da implementação do algoritmo em GPU, proporcionou bons tempos de execução em cada processo de dilatação geodésica, como é possível conferir na Figura 4.8.3.1. Cada iteração do *kernel* demorou aproximadamente 72 µ segundos a concluir, para uma imagem de resolução 388x262. Os resultados continuam a ser favoráveis, quando se eleva a resolução da imagem analisada para 1024x820, apresentando em média 530 µ segundos por dilatação, como demonstra Figura 4.8.3.2.

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread
6	kernel_dilate	{262, 1, 1}	{388, 1, 1}	497,309.937	71.488	81.25 %	9
7	kernel_dilate	{262, 1, 1}	{388, 1, 1}	521,344.945	72.192	81.25 %	9
8	kernel_dilate	{262, 1, 1}	{388, 1, 1}	544,437.425	72.000	81.25 %	9

Figura 4.8.3.1 – Profiling da Reconstrução Morfológica Paralela de uma imagem de 388x262

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread
6	kernel_dilate	{820, 1, 1}	{1024, 1, 1}	414,470.488	524.927	100.00 %	9
7	kernel_dilate	{820, 1, 1}	{1024, 1, 1}	446,320.728	530.687	100.00 %	9
8	kernel_dilate	{820, 1, 1}	{1024, 1, 1}	478,249.911	526.944	100.00 %	9
9	kernel_dilate	{820, 1, 1}	{1024, 1, 1}	509,760.695	525.856	100.00 %	9
10	kernel_dilate	{820, 1, 1}	{1024, 1, 1}	545,764.791	527.840	100.00 %	9

Figura 4.8.3.2 – Profiling da Reconstrução Morfológica Paralela de uma imagem de 1024x820

No entanto, apesar do elevado paralelismo demonstrado pela metodologia, o número de iterações até atingir a estabilidade, ou pelo menos até obter bons resultados no processo de segmentação é muito incerto, o que inviabiliza a implementação da metodologia. Algumas imagens, com a mesma resolução, apresentam diferenças no número de iterações na casa das

centenas. Desta forma, a metodologia proposta nas Figura 4.6.1.1 e Figura 4.6.4.1 não podem ser consideradas alternativas viáveis à implementação.

4.8.4. Reconstrução Morfológica Sequencial

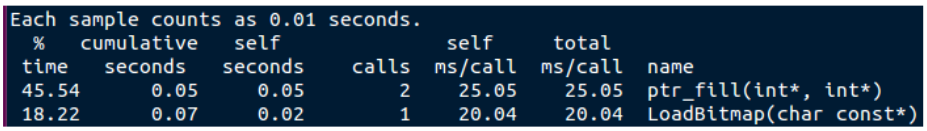
O processo de reconstrução morfológica sequencial, proposto na Figura 4.6.5.1 e Figura 4.6.5.2, demonstrou ser a opção mais robusta, uma vez que apresenta um número muito reduzido de iterações até atingir a estabilidade. Como referido anteriormente, em alguns casos, não é necessário atingir a estabilidade do processo de reconstrução, para se obter bons resultados no processo de segmentação. A primeira iteração do algoritmo é a mais demorada, uma vez que a Equação 4.6.5.2 adiciona uma condição, que não verifica a vizinhança dos pixéis com valor 1 na imagem marcador, ou seja, dos pixéis preenchidos nas iterações anteriores do algoritmo.

A otimização do algoritmo mostrou-se como um ponto muito importante, conseguindo diminuir em 50% o tempo de execução. O código foi otimizado através da análise das instruções em *assembly* geradas pela rotina e o *profiling* realizado utilizando a ferramenta *gprof*. Na Figura 4.8.4.1 é possível verificar que o algoritmo (*ptr_fill*) demorou cerca de 10 milissegundos a executar duas iterações, sobre uma imagem de resolução 1024x840. Devido à precisão da ferramenta de *profiling*, o teste foi repetido com uma imagem com o dobro da resolução (2048x1640), representada na Figura 4.8.4.2, onde o módulo de reconstrução morfológica demorou um pouco mais de 50 milissegundos a executar duas iterações.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.40      0.01     0.01        2     5.01     5.01  ptr_fill(int*, int*)
 33.40      0.02     0.01        1    10.02    10.02  LoadBitmap(char const*)
```

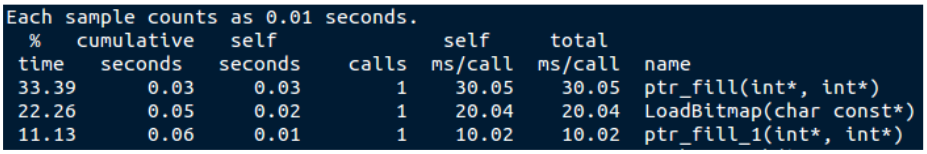
Figura 4.8.4.1 – Profiling do módulo de Reconstrução Morfológica Sequencial (1024x840).



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 45.54      0.05     0.05        2    25.05    25.05  ptr_fill(int*, int*)
 18.22      0.07     0.02        1    20.04    20.04  LoadBitmap(char const*)
```

Figura 4.8.4.2 – Profiling do módulo de Reconstrução Morfológica Sequencial (2048x1640)

Através da utilização de uma imagem de maior resolução, é possível obter resultados mais conclusivos relativamente ao comportamento do sistema. Na Figura 4.8.4.3, a Equação 4.6.5.2 foi utilizada na segunda iteração do algoritmo (*ptr_fill_1*). O resultado do *profiling* demonstra que o processo é cerca de 10 milissegundos mais rápido a executar para uma imagem de resolução 2048x1640.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.39      0.03     0.03        1    30.05    30.05  ptr_fill(int*, int*)
 22.26      0.05     0.02        1    20.04    20.04  LoadBitmap(char const*)
 11.13      0.06     0.01        1    10.02    10.02  ptr_fill_1(int*, int*)
```

Figura 4.8.4.3 – Profiling do módulo de Reconstrução Morfológica Sequencial EQ2 (2048x1640)

Os resultados obtidos, após a otimização do algoritmo, foram muito positivos e viabilizam a metodologia como um processo de segmentação válido para sistemas de tempo considerado real em aplicações de processamento de imagem, cerca de 20 fotogramas por segundo. A otimização introduzida pela Equação 4.6.5.2 poderá não ser considerada, dependendo da resolução da imagem a processar. O algoritmo demonstra bons resultados no processo de segmentação com apenas uma iteração, pelo que em certas aplicações a estabilidade do processo de reconstrução morfológica não será necessária, podendo assim ser estabelecido um compromisso entre a fiabilidade do sistema e o tempo de execução do sistema.

4.9. Discussão

O módulo de reconstrução morfológica paralelo (GPU), não pode ser considerado em futuras implementações do sistema, devido ao elevado número de iterações e imprevisibilidade do algoritmo. No entanto, a execução do processo de reconstrução morfológica sequencial no CPU não inviabilizada a aceleração das restantes rotinas em CUDA.

Na Figura 4.9.1 é possível verificar, que a cópia dos valores referentes às imagens entre a memória do CPU e do GPU, e vice-versa, demorou aproximadamente 3,7 milissegundos. O simples desenrolar dos ciclos da rotina de *Threshold* local, demonstra uma otimização no tempo de execução do *kernel* em 3,3 milissegundos, para uma imagem de resolução 1024x820. Assim, a aceleração do processo de *Threshold* local em CUDA, que cria ambas as imagens máscara e marcador, necessárias pelo módulo de reconstrução morfológica, apresenta um tempo de execução total inferior a 5 milissegundos, um resultado que pode ser considerado positivo. No entanto, é necessário alocar a memória necessária em GPU apenas no início da aplicação, visto ser um processo moroso. Como se pode verificar na Figura 4.9.1, o processo demorou aproximadamente 170 milissegundos a reservar o espaço necessário para as imagens em tons de cinza, *Threshold* e *Floodfill*.

	Name	Count	Capture Time %	# Errors	Total Time (µs)	Min (µs)	Avg (µs)	Max (µs)
1	cudaMalloc	3	6.56	0	170,696.167	386.185	56,898.722	169,890.690
2	cudaMemcpy	3	0.14	0	3,687.637	620.135	1,229.212	2,314.742

Figura 4.9.1 – Profilling das API calls mais relevantes do CUDA

O módulo de reconstrução morfológica sequencial demonstrou um tempo de execução de cerca de 10 milissegundos, para duas iterações do algoritmo. Assim, o processo de segmentação apresenta tempos de execução que variam entre os 15 e os 20 milissegundos, para uma imagem com 1024x820 de resolução, apresentando valores superiores a 50 fotogramas por segundo, numa execução sequencial. Este valor poderá ser inferior através da adição de um *pipeline*, que permite executar diversas operações em simultâneo. Este ponto é desenvolvido mais profundamente no capítulo 6.

O algoritmo reconstrução morfológica demonstra algum paralelismo ao nível da instrução, representado na Equação 4.6.5.1 e na Equação 4.6.5.2, o que o torna um bom candidato para implementação em FPGA. No entanto, a sequencialidade do algoritmo poderá condicionar a implementação do sistema. Os restantes módulos, do sistema proposto, apresentam um elevado nível de paralelismo, pelo que o sucesso da implementação em FPGA depende dos resultados obtidos pelo módulo de reconstrução morfológica.

5. FPGA

5.1. Introdução

Os *Field-Programmable Gate Arrays* (FPGAs) tornaram-se uma parte integral na implementação de circuitos digitais, devido à possibilidade de programar eletronicamente o silício pelo qual é composto, transformando-se em praticamente qualquer tipo de circuito digital [60]. As características de paralelismo apresentadas pela sua arquitetura fazem, com que o FPGA seja uma plataforma escolhida para acelerar aplicações computacionalmente intensivas, em diversas áreas como: processamento digital de sinal, visão por computador e imagem médica, criptografia, prototipagem de ASICs, virtualização, bioinformática e até mesmo áreas em crescimento, como eletrónica de consumo em veículos e "*gadgets*" [61].

A configuração do sistema é feita com recurso a uma *Hardware description language* (HDL), programando a estrutura, *design* e operações dos circuitos eletrónicos e lógicos. As ferramentas de *synthesis* atuais geram o *bitstream,* necessário para programar o FPGA através da interpretação das especificações em HDL (VHDL, Verilog HDL) [62].

Recorrendo a ferramentas de *High-level synthesis* (HLS) como o Vivado, torna-se possível a geração de código HDL, partindo de especificações em linguagens de mais alto nível, como o ANSI-C. Esta abordagem, não proporciona tanto controlo sobre o *workflow,* nem otimização do sistema, devido ao nível de abstração superior que apresenta.

Os SoCs da Altera e Xilinx integram um *ARM-based Hard Processor System* (HPS), constituído pelo processador, periféricos e interfaces de memória em conjunto com um FPGA, utilizando interligação com largura de banda elevada. Esta solução permite melhorar o desempenho do sistema, recorrendo aos *high-throughtput data paths* entre o HPS e o FPGA, sendo possível migrar as funções computacionalmente intensivas para *hardware* [63].

Neste capítulo é proposta, uma implementação do algoritmo de reconstrução morfologia em FPGA, uma vez que este módulo é considerado o mais critico para o sucesso da implementação do sistema proposto. Assim, uma imagem foi exportada da aplicação CUDA para um ficheiro .coe, de forma a ser carregado na memória interna do sistema e continuar o processo

de segmentação. Após a conclusão do processo de reconstrução morfológica em FPGA, os resultados são enviados para o processador ARM, através do barramento de dados AXI, que por sua vez envia os valores para um computador, através da porta série, de forma a depurar os resultados obtidos.

5.2. Desenho

5.2.1. Plataforma de desenvolvimento

A *Zybo Zynq-7000,* Figura 5.2.1.1, foi a plataforma selecionada, uma vez que permite comprovar a viabilidade do sistema proposto sem elevar os custos de desenvolvimento. Inclui um SoC composto por um processador *dual-core ARM Cortex A9* a 650 Mhz e uma FPGA equivalente à Artix-7 com 28 K *logic cells*, 240 KB *Block RAM*, 80 DSP *slices* e uma frequência operacional entre os 100 Mhz e os 250 MHz. Em termos de memória interna, o sistema apresenta um *MicroSD slot*, que permite a utilização de um sistema operativo Linux, e 512 MB x 32 DDR3 com um *bandwidth* de 1050 MBps e 8 canais de acesso direto à memória (DMA). A placa possuí também output de vídeo através das portas HDMI e VGA e ligação à rede através da porta *Ethernet*.

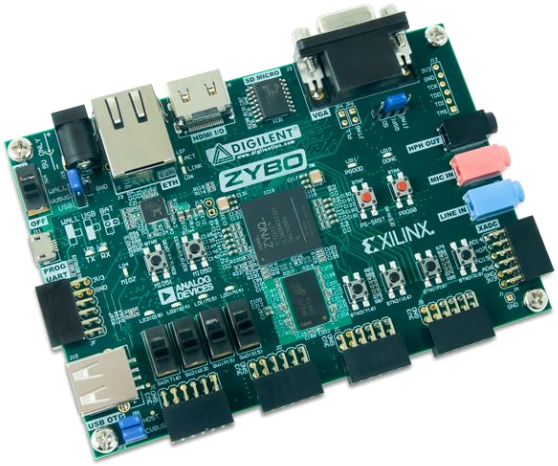


Figura 5.2.1.1 – Plataforma de desenvolvimento Zybo Zynq-7000

Enquanto o FPGA presente na *Zybo,* não apresenta a capacidade necessária para a implementação do sistema na sua totalidade, a sua arquitetura e *software* disponibilizados pela *Xilinx*, permitem um estudo aprofundado sobre a sua viabilidade e depuração de dados de uma forma relativamente acessível. A migração do sistema para uma placa de desenvolvimento com maior capacidade como a *ZedBoard,* não é um processo complexo uma vez que ambas se baseiam na mesma arquitetura.

5.2.2. Diagrama de blocos do sistema

O Sistema representado na Figura 5.2.2.1 é composto por dois blocos de memória (RAM e ROM), e o módulo de reconstrução morfológica, *ffill_0*. A imagem máscara encontra-se armazenada na ROM do sistema (*dist_mem_gen_0*), e a memória RAM (*dist_mem_gen_1*) armazena os valores intermédios e finais da imagem marcador. O significado das variáveis representadas pode ser consultado na Tabela 5.2.2.1.

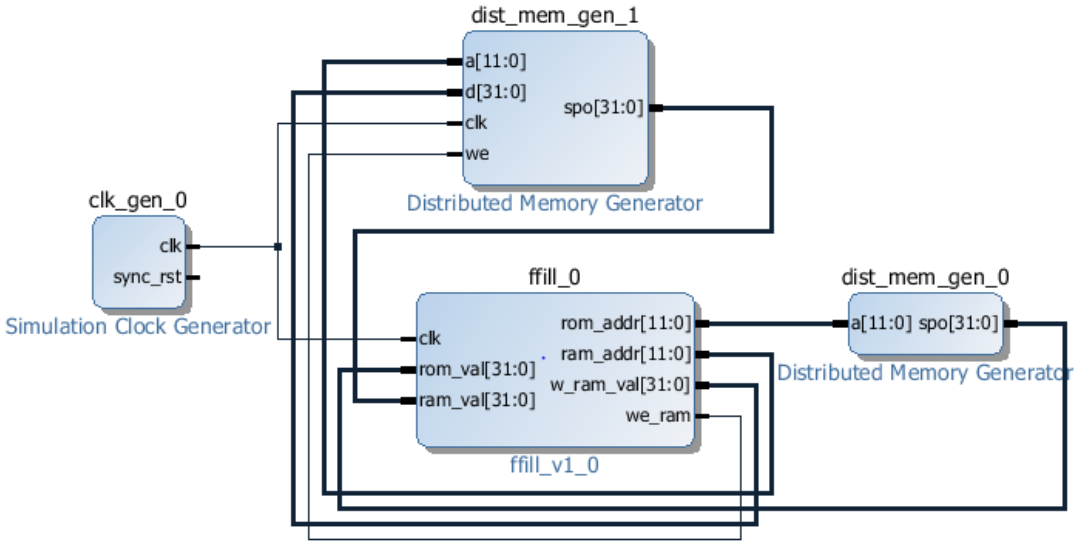


Figura 5.2.2.1 – Diagrama de blocos do Sistema

Variável	Significado	Dispositivo (Porta)	Entrada / Saída
clk	Frequência de operação do sistema	Todos (FFill,RAM)	E
rom_val	Valor de leitura da ROM	ROM(spo)	E
rom_addr	Endereço de leitura na ROM	ROM(a)	S
ram_val	Valor de leitura da RAM	RAM(spo)	E
w_ram_val	Valor a escrever na RAM	RAM(d)	S
ram_addr	Endereço de escrita/leitura da RAM	RAM(a)	S
we_ram	Habilita a escrita na memória	RAM(we)	S

Tabela 5.2.2.1 – Significado das variáveis do diagrama de blocos

De forma a ser possível testar o funcionamento do sistema, a partir do módulo de reconstrução morfológica, foi desenvolvida uma sub-rotina na aplicação CUDA, que exporta a informação referente à imagem após o processo de *Threshold* e a converte para o formato .coe, para que possa ser carregada para a memória ROM. Assim, a largura da imagem foi partida em diversos registos de 32 bits, facilitando o acesso à memória por parte do sistema. Como é possível verificar na Figura 5.2.2.2, cada pixel corresponde a um dígito na base numérica 2 e cada linha do ficheiro corresponde a um registo.

```
1    memory_initialization_radix=2;
2    memory_initialization_vector =
3    00000000000000000000000000000000,
4    00000000000000000000000000000000,
5    00000000000000000000000000000000,
6    00000000000000000000000000000000,
7    00000000000000000000000000000000,
8    00000000000000000000000000000000,
9    00000000000000000000000000000000,
10   00000000000000000000000000000000,
11   00000000000000000000000000000000,
12   00000000000000000000000000000000,
13   00000000000000000000000000000000,
14   00000000000000000000000000000000,
15   00000000000000000000000000000000,
16   00000000000000000000000000000000,
17   00000000000000000000000000000000,
18   00000000000000000000000000000000,
19   00000000000000000000000000000000,
20   00000000000000000000000000000000,
21   00000000000000000000000000000000,
22   00000000000000000000000000000000,
23   00000000000000000000000000000000,
24   00000000000000000000000000000000,
25   00000000000000000000000000000000,
```

```
1183 11111111100000000000000000000000,
1184 00000000000000000000000001000001,
1185 10000000000000010000000000000000,
1186 00000000100111100111111111111100,
1187 00000000011111100000000000000000,
1188 00000000000000000000000000000000,
1189 00000000000000000001111110000000,
1190 00000000000000000000001111000000,
1191 00000000010001100000000000000000,
1192 00100000000100000000000000000000,
1193 00000000000000000000000000000000,
1194 00000000000000000000000000000000,
1195 00000001111111111110000000000000,
1196 00000000000000000000000000100000,
1197 00100000000001000000000000000000,
1198 00000000010111111111111000000000,
1199 00011111100000000000000000000000,
1200 00000000000000000000000000000000,
1201 00000000000000011111100000000000,
1202 00000000000000000000001000000010,
1203 00000000000010100000000000000000,
1204 00010000000001100000000000000000,
1205 00000000000000000000000000000000,
1206 00000000000000000000000000000000,
1207 00000000000000001111111111111100,
```

Figura 5.2.2.2 – Formato do ficheiro .coe

Como descrito no capítulo 3, o processo de reconstrução morfológica necessita de duas imagens, a máscara e o marcador. O inverso da imagem presente na memória ROM corresponde à imagem máscara. A imagem marcador, numa fase inicial consiste num fundo preto com bordas brancas, pelo que não é necessário guardar os valores em memória, poupando assim ciclos de processamento. A Tabela 5.2.2.2 demonstra a representação binária de uma imagem marcador, o princípio base desta imagem, é que as suas margens devem ser colocadas a um, deixando o resto da imagem com o valor zero. A primeira e a última linha da imagem podem ser escritas diretamente na memória RAM, quando o módulo começar o processamento.

1	1	1	1	1	1	1
1	0	0	0	0	0	1
1	0	0	0	0	0	1
1	0	0	0	0	0	1
1	0	0	0	0	0	1
1	0	0	0	0	0	1
1	0	0	0	0	0	1
1	1	1	1	1	1	1

Tabela 5.2.2.2 – Representação binária de uma imagem marcador

Uma variável denominada *marker_canvas,* foi utilizada para resolver os restantes casos, onde o primeiro e o último pixel de cada linha são colocados a um, e os remanescentes mantêm o valor de zero. Como é possível verificar no fluxograma presente na Figura 5.2.2.3, o valor a processar do marcador depende do registo a ser processado atualmente (*reg_pos*). Assim, o primeiro registo de cada linha tem um valor decimal $2^{reg_{size}-1}$, onde reg_{size} corresponde número de bits por registo, e o último o valor decimal 1, sendo que todos os registos intermédios são colocados a zero. Após a conclusão do cálculo do primeiro registo, do módulo de reconstrução morfológica, o resultado é guardado na memória RAM, no entanto, o facto de não necessitar de reserva prévia em memória do valor do marcador, permite ao sistema poupar *Height* Width* ciclos de processamento.

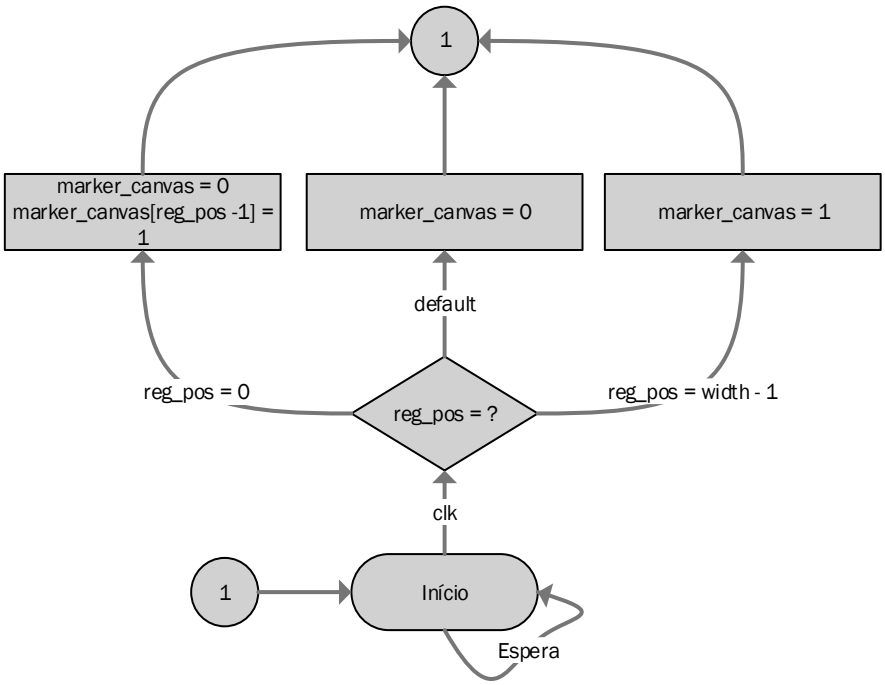


Figura 5.2.2.3 – Marker Canvas

5.2.3. Módulo de Reconstrução Morfológica

Os resultados obtidos na implementação em CUDA do sistema, permitem concluir que o algoritmo de reconstrução paralelo, baseado em dilatações geodésicas, mesmo apresentando um elevado nível de paralelismo, ideal para implementações em FPGA e GPU, apresenta uma grande variabilidade no número de iterações até que a estabilidade seja atingida, criando uma instabilidade no desempenho do sistema e não apresentando resultados superiores a outras abordagens.

Contrariamente, o algoritmo de reconstrução sequencial geralmente demonstrou bons resultados após a primeira iteração, atingindo a estabilidade durante a segunda iteração em grande parte dos testes efetuados. O estudo e implementação deste módulo em FPGA é considerado muito pertinente, devido às características sequenciais apresentadas pelo algoritmo. Como a Figura 4.6.1 demonstra, o algoritmo de reconstrução morfológica não apresenta características ideais para aceleração em CPU, devido ao elevado tamanho de dados, apresentado pelas imagens, nem em GPU pois o algoritmo é sequencial. A viabilidade da implementação do sistema proposto em FPGA é posta em causa pelo sucesso e resultados apresentados pelo módulo

de reconstrução morfológica, pelo que todo o trabalho realizado neste capítulo, incide neste ponto e na comunicação com o barramento de dados AXI, de forma a ser possível depurar os resultados obtidos.

O módulo de reconstrução morfológica funciona sobre dois princípios: o posicionamento do registo a processar em relação à sua posição na imagem e o carregamento de dados, é efetuado na transição ascendente do ciclo do relógio. O processamento e armazenamento de dados é efetuado, na transição descendente do ciclo de relógio. Desta forma, é possível assegurar que a variável de posição se encontra na posição correta, antes de começar o processamento. O sistema é composto, essencialmente, por quatro máquinas de estados: contador de posição, processamento de dados, acesso e controlo sobre a memória RAM e controlo de armazenamento.

5.2.3.1. Contador de posição

O objetivo da máquina de estados do contador de posição é percorrer a imagem de cima a baixo, ignorando a primeira e ultima linha/coluna, em *Raster Order.* Após o término da operação, a imagem é percorrida de baixo para cima, continuando a ignorar a primeira e ultima linha/coluna, em *Anti-Raster Order*. Como é possível observar na Figura 5.2.3.1.1, a direção pela qual a imagem é percorrida é definida pelo *fwd_clk (forward clock)* ou *rev_clk (reverse clock)*, voltando sempre ao estado inicial, após a conclusão do processamento no estado, ficando à espera do próximo sinal de relógio.

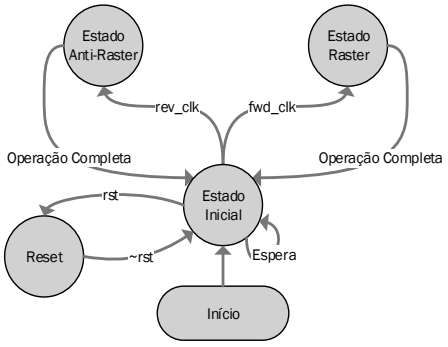


Figura 5.2.3.1.1 – Máquina de estados do contador de posição

Esta máquina de estados, determina qual o pixel, registo e endereço a ser processado atualmente, pelo que executa no ciclo ascendente do sinal de relógico, garantindo que a informação referente ao pixel a processar, e a sua posição atual na imagem esteja disponíveis a quando do processamento dos dados. Na Tabela 5.2.3.1.1, é explicado em detalhe o significado de cada variável utilizada na representação dos fluxogramas e máquinas de estado, referentes ao contador de posição.

Variável	Significado
Initial State	Estado inicial.
wait	Modo de espera.
fwd_clk	Sinal de relógio da operação *Raster.*
rev_clk	Sinal de relógio da operação *Anti-Raster.*
rst	*Reset* ativo.
pos	Posição do bit atual no registo.
reg_pos	Registo em uso.
reg_size	Tamanho do registo.
reg_addr	Posição do registo atual em memória.
width	Largura da imagem (expressa em número de registos).
height	Altura da imagem.
size	Tamanho total da imagem (expressa em *width x height).*
reverse	Altera o modo de contagem de *Raster* para *Anti-Raster.*

Tabela 5.2.3.1.1 – Descrição das variáveis utilizadas

O fluxograma que representa o funcionamento do estado *Raster,* da máquina de estados da Figura 5.2.3.1.1, pode ser observado na Figura 5.2.3.1.2. Como descrito anteriormente, o objetivo é percorrer todos os pixéis, da imagem marcador, de cima a baixo em *Raster Order,* ignorando a primeira linha/coluna. Assim, as condições sobre as quais a posição da imagem,

registo e endereço devem ser alteradas, são descritas no fluxograma. Na Figura 5.2.3.1.3, é possível verificar o comportamento do sistema, durante o estado *Anti-Raster*, demonstrando uma lógica essencialmente complementar, da apresentada durante o *Raster Order*.

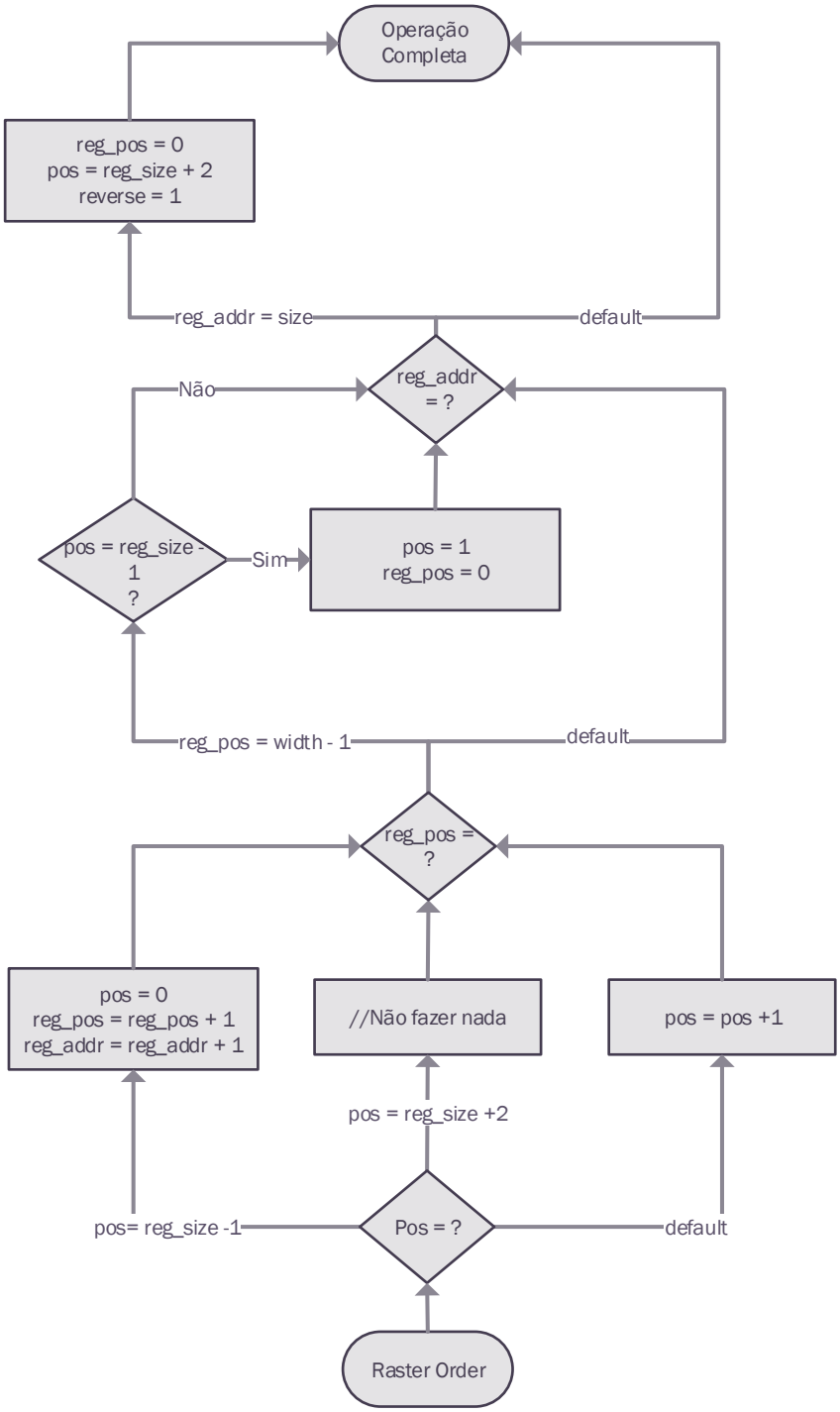


Figura 5.2.3.1.2 – Estado Raster

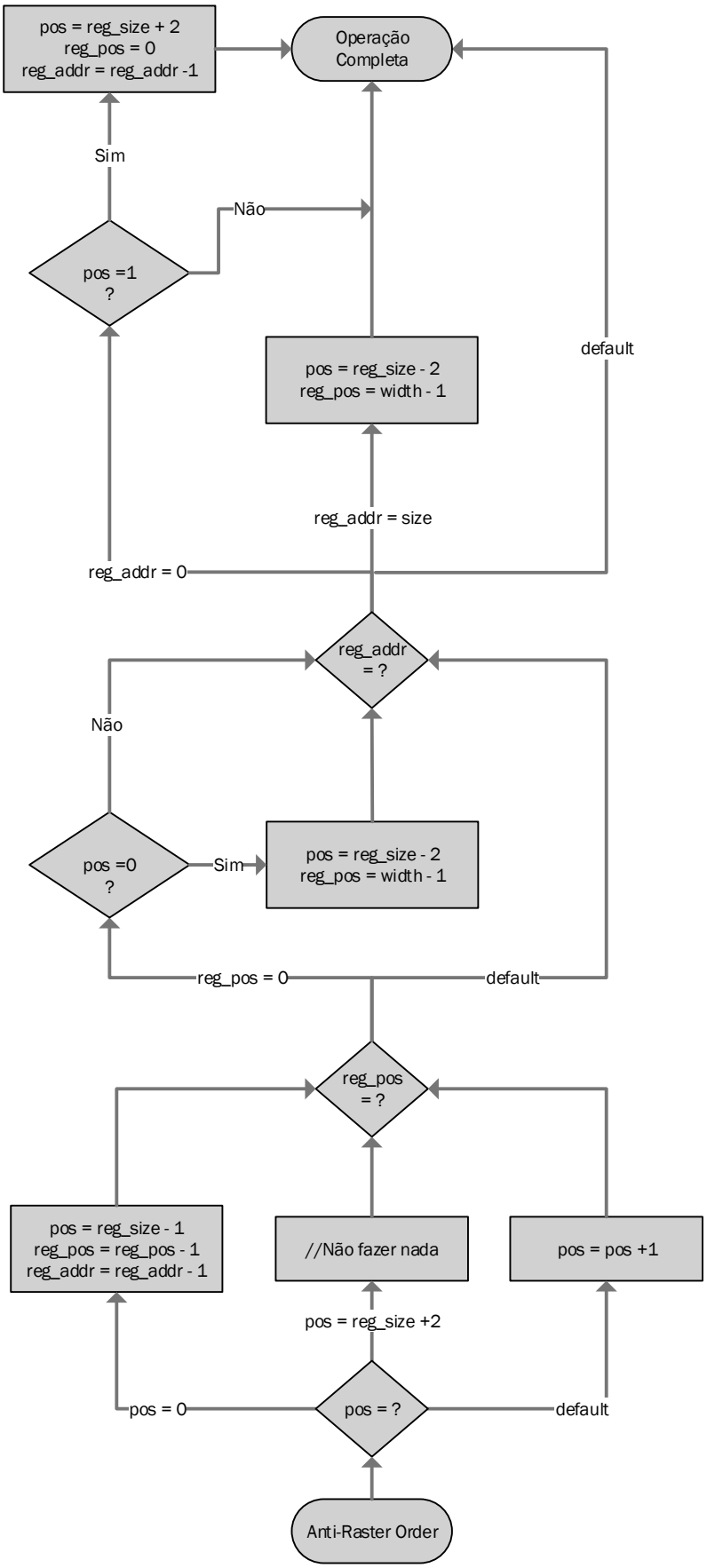


Figura 5.2.3.1.3 – Estado Anti-Raster

Finalmente, o estado de *reset* é ativado, quando o botão for ativado. O objetivo deste estado, perante o contador de posição, é limpar os valores de contagem atuais e atribuir os valores iniciais. Como representado na Figura 5.2.3.1.4, o valor de posição deve ser colocado a 1, e o endereço de memória deve tomar o primeiro valor da segunda linha, uma vez que as margens da imagem não podem ser processadas pelo algoritmo.

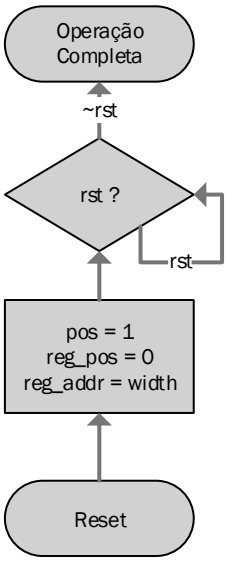


Figura 5.2.3.1.4 – Reset state, perante o contador de posição

5.2.3.2. Processamento de dados

O intuito desta máquina de estados é o processamento dos dados, relativamente à operação de reconstrução morfológica. Executa na transição descendente do ciclo do relógio, após a determinação da posição do pixel atual no registo e na memória, realizado pela máquina de estados de contador de posição, na transição ascendente do relógio. Como é possível analisar na Figura 5.2.3.2.1, não são necessários os dois estados de processamento *Raster* e *Anti-Raster,* uma vez que o algoritmo apresenta simetria, relativamente à sua posição na imagem. Não necessita de um estado de *Reset*, uma vez que os resultados obtidos apenas são guardados em variáveis temporárias, para mais tarde serem guardados em memória.

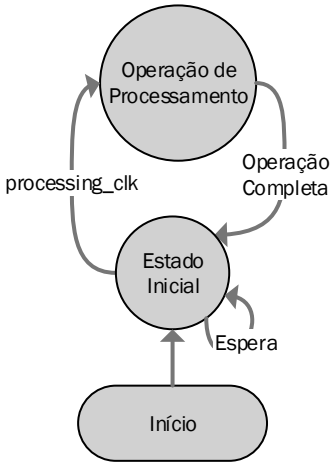


Figura 5.2.3.2.1 – Máquina de estados do processamento de dados

Três registos são necessários para efetuar o cálculo, o registo de marcador atual, o registo atual que contém o valor da ROM (máscara) e o registo do marcador anterior, onde a sua posição em memória, pode ser definida pela expressão $reg\_pos - width$. Antes de chegar ao último bit do registo atual, é necessário carregar os próximos valores de marcador. Assim, a Figura 5.2.3.2.2 representa os registos necessários da imagem marcador, na operação de reconstrução morfológica. Na Tabela 5.2.3.2.1, são apresentadas as diversas variáveis utilizadas, e o seu significado durante a operação de processamento de dados.

Variável	Significado
curr_marker[reg_pos][m_pos]	Corresponde ao registo e bit do marcador atual a ser processado.
prev_marker[reg_pos][m_pos]	Corresponde à linha anterior (*Raster*) ou seguinte (*Anti-Raster*) da imagem marcador.
tmp[]	Variável temporária. Utilizada na transição de informação entre registos.
rom_val[m_pos]	Bit da memória ROM que corresponde à posição atual.
m_pos	Posição no bit do registo atual (*Marker_pos*).

Tabela 5.2.3.2.1 – Descrição das variáveis utilizadas no Processamento de dados

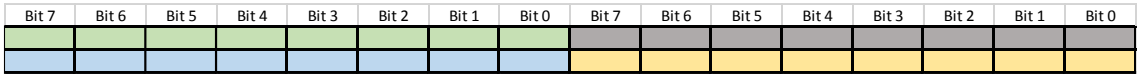


Figura 5.2.3.2.2 -– Registos necessários da imagem marcador (RAM)

De forma a ser possível executar o processamento de uma forma simétrica, uma variável denominada *m_pos* (*Marker Position*) é criada, com o intuito de alterar a posição no registo consoante o modo de ordenação da imagem. A Equação 5.2.3.2.1 permite então, utilizar a mesma variável durante o processo de ordenação em *Raster* e *Anti-Raster*.

$$\text{m\_pos} = (\text{reg\_size} - 1) * \text{forward} - \text{pos}$$

$$Dir = Reverse - Forward$$

Equação 5.2.3.2.1 – Variável de ordenação comum a ambas as ordenações

A posição do marcador atual e anterior, em relação à imagem, alterna consoante a ordenação pela qual a imagem é percorrida. Na Figura 5.2.3.2.3 é possível observar, o posicionamento dos registos durante as operações em *Raster Order,* e na Figura 5.2.3.2.4 em *Anti-Raster Order*. A simetria entre operações permite a reutilização dos registos para ambas as operações, mudando apenas a sua posição perante a imagem.

prev_marker[reg_pos]	prev_marker[reg_pos+1]
curr_marker[reg_pos]	curr_marker[reg_pos+1]

Figura 5.2.3.2.3 – Registos da imagem marcador em Raster Order

curr_marker[reg_pos-1]	curr_marker[reg_pos]
prev_marker[reg_pos-1]	prev_marker[reg_pos]

Figura 5.2.3.2.4 – Registos da imagem marcador em Anti-Raster Order

Na Figura 5.2.3.2.5 são apresentados, os bits necessários ao processo, a vermelho, e o primeiro bit, representado por *m_pos,* a laranja, processado em *Raster Order*. Como referido

anteriormente, a primeira linha e coluna não são processadas, uma vez que o algoritmo necessita do valor do pixel anterior. Assim, são apresentados os *offsets,* relativamente à variável *m_pos*, no processo de reconstrução morfológica.

Bit 7	Bit 6	Bit 5	Bit 4
1	0	-1	
1	m_pos		

Figura 5.2.3.2.5 – Primeiro bit processado em Raster Order

Na Figura 5.2.3.2.6 o princípio é o mesmo, no entanto, a operação é realizada em *Anti-Raster Order.* Neste caso, a última coluna e a última linha não podem ser processadas, uma vez que o algoritmo necessita de informação relativa à posição seguinte.

Bit 3	Bit 2	Bit 1	Bit 0
		m_pos	-1
	1	0	-1

Figura 5.2.3.2.6 – Primeiro bit processado em Anti-Raster Order

A simetria entre ambas as operações pode mais uma vez ser observada, pelo que a Equação 5.2.3.2.2, a qual dá o valor referente ao pixel m*_pos,* pode ser estabelecido para ambas as ordenações, onde *CM* corresponde a *curr_marker[reg_pos]* e *PM* a *prev_marker[reg_pos]*.

$$\text{MK\_tmp} = \text{CM[m\_pos]} \,\&\, \text{CM[m\_pos-Dir]} \,\&\, \text{PM[m\_pos-Dir]} \,\&\, \text{PM[m\_pos]} \,\&\, \text{PM[m\_pos+Dir]}$$

$$\text{CM[pos]} = \text{MK\_tmp} \mid \sim\text{rom\_val[m\_pos]}$$

Equação 5.2.3.2.2 – Cálculo do valor de Marker no pixel m_pos (Operação 1)

O cálculo de valor do marcador, no primeiro e último bit de cada registo, necessita de um estado especial, de forma a realizar a junção dos valores de ambos os registos. Como demonstrado na Figura 5.2.3.2.7, o valor do bit 7 (bit 0 em *Anti-Raster Order*) do *prev_marker* do registo

seguinte, é necessário para concluir a operação. Seja *PM_Dir*, prev_marker[reg_pos-Dir], a Equação 5.2.3.2.3 expressa o calculo do valor de marcador, no ultimo bit do registo.

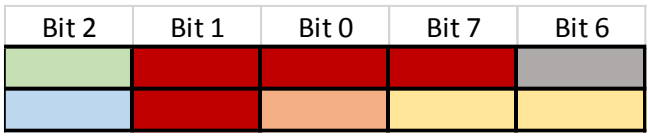


Figura 5.2.3.2.7 – Cálculo do valor de marcador no último bit do registo

$$MK\_tmp = CM[m\_pos] \,\&\, CM[m\_pos\text{-}Dir] \,\&\, PM[m\_pos\text{-}Dir] \,\&\, PM[m\_pos] \,\&\, PM\_Dir[m\_pos+Dir]$$

$$CM[m\_pos] = MK\_tmp \mid \sim rom\_val[m\_pos]$$

Equação 5.2.3.2.3 – Cálculo do valor de marcador no último bit do registo (Operação 2)

A próxima operação necessita do valor do último bit, do marcador atual e anterior, pelo que a melhor solução é guardar os seus valores numa variável temporária, como demonstra a Figura 5.2.3.2.8. Desta forma os valores da variável *tmp,* podem ser substituídos na Equação 5.2.3.2.2, para o cálculo do valor do marcador no bit 7 (bit 0 em *Anti-Raster Order*) do registo seguinte, como demonstra a Equação 5.2.3.2.3. A operação 3 pode ser analisada mais pormenorizadamente, na Figura 5.2.3.2.9, no que diz respeito ao *Raster order,* e na Figura 5.2.3.2.10 em *Anti-Raster Order*.

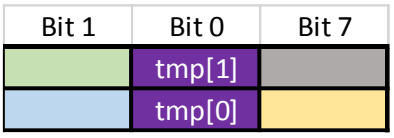


Figura 5.2.3.2.8 – Guardar último bit do registo marcador atual e anterior

$$MK\_tmp = CM[m\_pos] \,\&\, tmp[0] \,\&\, tmp[1] \,\&\, PM[m\_pos] \,\&\, PM[m\_pos+Dir]$$

$$CM[m\_pos] = MK\_tmp \mid \sim rom\_val[m\_pos]$$

Equação 5.2.3.2.4 – Cálculo do valor do marcador no primeiro bit do registo (Operação 3)

Bit 1	Bit 0	Bit 7	Bit 6	Bit 5
	tmp[1]	0	-1	
	tmp[0]	m_pos		

Figura 5.2.3.2.9 – Operação 3, Raster Order

Bit 2	Bit 1	Bit 0	Bit 7	Bit 6
		m_pos	tmp[0]	
	1	0	tmp[1]	

Figura 5.2.3.2.10 – Operação 3, Anti-Raster Order

Esta metodologia proposta, que retira partido da simetria das operações de ordenação, permite simplificar e consolidar o processo, e diminuir os recursos necessários à sua implementação. O bloco de decisão presente na Figura 5.2.3.2.11 é considerado, o segundo mais importante do módulo de reconstrução morfológica, apenas superado pelo contador de posição.

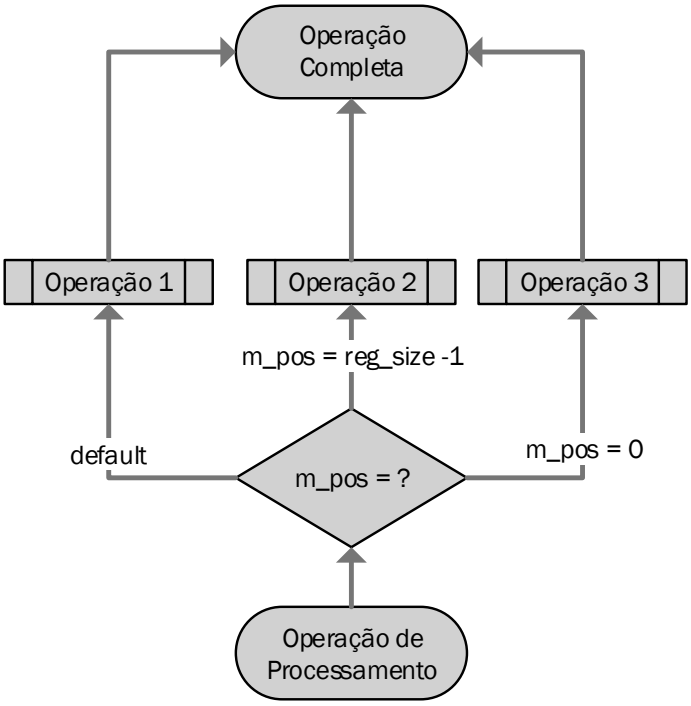


Figura 5.2.3.2.11 – Fluxograma do estado de Processing Operation

5.2.3.3. Controlo de Armazenamento

A próxima máquina de estados, do módulo de reconstrução morfológica, é o controlo de armazenamento. O objetivo do módulo é determinar quando o processamento do registo marcador é concluído, determinar a ordenação pela qual a imagem está a ser percorrida, guardar o valor pretendido numa variável temporária, e sinalizar o módulo de controlo da memória RAM, para que o resultado possa ser guardado. Assim, a Figura 5.2.3.3.1 descreve o processo relacionado com o pedido de escrita de um registo em memória.

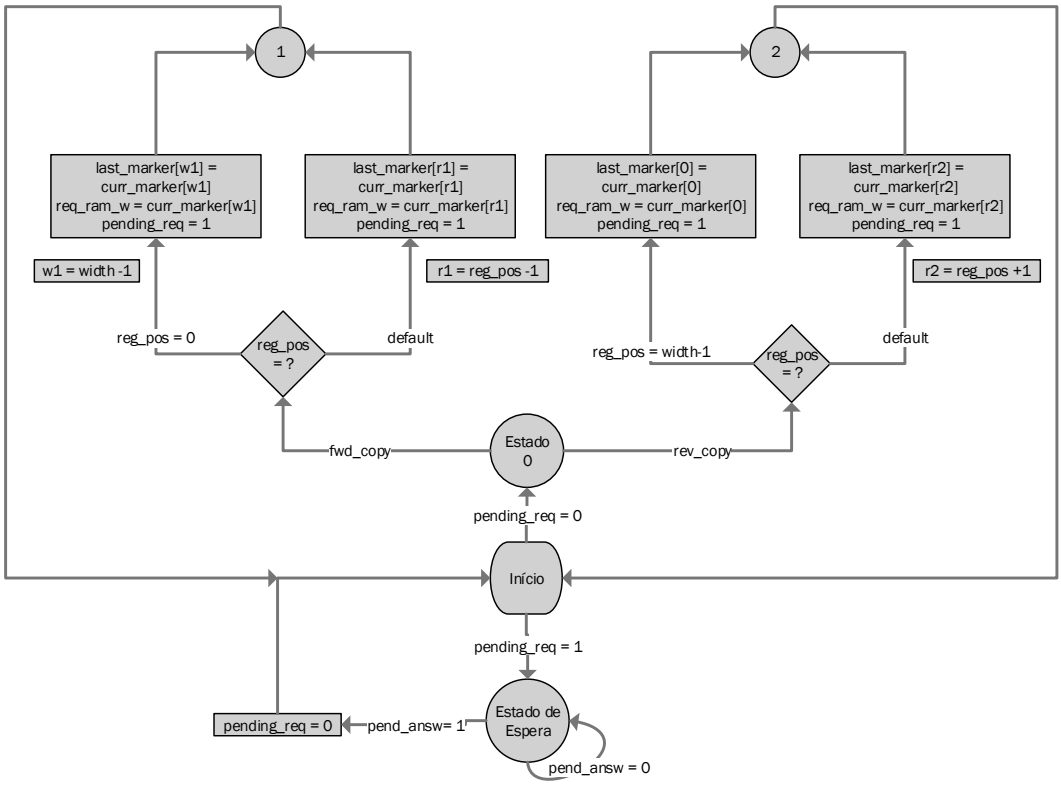


Figura 5.2.3.3.1 – Fluxograma referente ao controlo de armazenamento

O significado de algumas das variáveis utilizadas não foi explicado anteriormente, pelo que a consulta da Tabela 5.2.3.3.1, em conjunto com as apresentadas anteriormente ajuda na compreensão do algoritmo.

Variável	Significado
pending_req (*Pending Request*)	Pedido de escrita na memória RAM
pend_answ (Pending Request Answered)	Resposta ao pedido de escrita em memória
req_ram_w (Request RAM Write)	Pedido de escrita do registo

Tabela 5.2.3.3.1 – Descrição das variáveis utilizadas no módulo de controlo de armazenamento

O módulo apresenta dois principais estados: o primeiro fica à espera da sinalização do relógio, para efetuar o pedido de cópia da variável para a memória, o segundo fica à espera que o pedido de acesso à memória seja atendido. A sinalização do relógio indica a ordenação, pela qual o sistema está a percorrer a imagem, *Raster* ou *Anti-Raster*.

O bloco de decisão, relativamente à posição do registo, determina qual das duas opções executa. O princípio da posição *default* é que o registo atual, se encontra na mesma linha do anterior (que é guardado em memória), pelo que apenas é necessário decrementar a posição do registo (*Raster Order*) ou incrementar o Registo (*Anti-Raster Order*), em relação ao registo atual, para obter a posição do registo a guardar em memória. O valor é então colocado na variável temporária *req_ram_w,* e a *flag pend_req* sinaliza o módulo de controlo de memória, que é necessário escrever o valor. As situações especiais situam-se, no primeiro registo de cada linha (Raster Order) ou no último (*Anti-Raster Order*). Neste caso, é necessário guardar o valor do último e primeiro registo da linha anterior, respetivamente. Como na posição *default,* o valor do registo a escrever, é guardado no registo temporário e a *flag* que realiza o pedido de escrita é ativada.

Após um pedido de escrita em memória, a máquina de estados é colocada em modo de espera, enquanto o pedido não é atendido. Quando o módulo de controlo de memória termina o processo de escrita, coloca a *flag req_answ* com o valor de um, sinalizando que o processo de escrita foi concluído, colocando a *flag* de *pend_req* a zero e voltando ao estado zero, até à próxima sinalização do relógio.

5.2.3.4. Controlo da memória RAM

O módulo de controlo da memória RAM é responsável por todas as operações de escrita na memória, por parte do módulo de reconstrução morfológica. Como tal, a atribuição de vários estados é necessário para responder a todas as necessidades do sistema. Na Figura 5.2.3.4.1 é possível analisar os diversos estados, bem como a lógica de funcionamento do módulo. Como o módulo de controlo de armazenamento e de processamento de dados, a sua execução é realizada na transição descendente do relógio.

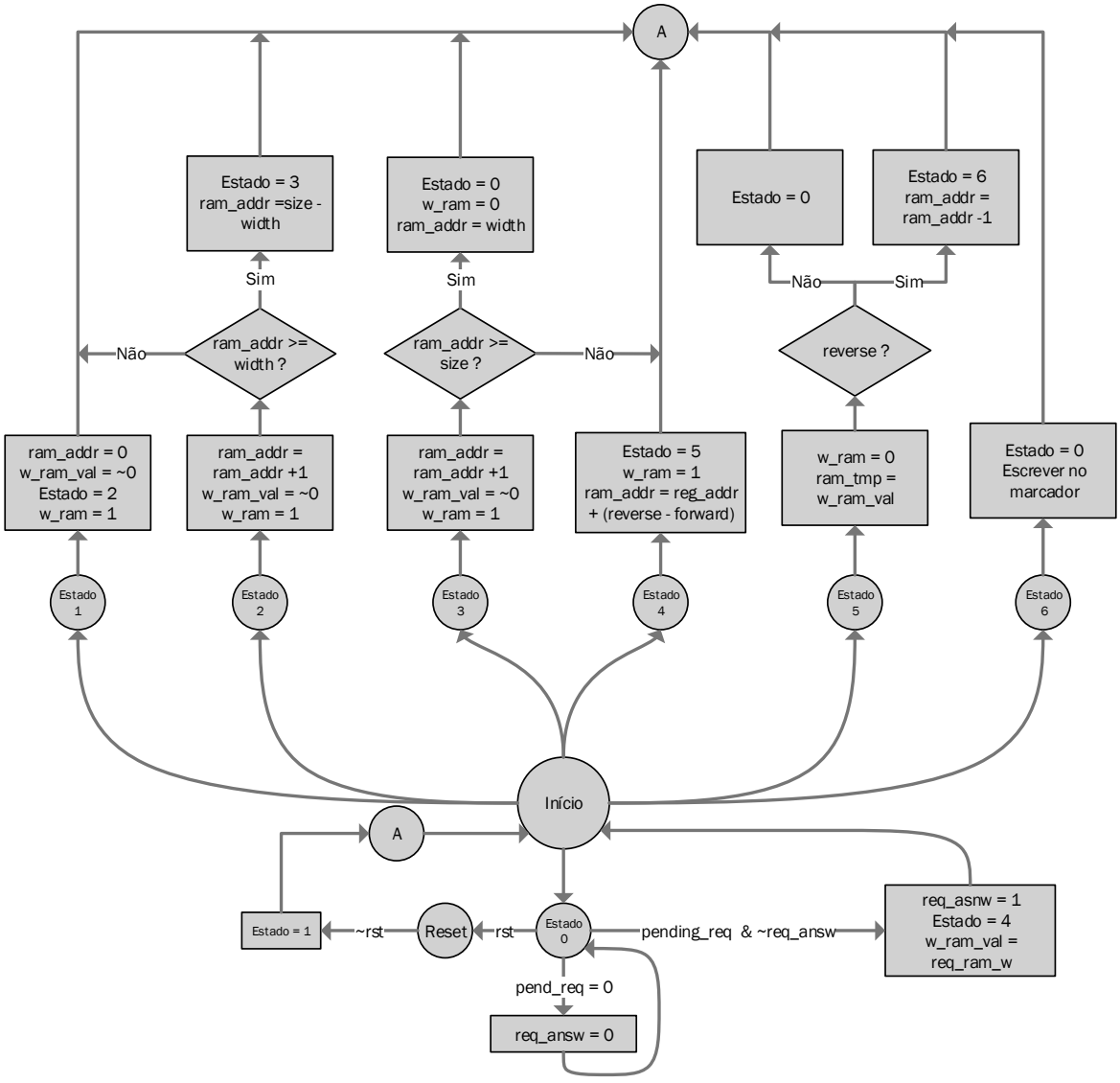


Figura 5.2.3.4.1 – Fluxograma referente ao controlo de escrita em memória RAM

Os três primeiros estados do sistema (1, 2 e 3) têm como principal objetivo, preencher a primeira e última linha da imagem em memória com o valor de 1 em todos os bits do registo. Assim, o módulo inicia no estado 1, que coloca o endereço da memória RAM no primeiro valor, ativa a escrita, atribuí um registo com todos os bits a 1, e altera o estado do sistema para o segundo. Este estado prepara a posição e escrita na memória, enquanto o segundo continua o processo de escrita, incrementando o endereço de posição, até atingir o último registo da linha (*width* - 1). Quando o último registo é escrito, o estado é alterado para o terceiro, situando o endereço da memória, na última linha da imagem (*size – width*). O comportamento é similar ao do segundo estado, onde o objetivo é preencher todos os valores dos registos, da última linha da imagem com o valor de um. Quando o endereço da memória atinge um valor igual ao tamanho da imagem (*reg_addr >= size*), a escrita em memória é desativada (*w_ram = 0*) e o endereço é colocado no primeiro registo da segunda linha da imagem.

Após o processo inicial, o sistema fica à espera, no estado 0, de um pedido de escrita por parte do módulo de controlo de armazenamento. Quando um pedido é executado e não existe resposta pendente, o sistema escreve o valor temporário (*req_ram_w*) na entrada da porta RAM, sinaliza o módulo de controlo que o pedido foi atendido e muda o estado do sistema para o quarto.

O quarto estado começa por posicionar o endereço a escrever e habilitar a escrita em memória, avançando em seguida para o estado 5. O objetivo deste estado, que executa um ciclo de relógio após o anterior, é desligar a escrita em memória, uma vez que a operação já foi concluída, e decidir se deve regressar ao estado de espera zero (*Raster Order*), ou ao sexto estado (*Anti-Raster Order*) e guardar o valor escrito num registo temporário, de forma a confirmar o sucesso da operação. Se o sistema processar os dados em *Anti-Raster Order,* o quinto estado decrementa o endereço da memória RAM e passa para o estado 6, que envia o valor do registo lido da memória para o registo de marcador, a processar após a conclusão do atual, permitindo que o sistema tenha sempre o próximo registo a processar disponível. Após o término da operação volta ao estado zero e espera pelo próximo pedido.

Por final o estado de *Reset*, coloca o sistema de volta no estado um, quando o seu valor voltar a zero, reiniciando assim o processo de escrita em memória (primeira e última linha apenas). Uma vez que a primeira iteração do algoritmo (*Raster Order*), não necessita de informação armazenada na memória RAM, não é necessário limpar a memória após o estado de *reset*, uma

vez que esta vai ser reescrita, antes de voltar a ser consultada na operação de reconstrução morfológica.

5.3. Implementação

O processo de implementação do sistema pode ser dividido em duas partes distintas, a primeira, composta pelo sistema representado na Figura 5.2.2.1, onde todos os resultados são confirmados através da ferramenta de simulação do *Vivado*. O sistema foi implementado sobre a HDL verilog, utilizando 32 bits por registo, onde cada linha é composta por 12 registos. A Imagem utilizada apresenta uma resolução de 384x260 pixéis.

Posteriormente, o sistema é ligado ao *ZYNQ7 Processing System* através do barramento *AXI-4* (em modo simples), de forma a ser possível confirmar os resultados da simulação em *hardware*. Este tópico é desenvolvido na secção de resultados obtidos. Em ambos os casos foi utilizada a frequência de operação máxima permitida, 250 MHz, pela placa de desenvolvimento *Zybo*.

5.3.1. Contador de posição

Na Figura 5.3.1.1 é possível analisar o início do processo de contagem, que percorre a imagem de cima para baixo em *Raster Order* e de baixo para cima em *Anti-Raster Order*. O registo *counter* é incrementado, em todas as transições ascendentes do relógio, servindo como o contador do número total de operações realizadas pelo módulo. A variável de posição (*pos*) começa no segundo bit do registo, apenas os bits 1 a 31 são processados, uma vez que o primeiro pertence às margens da imagem. O endereço do registo e da memória ROM começam na segunda linha (posição 12), pelo mesmo motivo.

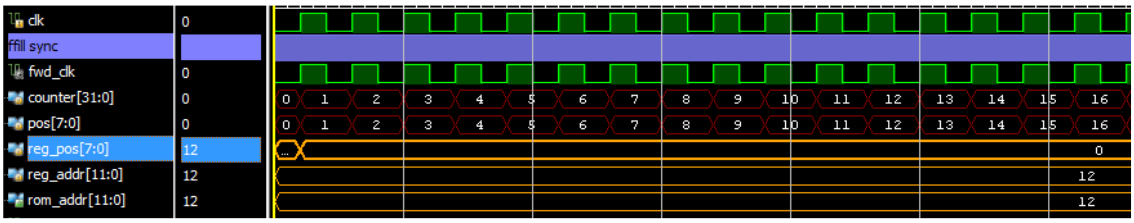


Figura 5.3.1.1 – Inicio do sistema de contagem

A primeira transição de registo pode ser observada na Figura 5.3.1.2. Como o registo 13 não pertence às margens da imagem, a variável de posição assume os valores 0 até 31. O registo *reg_pos* indica a posição na coluna da imagem, de 0 a 11, e o *fwd_clk*, indica que a operação está a ser realizada em *Raster Order*.

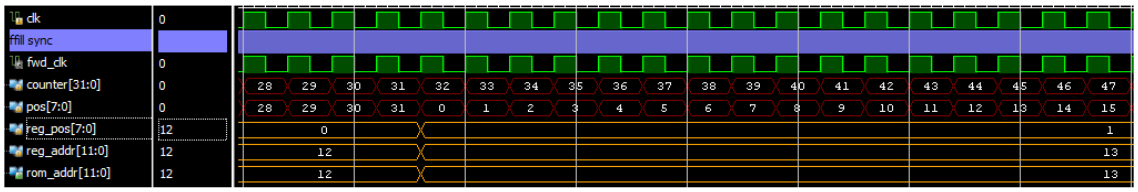


Figura 5.3.1.2 – Primeira transição de registo

A Figura 5.3.1.3 demonstra a transição de linha, por parte do sistema. O contador de posição salta do bit 30 do registo atual, para o bit 1 do registo seguinte, uma vez que, tanto o bit 31 do registo 11 como o bit 0 do registo 0 pertencem à margem da imagem. O endereço global do registo e da memória ROM é incrementado normalmente, sem nenhuma condição especial.

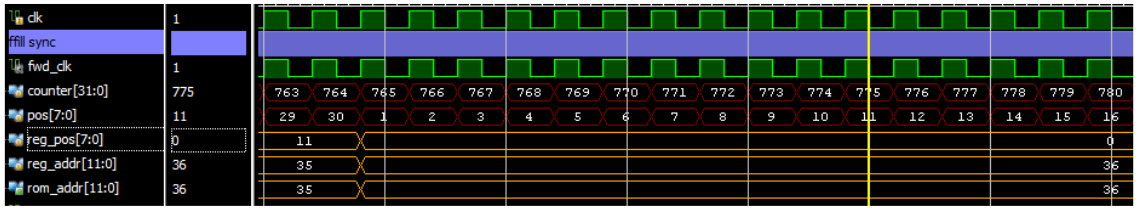


Figura 5.3.1.3 – Transição de linha

O processo inverso pode ser observado na Figura 5.3.1.4, onde a transição de linha é feita com a ordenação *Anti-Raster*. O sistema vai do registo 0 e endereço 3120, para o registo 11 do endereço de memória 3119. A variável de posição é decrementada até atingir o penúltimo bit do primeiro registo da linha atual, e salta para o segundo bit do último registo da linha anterior, evitando assim as margens da imagem.

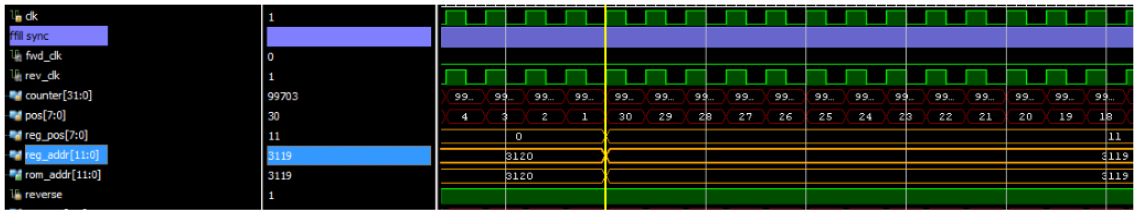


Figura 5.3.1.4 – Transição de linha Anti-Raster Order

5.3.2. Processamento de dados

O módulo de processamento de dados atua na transição descendente do relógio, tal como descrito anteriormente. A Figura 5.3.2.1 demonstra o preenchimento do marcador atual. O bit 31 do registo recebe o valor 1, uma vez que pertence às margens da imagem, assim, o bit 30 é o primeiro a ser processado pelo sistema. Na Figura 5.3.2.2 é possível visualizar o sistema, a preencher os diversos registos de marcador disponíveis, consoante a iteração do algoritmo.

Figura 5.3.2.1 – Preenchimento do valor do marcador atual

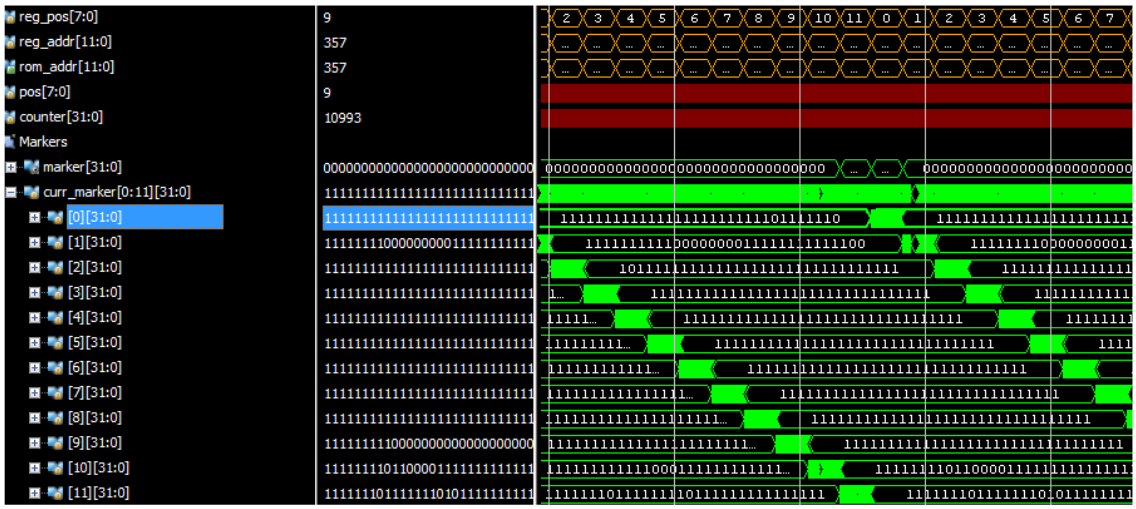


Figura 5.3.2.2 – Preenchimento dos diversos registos do marcador atual

A relação entre os valores do marcador atual (*curr_marker*), e os do marcador anterior (*last_marker*) pode ser analisada na Figura 5.3.2.3. Os valores do *last_marker* são utilizados no processamento dos valores do *curr_marker.* No entanto, quando o processamento do registo atual termina, o seu valor é copiado, para o registo de marcador anterior, para que possa ser utilizado no processamento da linha seguinte.

Figura 5.3.2.3 – Relação entre last_marker e curr_marker

O *Marker Canvas*, representado na Figura 5.3.2.4, é utilizado durante a primeira iteração do algoritmo em *Raster Order*. O registo retorna os valores de uma imagem, com todos os bits a zero e as margens brancas, a um, consoante o valor da variável de posição (*pos*). O valor do registo corresponde a 31 bits com o valor 1 na primeira e última linha da imagem. As restantes linhas apresentam o comportamento demonstrado na Figura 5.3.2.4. Assim, apenas o primeiro bit do primeiro registo tem o valor 1, os registos intermédios têm todos o valor 0 e o último registo da

linha apresenta o valor decimal 1. Este registo torna desnecessária a alocação do valor inicial da imagem marcador em memória, acelerando e otimizando o processo.

Figura 5.3.2.4 – Marker Canvas

5.3.3. Escrita na memória RAM

Após a inicialização da imagem (estados 1, 2 e 3), o módulo que controla o acesso e escrita na memória RAM fica em modo de espera (estado 0), até receber um pedido de escrita. Na Figura 5.3.3.1 é possível analisar um pedido de escrita quando o estado transita do 0 para o 4, o valor a escrever (*w_ram_val*) é carregado do registo, o endereço de escrita é incrementado e a *flag* (*w_ram*) que possibilita a escrita em memória é ativada. O estado 5 desabilita a *flag w_ram* e guarda o valor escrito num registo temporário (*ram_temp*). Uma vez que o sistema estava a executar em *Raster Order,* o módulo de controlo de memória voltou ao estado zero, onde permanece até receber um novo pedido de escrita.

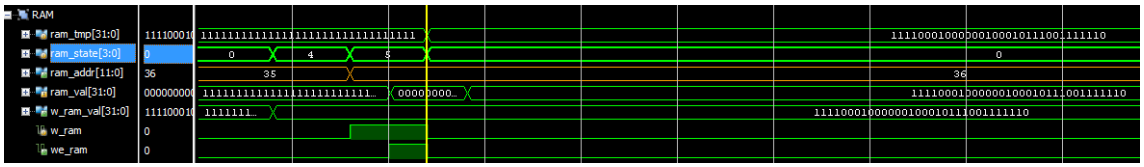


Figura 5.3.3.1 – Escrita na memória RAM

5.3.4. Execução do sistema

O sistema proposto apresenta dois modos de funcionamento distintos, devido à ordenação pela qual a imagem é percorrida. Assim, na primeira iteração a imagem é percorrida, da segunda à penúltima linha em *Raster order*. Com uma frequência de operação do sistema de 250MHz, uma imagem de 384x260 pixéis, pode ser percorrida em *Raster Order* em cerca de 405 µ segundos, como é possível verificar na Figura 5.3.4.1.

Figura 5.3.4.1 – Primeira iteração (Raster)

Após a primeira iteração, o sistema deve percorrer a imagem novamente, a partir da penúltima linha, até à segunda da imagem. Nesta segunda iteração, os valores do registo marcador são carregados a partir da memória RAM, contrariamente à primeira iteração, que utiliza os valores fornecidos pelo registo *marker_canvas,* trabalhando assim, com os resultados obtidos na primeira iteração. O sistema apresenta, em simulação, um tempo de processamento de cerca de 796µ segundos e 199022 ciclos de relógio para concluir o processo de reconstrução morfológica.

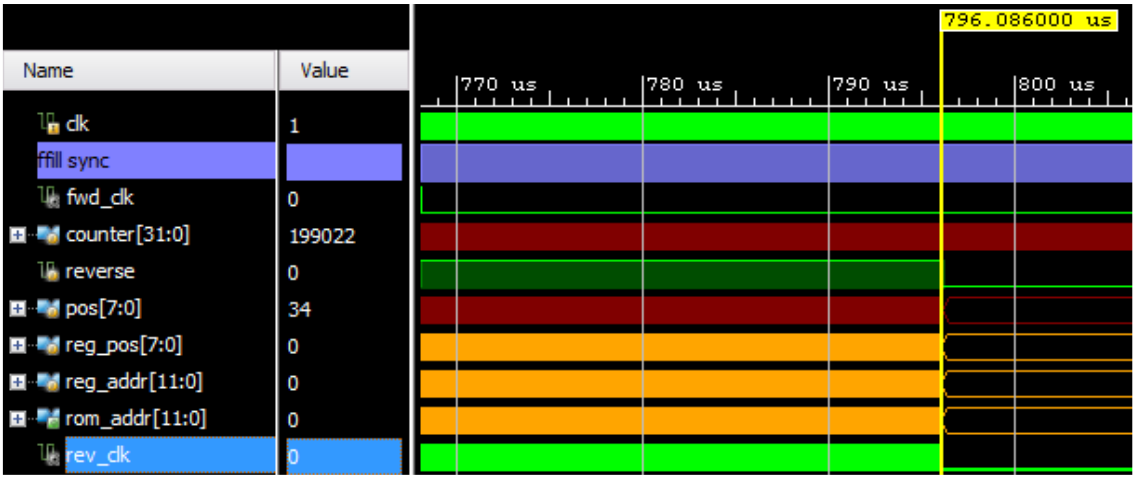


Figura 5.3.4.2 – Tempo de execução do módulo de reconstrução morfológica

5.4. Resultados Práticos

Os resultados obtidos pelo módulo de reconstrução morfológica foram confirmados, através do envio dos dados refentes à imagem pela porta série. Para tal, o sistema (*ffill_bus*) foi ligado ao *ZYNQ7 Processing System,* através de um barramento AXI-4 (*fill_bus*) no modo simples. Todos os resultados foram obtidos utilizando a placa de desenvolvimento Zybo.

A seguir à conclusão do processo de reconstrução morfológica, o controlo sobre o endereço e acesso à memória RAM (*dist_mem_gen_1*), é passado ao barramento de dados, para que este possa comunicar os resultados obtidos com o CPU.

Após a construção e programação dos blocos constituintes do sistema, a arquitetura é validada pelo processo de *synthesis*. O *bitstream* gera o ficheiro de programação da placa, para que a implementação possa ser testada. Uma vez que o CPU é necessário para confirmar os resultados obtidos, na metodologia de teste utilizada, o *hardware* é exportado para o *Vivado SDK*, onde é possível programar o CPU, para interagir com a arquitetura do sistema proposto.

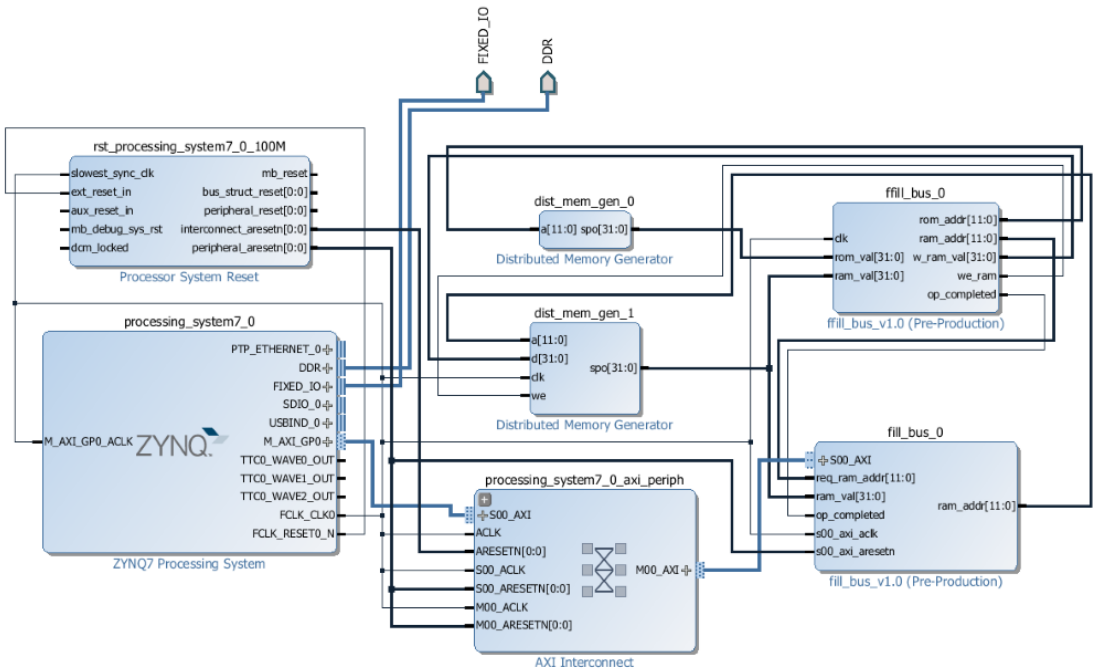


Figura 5.4.1 – Sistema de teste

Após carregar os drivers referentes ao sistema de teste, e enviar a configuração do *hardware* da FPGA para a placa de desenvolvimento, é necessário criar uma rotina de teste, representada na Figura 5.4.2, a qual permite validar os resultados. No registo *REG 0* é escrito o endereço de memória que se pretende ler, através da função *mWriteReg*. O resultado é lido no *REG 1* como um valor inteiro, sem sinal, de 32 bits, pela função *mReadReg*. O valor é convertido de novo em binário e enviado pela porta série.

```
#include <stdio.h>
#include "platform.h"
#include "fill_bus.h"
#include "xparameters.h"
#include "xil_io.h"


int main()
{
    init_platform();

      int count = 0;
      uint32_t result = 0;
      int bin = 0;
      int nmbr;
      do
      {

            FILL_BUS_mWriteReg(XPAR_FILL_BUS_0_S00_AXI_BASEADDR,
            FILL_BUS_S00_AXI_SLV_REG0_OFFSET,count);

            result = FILL_BUS_mReadReg(XPAR_FILL_BUS_0_S00_AXI_BASEADDR,
            FILL_BUS_S00_AXI_SLV_REG1_OFFSET);

            xil_printf("POS:%d,", count);
            for (bin = 31; bin >=0; bin--)
            {
                  nmbr = result  >> bin;
                  nmbr &=1;
                  xil_printf("%d",nmbr);
            }
            xil_printf("\n\r");
            count++;

      }while(count < 3131);

    cleanup_platform();
    return 0;
}
```

Figura 5.4.2 – Rotina de teste do CPU

O modo de funcionamento simples do barramento *AXI-4* é muito útil, na depuração de resultados, no entanto a latência que apresenta, entre pedidos, não permite que seja utilizado em aplicações. Com uma frequência de operação de 250 MHz na FPGA, dois pedidos de leitura consecutivos, apresentam uma latência de 25 ciclos de relógio. O acesso por DMA à memória

externa, ou a utilização do barramento em modo *stream*, apresentam-se como soluções viáveis a este problema.

A aplicação em CUDA foi modificada para fornecer o valor dos pixéis da imagem, em registos de 32 bits, após o processo de reconstrução morfológica, criando assim uma base de comparação e validação de resultados entre ambas as abordagens. Na Figura 5.4.3 e Figura 5.4.4 é possível observar, à esquerda, os resultados recebidos, pela porta série, do sistema em FPGA e comparar os valores com os resultados obtidos pela aplicação em CUDA (imagem da direita), comprovando assim os resultados da implementação.

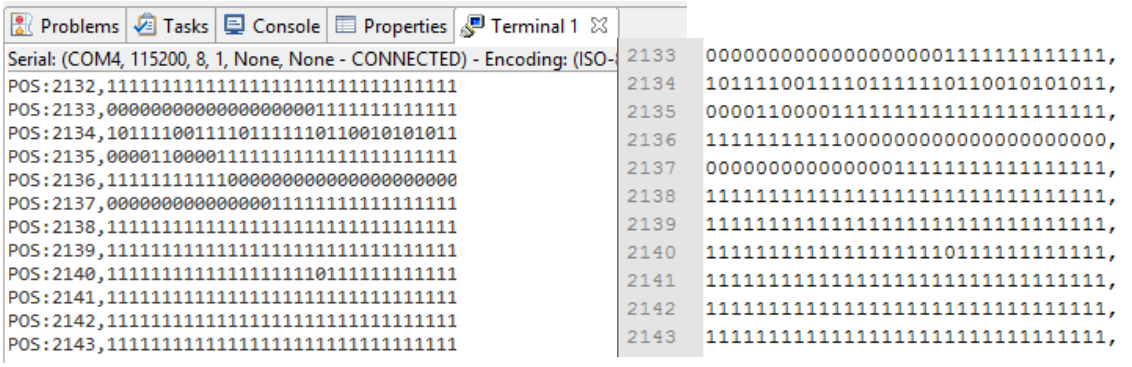


Figura 5.4.3 – Comparação de resultados entra FPGA e CUDA

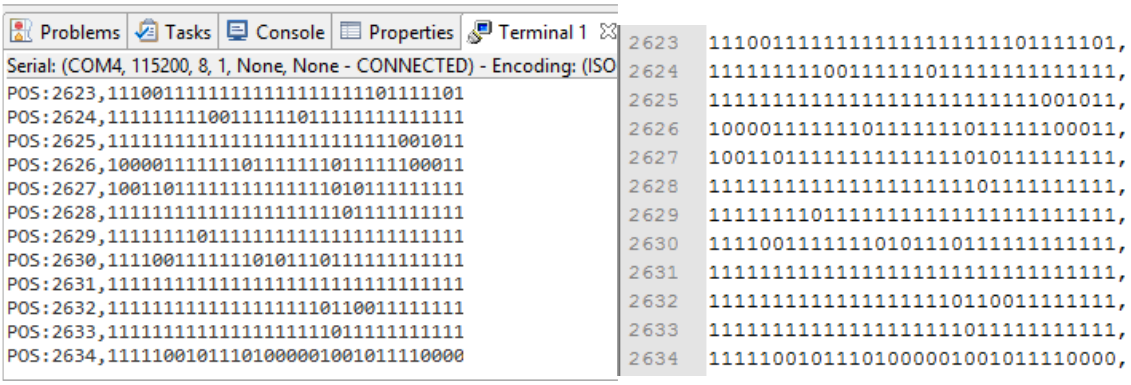


Figura 5.4.4 – Comparação de resultados entra FPGA e CUDA 2

5.4.1. Escalabilidade do sistema com o aumento da resolução de imagem

Após a implementação do algoritmo, é possível elaborar algumas conclusões face à sua escalabilidade. Os resultados práticos demonstraram que o sistema necessita de 199022 ciclos de relógio, para efetuar a reconstrução morfológica, de uma imagem com as dimensões de

384x260. A partir deste resultado, é possível estimar o número de ciclos necessários para imagens de outras dimensões e o tempo de processamento, consoante a frequência de operação do sistema. O algoritmo é capaz de processar um valor um pouco superior a um pixel por ciclo de relógio, uma vez que as margens da imagem não são processadas. A Tabela 5.4.1.1 demonstra o estudo realizado sobre a escalabilidade do sistema, face ao aumento da resolução de imagem e diferentes frequências de operação.

Largura	Altura	Número de Pixéis	Ciclos de relógio (estimativa)	Frequência 100Mhz	Frequência 250MHz	Frequência 500 MHz
384	**260**	99840	199022	1,99 ms	0,796 ms	0,398 ms
640	**480**	307200	612375	6,12 ms	2,45 ms	1,225 ms
1024	**640**	655360	1306401	13,06 ms	5,226 ms	2,613 ms
1024	**820**	839680	1673826	16,74 ms	6,695 ms	3,348 ms
1280	**720**	921600	1837126	18,37 ms	7,349 ms	3,674 ms
1920	**1080**	2073600	4133534	41,34 ms	16,534 ms	8,267 ms

Tabela 5.4.1.1 – Escalabilidade do algoritmo

5.4.2. Recursos utilizados no FPGA

De uma forma geral, o módulo de reconstrução morfológica não ocupa muito espaço, mesmo numa FPGA com menor capacidade como a Zybo, pelo que a implementação do módulo de pré-processamento e de *Threshold* local são totalmente viáveis, numa placa de desenvolvimento com um pouco mais de capacidade. Pela análise da Figura 5.4.2.1, que representa a ocupação, em conjunto com o diagrama do sistema proposto (Figura 5.2.2.1), é possível verificar que grande parte do espaço utilizado advém da utilização de blocos de RAM internos. Na Figura 5.4.2.2 é possível analisar, mais detalhadamente, a utilização do sistema.

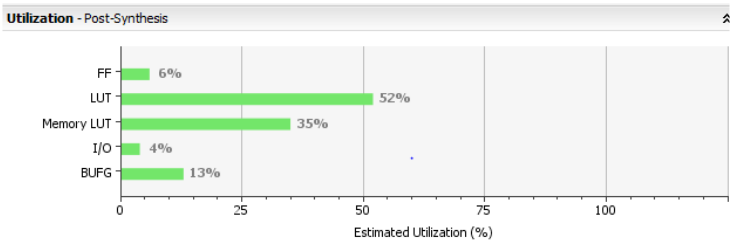


Figura 5.4.2.1 – Utilização de recursos do módulo de reconstrução morfológica

Utilization - Post-Synthesis

Resource	Estimation	Available	Utilization %
FF	2242	35200	6.37
LUT	9210	17600	52.33
Memory LUT	2118	6000	35.30
I/O	4	100	4.00
BUFG	4	32	12.50

Figura 5.4.2.2 – Utilização de recursos (detalhada)

De uma forma geral, a placa de desenvolvimento apresenta uma arquitetura de memória externa unificada, onde é necessário a utilização de canais de DMA, para acesso à RAM externa, o que condicionou o processo de desenvolvimento. O sistema proposto apresenta um acesso repetido à memória, pelos diversos módulos de processamento, pelo que um sistema que permita maior velocidade de acesso, do FPGA à memória é essencial para o sucesso da metodologia.

5.5. Discussão

A implementação do módulo de reconstrução morfológica em FPGA pode ser considerada bem-sucedida, tendo sido alcançados resultados teóricos comparáveis à implementação em CPU, para uma resolução de 1024x820, com uma frequência de operação de 250 Mhz, como é possível verificar pela comparação da Tabela 5.4.1.1 com a Figura 4.8.4.1. No entanto, é preciso entrar em conta com a limitação de recursos disponíveis num FPGA, pelo que a resolução da imagem está mais limitada.

A implementação do sistema proposto em FPGA é considerada viável, especialmente porque apresenta uma solução embebida, com resultados próximos da implementação em desktop, para o módulo de reconstrução morfológica, que é considerado o mais crítico do sistema.

Uma nova abordagem, que melhora o tempo de execução do módulo de reconstrução morfológica, mas limita a resolução das imagens analisadas, devido a uma maior utilização dos recursos disponíveis no FPGA, é proposta como trabalho futuro, no próximo capítulo.

6. Conclusão e Trabalho Futuro

Neste capítulo, são apresentadas as principais conclusões relativas ao trabalho desenvolvido nesta dissertação de mestrado. De seguida, são propostas possíveis linhas de investigação a serem realizadas como trabalho futuro.

6.1. Conclusão

O principal problema das aplicações de seguimento é relacionar o mesmo objeto, em dois fotogramas diferentes. No seguimento de veículos, a matrícula pode ser utilizada como identificador único, que relacionará os veículos presentes em diferentes fotogramas. No entanto, as técnicas de segmentação da região da matrícula propostas na literatura, não apresentam as condições ideais para a implementação de um sistema de seguimento de veículos robusto. O objetivo dos sistemas ANPR tem passado apenas pela deteção da matrícula em pelo menos um, dos diversos fotogramas analisados, onde a região da matrícula ocupa uma proporção considerável da imagem e o *background* não sobressai. O seguimento é realizado recorrendo a janelas de pesquisa que limitam muito a área a analisar, e o número máximo de veículos detetados corresponde ao número de janelas aplicadas sobre a imagem. Neste trabalho de dissertação, foi proposta uma nova abordagem na segmentação da região da matrícula, que permitirá obter resultados superiores no seguimento de veículos. Numa primeira fase, a técnica de *Threshold* local é utilizada para: diminuir o nível de detalhe, tornar o sistema mais imune às diferenças de luminosidade, e especialmente realçar o contraste entre os caracteres e a chapa da matrícula. Numa segunda fase, é aplicada a técnica de *Fill Hole* sobre a imagem binária, preenchendo todas as regiões ligadas às margens, com a exceção dos buracos da imagem que correspondem aos caracteres da matrícula. Após o processo de segmentação são analisados os histogramas verticais e horizontais da imagem resultante, onde apenas os caracteres das diversas matrículas apresentam dimensões e proporções, para serem considerados como região da matrícula. Assim, esta nova abordagem procura encontrar todos os veículos presentes em praticamente qualquer posição e distância na imagem, desde que a sua matrícula se encontre visível, permitindo a implementação de um sistema de seguimento de veículos mais robusto.

A implementação do módulo de segmentação em CUDA revelou-se essencial para o sucesso do projeto. Recorrendo à API OpenGL foi possível observar os resultados dos diversos módulos do sistema, e o impacto que as diversas variáveis de calibração apresentavam perante o resultado do processo de segmentação. A imprevisibilidade no número de iterações do módulo de reconstrução por dilatações geodésicas em GPU, inviabilizou a sua implementação. Assim, o processo de segmentação proposto, recorre à aceleração por GPU do módulo de *Threshold* local, devido ao paralelismo inerente apresentado pelo algoritmo, e executa o processo de reconstrução morfológica no CPU, devido à sequencialidade demonstrada pela técnica utilizada. Recorrendo às ferramentas de *profiling, Nsight* e *gprof,* foi possível identificar o módulo de reconstrução morfológica sequencial, como o mais crítico do sistema. O sucesso da implementação da metodologia proposta, num sistema heterogéneo composto por FPGA e CPU depende dos resultados obtidos, no processo de reconstrução morfológica. O processo de segmentação proposto, demora cerca de 20 milissegundos a executar, em CUDA, para uma imagem com a resolução de 1024x820, correspondendo a cerca de 50 fotogramas por segundo, respondendo assim aos requisitos temporais impostos ao sistema. Não sendo possível utilizar uma imagem com a mesma resolução, na implementação em FPGA do módulo de reconstrução morfológica, devido às restrições impostas pela plataforma de desenvolvimento utilizada, a temporização do módulo foi realizada para uma imagem com 384x260 de resolução. Uma estimativa sobre a escalabilidade do módulo foi apresentada, para diferentes resoluções e frequências de operação, através da comparação do número de pixéis da imagem utilizada, com o número de ciclos de relógio necessários, para concluir uma iteração do módulo de reconstrução morfológica. A estimativa obtida aponta para resultados muito próximos da implementação em CPU, no processo de reconstrução morfológica, que é o mais crítico do sistema. Assim, é possível comprovar a viabilidade da implementação do sistema proposto em FPGA. Pode-se então concluir, que a implementação da metodologia proposta num sistema embebido seria bem-sucedida.

Relativamente ao processo de implementação, pode concluir-se que em aplicações de processamento de imagem, a implementação da metodologia, num sistema heterogéneo composto por CPU e GPU, recorrendo a uma *framework* como o CUDA, numa fase, inicial será o caminho ideal. A semelhança que apresenta com o ANSI C, a rapidez apresentada pelo processo de depuração dos resultados obtidos, recorrendo a APIs gráficas como o DirectX e o OpenGL, e a

qualidade apresentada pelas ferramentas de *profiling* disponíveis, permitem o desenvolvimento de uma metodologia, sem que as limitações de *hardware* sejam evidentes e alterem o processo. O OpenCL pode ser utilizado como alternativa ao CUDA, sendo suportado por um maior número de plataformas. No entanto, o processo é mais complexo e requer um melhor conhecimento das APIs. A implementação de sistemas de processamento de imagem em FPGA, é um processo complexo e moroso, onde as limitações de *hardware* são mais evidentes, especialmente em imagens de elevada resolução. As metodologias e o funcionamento do sistema devem estar bem definidos, antes de se poder considerar a implementação do sistema viável. O FPGA continua a ser uma das melhores plataformas de aceleração em sistemas embebidos, mesmo em aplicações de processamento de imagem. A conversão do sistema projetado em FPGA para ASIC, é outro fator que viabiliza a utilização da tecnologia. É possível então concluir que, a implementação em CUDA apresenta maior flexibilidade na prototipagem e depuração do sistema, diminuindo o *time-to-market* e o custo de desenvolvimento. A implementação em FPGA será especialmente importante, em sistemas embebidos e em aplicações de tempo real. No entanto, além da complexidade da implementação, apresenta um custo de desenvolvimento e *time-to-market* superior.

6.2. Trabalho Futuro

O trabalho realizado nesta dissertação teve como principal foco, a apresentação de uma nova abordagem na segmentação da região da matrícula e otimização do processo. No entanto, os módulos de pré e pós-processamento apenas foram trabalhados em matlab. Assim, a próxima etapa lógica do trabalho passa pela implementação dos referidos módulos em CUDA, e pelo estudo sobre o impacto que apresentarão no tempo de execução do sistema. Relativamente à implementação em FPGA, o próximo passo lógico será a utilização de um sistema externo de memória, e a análise do impacto que o aumento na resolução das imagens analisadas exerce sobre os recursos disponíveis. Posteriormente, a implementação dos módulos de *Threshold local*, pré e pós-processamento será necessária, de forma a concluir o sistema proposto.

Após a implementação do sistema, será possível otimizar o processo através da adição de um *pipeline*. Do ponto de vista da aplicação CUDA, o *pipeline* tem como objetivo diminuir o tempo de espera que cada módulo apresenta, garantindo que o GPU não fica à espera da conclusão do

processo de reconstrução morfológica, para iniciar o pré-processamento do fotograma seguinte. A adição de um *pipeline* ao módulo de reconstrução morfológico em FPGA, permitirá diminuir o tempo de execução do módulo em mais de 50%. O algoritmo utilizado apresenta uma dependência, do valor de três pixéis da linha anterior e do valor do pixel anterior, em relação à posição atual. Numa arquitetura síncrona, é possível iniciar o processamento da linha seguinte, após calcular o valor de pelo menos três pixéis da linha atual. Assim, o número de linhas a processar em simultâneo depende: dos recursos de *hardware*, da largura da imagem e do acesso à memória. Esta é uma nova abordagem proposta, que caso se verifiquem os resultados esperados, torna a implementação do algoritmo de reconstrução morfológica de imagens binárias em FPGA a melhor opção, sendo ainda necessário trabalhar o conceito antes de se conseguir provar a sua viabilidade.

Finalmente, a última etapa será a implementação de um módulo de seguimento e reconhecimento de caracteres (OCR), temas que não foram abordados muito profundamente, durante a realização do trabalho de dissertação. No entanto, o sistema foi desenvolvido tendo em conta os módulos e as suas limitações. O principal objetivo da metodologia apresentada é proporcionar um sistema de seguimento mais robusto, através da deteção e reconhecimento de vários veículos, presentes numa imagem com resolução elevada, no maior número de fotogramas consecutivos possível.

Bibliografia

[1] J. Kang, M. Kang, C. Park, J. Kim e Y. Choi, "Implementation of embedded system for vehicle tracking and license plates recognition using spatial relative distance," *Information Technology Interfaces, 2004. 26th International Conference on,* vol. 1, pp. 167-172, 2004.

[2] A. Yilmaz, O. Javed e M. Shah, "Object Tracking: A Survey".

[3] X. Zhai, F. Bensaali e K. McDonald-Maier, "Automatic Number Plate Recognition on FPGA," *IEEE.*

[4] I. University, "5.0 Best Practices for Optical Character Recognition," [Online]. Available: http://www.library.illinois.edu/dcc/bestpractices/chapter_05_ocr.html.

[5] S. Elkosantini e S. Darmoul, "Intelligent Public Transportation Systems: A Review of Architectures and Enabling Technologies," *IEEE,* 2013.

[6] C. Strong e S. Albert, "Advanced Transportation Systems ITS Strategic Deployment Plan," *Western Transportation Institute ,* 2002.

[7] F. Baratian-Ghorghi, H. Zhou e J. Shaw, "Wrong-Way Driving Fatal Crashes in the United States," *ITE Journal,* vol. 84, nº Highways; Safety and Human Factors;, pp. 41-47, 2014.

[8] C.-N. E. Anagnostopoulos, I. E. Anagnostopoulos e e. al, "License Plate Recognition From Still Images and Video Sequences: A Survey," *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS,* vol. 9, pp. 377-391, 2008.

[9] W. K. Pratt, "Image Segmentation," em *Digital Image Processing*, 2001, pp. 551-591.

[10] R. Gonzalez e R. Woods, "Image Segmentation," em *Digital Image Processing*, pp. 567-634.

[11] Horowitz e Pavlidis, "Picture segmentation by a directed split-and-merge procedure.," *Proceedings of the Second International Joint Conference on Pattern Recognition,* pp. 424-433, 1974.

[12] E. Davies, "Region-Growing Methods," em *Computer & Machine Vision 4th edition*, pp. 83-84.

[13] E. Davies, "Thresholding Techniques," em *Computer & Machine Vision 4th edition*, pp. 82-110.

[14] J. Weska, "A survey of threshold selection techniques," *Image Process,* vol. 7, pp. 259-265.

[15] M. Sezgin e B. I. Sankur, "Survey over image thresholding techniques and quantitative performance evaluation," *Journal of Electronic Imaging,* vol. 13(1), pp. 146-165, 2004.

[16] N. OTSU, "A TIreshold Selection Method from Gray-Level Histograms," *IEEE TRANSACTIONS ON SYSTREMS, MAN, AND CYBERNETICS,* 1979.

[17] C.-N. E. Anagnostopoulos e V. L. I. D. Psoroulas, "License Plate Recognition From Still Images and Video Sequences: A Survey," *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS,* vol. 9, 2008.

[18] E. R. Davies, "Edge Detection," em *Computer and Machine Vision*, pp. 111-148.

[19] L. Yichuan e Z. Chunhong, "Vehicle License Plate Location Based on Mathematical Morphology and Variance Projection," *IEEE.*

[20] A. M. Al-Ghaili, S. Mashohor, A. R. Ramli e A. Ismail, "Vertical-Edge-Based Car-License-Plate," *IEEE.*

[21] S.-J. Yang, J.-B. Jiang, M.-K. Wu e C. C. Ho, "Real-Time License Plate Detection System with 2-level 2D Haar Wavelet Transform and Wiener-Deconvolution Vertical Edge Enhancement," *IEEE.*

[22] M. M. Roomi, M. Anitha e R. Bhargavi, "Accurate License Plate Localization," *IEEE.*

[23] R. C. Gonzalez e R. E. Woods, "Histogram processing," em *Digital Image Processing*, Prentice Hall, 2007, pp. 88-108.

[24] H. s. F. i. o. H. t. f. r.-t. applications, "Liberis Voudouris; Spiridon Nikolaidis; Abdoul Rjoub," *IEEE,* 2012.

[25] R. O. Duda e P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," *Communications of the ACM,* vol. Volume 15 Issue 1, pp. 11-15 , 1972.

[26] L. Voudouris, S. Nikolaidis e A. Rjoub, "High speed FPGA implementation of Hough transform for real-time applications," *IEEE,* 2012.

[27] E. R. Davies, "The Hough Transform and Its Nature," em *Computer and Machine Vision, 4th edition*, Elsevier, 2012, pp. 333-357.

[28] E. Davies, "Foreground - Background separation," em *Computer and Machine Vision, 4th edition*, pp. 584-591.

[29] E. Davies, "Optical Flow," em *Computer and Machine Vision, 4th edition*, pp. 505-512.

[30] MathWorks, "Tracking Cars Using Optical Flow," [Online]. Available: http://www.mathworks.com/help/vision/examples/tracking-cars-using-optical-flow.html.

[31] E. R. Davies, "Mathematical Morphology," em *Computer and Machine Vision*, pp. 185-197.

[32] P. Soille, "Erosion and Dilation," em *Morphological Image Analysis*, 2004, pp. 62-82.

[33] X. Zhai, F. Bensaali e S. Ramalingam, "Real-Time License Plate Localisation on FPGA," *IEEE.*

[34] E. Arianyan, S. A. Motamedi e I. Arianyan, "Efficient Optical Character Recognition on Graphics Processing Unit," *IEEE,* nº 6'th International Symposium on Telecommunications, pp. 789-793, 2012.

[35] X. Zhai e F. Bensaali, "Standard Definition ANPR System on FPGA and an," *IEEE,* 2013.

[36] S.-Y. Yang, Y.-C. Lu, L.-Y. Chen e D.-C. Cherng, "Hardware-accelerated Vehicle License Plate Detection at High-definition Image," *IEEE,* 2011.

[37] E. Davies, "Adaptive Thresholding," em *Computer and Machine Vision, 4th edition*, pp. 88-94.

[38] L. Vandevenne, "Flood Fill," [Online]. Available: http://lodev.org/cgtutor/floodfill.html. [Acedido em 16 12 2015].

[39] P. Gribble, "Memory : Stack vs Heap," 2012. [Online]. Available: http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html. [Acedido em 2 9 2015].

[40] P. Soille, "Geodesic Transformations," em *Morphological Image Analysis*, Springer-Verlag, 2004, pp. 183-218.

[41] L. Vincent, "Morphological Grayscale Reconstruction in Image Analysis: Applications and Efficient Algorithms," *IEEE Transactions on Image Processing,* vol. 2, pp. 176-201, 1993.

[42] R. C. Gonzalez, R. E. Woods e S. L. Eddins, "Morphological Reconstruction," [Online]. Available: http://www.mathworks.com/tagteam/64199_91822v00_eddins_final.pdf.

[43] I. JIVET, A. BRINDUSESCU e I. BOGDANOV, "Image Contrast Enhancement using Morphological Decomposition by Reconstruction," *WSEAS TRANSACTIONS on CIRCUITS and SYSTEMS,* vol. 7, 2008.

[44] P. Karas, "Efficient Computation of Morphological Greyscale Reconstruction".

[45] Y. Sharma e Y. K. Meghrajani, "Brain Tumor Extraction From MRI Image Using Mathematical Morphological Reconstruction," *IEEE,* 2014.

[46] L. WANG e H. LIU, "An efficient method for identifying and filling surface depressions in digital elevation models for hydrologic analysis and modelling," *International Journal of Geographical Information Science,* vol. 2, pp. 193-213, 2006.

[47] R. Barnes, C. Lehman e D. Mulla, 2013.

[48] M. Iwanowski e P. Soille, "Fast algorithm for order independent binary homotopic thinning".

[49] Nvidia, "HRAA: High-Resolution Antialiasing through Multisampling," [Online]. Available: http://www.nvidia.com/object/feature_hraa.html. [Acedido em 25 9 2015].

[50] "Nearest Neighbor Image Scaling," [Online]. Available: http://tech-algorithm.com/articles/nearest-neighbor-image-scaling/. [Acedido em 25 9 2015].

[51] D. Han, "Comparison of Commonly Used Image Interpolation Methods," *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering,* 2013.

[52] S. Asano, T. Maruyama e Y. Yamaguchi, "PERFORMANCE COMPARISON OF FPGA, GPU AND CPU IN IMAGE PROCESSING," *IEEE,* 2009.

[53] J. Fung e S. Mann, "COMPUTER VISION SIGNAL PROCESSING ON GRAPHICS PROCESSING UNITS," *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference,* vol. 5, pp. 93-96, 2004.

[54] Nvidia, " GeForce GTX 970 Tech specs," [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-970/specifications.

[55] J. Cheng, M. Grossman e T. McKercher, Professional CUDA C Programming, Indianapolis, Indiana: John Wiley & Sons, Inc., 2014.

[56] H. Kim, R. Vuduc, S. Baghsorkhi e J. Choi, "Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)," *Synthesis Lectures on Computer Architecture,* vol. 1, p. 96, 2012.

[57] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn e T. J. Purcell, "A Survey of General-Purpose Computation on Graphics," *The Eurographics Association,* 2005.

[58] B. Neelima e P. S. Raghavendra, "Recent Trends in Software and Hardware for GPGPU Computing: A Comprehensive Survey," *IEEE,* 2010.

[59] M. Shevtsov, "OpenCL*: the advantages of heterogeneous approach," *Intel,* 2013.

[60] I. Kuon, R. Tessier e J. Rose, "FPGA Architecture: Survey and Challenges," *Foundations and Trends in Electronic Design Automation,* Vols. %1 de %2Vol. 2, No. 2 (2007) 135–253, 2007.

[61] M. Awad, "FPGA SUPERCOMPUTING PLATFORMS: A SURVEY," *IEEE,* 2009.

[62] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons e F. Morgan, "Xilinx Vivado High Level Synthesis: Case studies," *Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014). 25th IET,* pp. 352 - 356, 2013.

[63] Altera, User-Customizable ARM-Based SoC, 2014.