UNIVERSIDADE ESTADUAL DE LONDRINA
Departamento de Engenharia Elétrica

GUILHERME BRANDÃO DA SILVA

Método heterogêneo de detecção de faixas de trânsito em tempo real para sistemas embarcados

Heterogeneous method for real-time lane detection in embedded systems.

Londrina
2021.

Guilherme Brandão da Silva

Método heterogêneo de detecção de faixas de trânsito em tempo real para sistemas embarcados

Heterogeneous method for real-time lane detection in embedded systems.

Dissertação de Mestrado apresentada ao Departamento de Engenharia Elétrica da Universidade Estadual de Londrina como parte dos requisitos para a obtenção do título de Mestre, na área de Engenharia Elétrica.

Master Thesis presented to the Departament of Electrical Engineering of the University of Londrina in partial fulfillment of the requirements for the degree of Master of Eletrical Enginnering, in the area of Electrical Engineering.

Orientador: Prof. Dr. Leonimer de Melo

Este exemplar corresponde à versão da Dissertação de Mestrado entregue à banca antes da defesa.

Londrina
2021.

Resumo

A detecção e análise de faixas sobre as vias de trânsito destaca-se como uma das tecnologias fundamentais na implementação de veículos inteligentes e são essenciais em sistemas que auxiliam motoristas e veículos autônomos a dirigirem com segurança. A queda no custo associado ao aumento no desempenho das comodities da eletrônica, tais como sensores de visão e chips gráficos integrados, impulsionaram pesquisas e aplicações de direção autônoma e semi-autônoma, onde a detecção de faixas de trânsito é uma das tecnologias fundamentais. Este trabalho propõe a implementação heterogênea otimizada de um sistema de detecção de faixas em tempo real capaz de analisar e estimar as características geométricas e espaciais das faixas centrais observadas. O sistema proposto, baseado em imagens de uma câmera monocular, funciona em uma sequência temporal de imagens. Os quadros são pré-processados para a obtenção de uma imagem integrada em perspectiva aérea, onde características das sinalizações de faixa são extraídas para a formação de mapas de evidências. Estes mapas combinados geram uma imagem binária robusta que contêm as marcações das faixas. A estimativa final da faixa é modelada por um polinômio e é detectada via método de janelas deslizantes. Para avaliação e validação do sistema proposto, o algoritmo foi implementado em um sistema embarcado convencional, NVIDIA Jetson Nano, com um SoC Tegra X1 dotado de um processador ARM A57 e um chip gráfico NVIDIA com 128 núcleos CUDA. São utilizadas e imagens reais de uma base de dados convencional para o problema de detecção de faixas de trânsito. Foram analisadas métricas de acurácia, taxa de detecção e tempo de execução para diferentes cenários de vias. Ademais, o método se mostrou eficiente para detectar as faixas de trânsito em diversos cenários como estradas e rodovias, em diferentes condições, horários e trânsito. O desempenho do método variou de 92,3% a 99,0% de acurácia no melhor subconjunto, detectando até 97,9% das faixas centrais disponíveis com 80% dos pontos detectados válidos. A implementação paralela otimizada foi capaz de executar, respectivamente, 140 vezes e 25 vezes mais rápido que a sua versão serial e híbrida, performando a uma taxa de 300 quadros por segundo, ultrapassando em dez vezes os requisitos de execução em tempo real.

Abstract

Decreasing costs and improvements on the performance of vision sensors, graphic chips, and embedded hardware have boosted research and applications related to autonomous and semi-autonomous driving. Within this area, lane detection is one of the most critical technologies. It is a fundamental part of the implementation of autonomous driving or advanced driver assistance systems. This work proposes the heterogeneous implementation of a real-time lane detection system that should be capable of analyzing and estimating geometrical and spatial characteristics of the observed lanes. The proposed system is based on images from a monocular camera and works on a temporal sequence of images. Each image frame is pre-processed to obtain a grayscale bird's view perspective version of the image. Then the features of the lane marks are extracted from that perspective by building some features maps. These maps are combined to generate a robust binary image with the lane marks information. The final estimation of the lanes is modeled by a polynomial fit, which is detected through an iterative sliding window method. For evaluation and testing of the proposed method, the algorithm was implemented in a conventional embedded system with the SoC Tegra X1 equipped with an ARM A57 processor and an NVIDIA graphics chip with 128 CUDA cores. Real images of a common dataset are used as input of the system. The method proved efficient for detecting ego-lanes on roads and highways running TuSimple dataset images with different scenarios. The method's performance ranged from 92.3% to 99.0% accuracy in the best subset, detecting up to 97.9% of the ego-lanes available with 80% of the valid points. In this way, the work is in the same range as other similar methods with the difference of being executed in an inexpensive graphics-processing tailored real-time embedded system. The method performs within the established real-time limits. The embedded system executed it in just 2.34 ms, which is 25 faster than the hybrid version and 140 times faster than the CPU-executed sequential version. Additionally, considering the method's run-time, future implementations can improve even further and enhance the detection quality still copping with the real-time criteria.

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

CUDA	*Compute Unified Device Architecture* (Arquitetura de Dispositivo de Computação Unificada)
GPU	*Graphics Processing Unit* (Unidade de Processamento Gráfico)
CPU	*Central Processing Unit* (Unidade de Processamento Central)
ROI	*Region of Interest* (Região de Interesse)
BEV	*Bird's Eyes View*
IPM	*Inverse Perspective Transform* (Tranformação de Perspectiva Inversa)
SIMT	*Single-Instruction, Multiple-Thread* (Instrução Única, Múltiplas Threads)
SIMD	*Single-Instruction, Multiple-Data* (Instrução Única, Múltiplos Dados)
AoS	*Array of Structs* (Arranjo de Estruturas)
SoA	*Struct of Arrays* (Estrutura de Arranjos)
API	*Application Programming Interface* (Interface de Programação de Aplicativo)
SM	*Shared Memory* (Memória Compartilhada)
GSL	*Grid-Stride Loop*
DLD	*Dark-Light-Dark* (Escuro-Claro-Escuro)
SLAM	*Simultaneous Location And Mapping* (Localização e Mapeamento Simultâneos)
SSH	*Secure Shell*
TFP	Taxa de Falsos Positivos
EMQ	Erro Médio Quadrático
DOG	*Difference of Gaussians* (Diferença entre Gaussianas)
UCF	*Unilateral Correlation Filter* (Filtro de Correlação Unilateral)
ACC	Acurácia
SRF	*Step-Row Filter*
RGB	*Red-Green-Blue*

Sumário

Capítulo 1

Introdução

1.1 Conceitos e Motivações

O aumento do fluxo de veículos nas últimas décadas foi acompanhado da alta no número de acidentes reportados e da consequente alta no número de vítimas. Segundo Transp. (2010), as falhas relacionadas ao tempo de resposta dos motoristas é responsável por 70% dos acidentes na Grã-Bretanha. De acordo com o estudo da NHTSA (2015), aproximadamente 94% das falhas críticas que precedem a cadeia de eventos de um acidente de trânsito podem ser atribuídas aos motoristas.

Buscando a redução dos acidentes, a indústria automobilística realiza constantes investimentos em sistemas de auxílio aos motoristas durante a condução, e até mesmo em sistemas que intervém de forma autônoma em condições identificadas como de grande risco ou críticas (BENGLER et al., 2014).

Neste sentido, diversos sistemas automotivos foram desenvolvidos nas últimas décadas. Atualmente, o desenvolvimento destas tecnologias aponta para a criação de sistemas que futuramente evoluam para a direção autônoma, criativa e cooperativa. Tais sistemas tem por fonte de informação uma rede integrada de sensores, tais como uma rede multiplataforma com câmeras, radares, sistemas de visão noturna, inerciais, entre outros.

Ainda segundo Bengler et al. (2014), para garantir que as tecnologias de direção autônoma e cooperativa funcionem adequadamente, diversos subsistemas devem atuar em conjunto. Um destes subsistemas essenciais é a detecção e identificação de faixas em vias e rodovias, que é tópico de estudo importante no campo dos sistemas de carros inteligentes.

Através da detecção das faixas, é possível fornecer a posição do veículo em relação às faixas, possibilitando a determinação de uma direção eficaz para o carro. Esse recurso pode aprimorar significativamente a eficiência e a segurança na direção (KUM et al., 2013).

A informação visual gerada por câmeras é um dos meios mais utilizados para a detecção de faixas e consequentemente para a direção inteligente. A principal vantagem desta é a quantidade de informação que pode ser obtida em função do custo, tanto operacional quanto monetário. Adicionalmente, outros sensores, tais como ultrassônicos, radares e lasers, podem ser utilizados em conjunto com as câmeras (BAR HILLEL et al., 2014a).

É essencial que os algoritmos para a detecção de faixas tenham a menor complexidade computacional possível, uma vez que estes devem atender a requisitos de tempo real para sua operação em um sistema embarcado no veículo. Além disso, sistemas embarcados possuem recursos limitados e reduzidos, tais como a capacidade de processamento e a quantidade de memória, frente a um sistema computacional completo de uso pessoal, por exemplo.

Grande parte dos trabalhos sobre detecção de faixa encontrados na literatura desconsideram a complexidade computacional das soluções propostas, pois não tem como objetivo a implementação destes em sistemas embarcados. Este fato limita ou impossibilita a reprodução destes resultados em aplicações reais embarcadas.

O processamento de imagem, em geral, é uma tarefa que demanda alto tempo de execução devida a sua complexidade elevada, limitando seu uso em aplicações que demandam requisitos de tempo real (MONDAL; BISWAL; BANERJEE, 2016). Além disto, a complexidade do processamento aumenta conforme o número de imagens e a resolução destas, dificultando ainda mais algumas aplicações em sistemas com requisitos de tempo real (MERTES; MARRANGHELLO; PEREIRA, 2013). Uma das formas de mitigar este problema é a utilização de técnicas de programação paralela, fazendo o uso de sistemas com múltiplas unidades de processamento ou uma unidade de processamento com múltiplos núcleos (DÍAZ-PERNIL et al., 2013). Em geral, os algoritmos paralelos têm suas bases em algoritmos sequenciais e possuem uma maior complexidade de implementação, contudo, seu uso pode acelerar a execução do algoritmo e possibilitar a implementação deste em aplicações reais (HUQQANI et al., 2013).

1.2 Objetivos

O objetivo principal deste trabalho é a implementação de um método de detecção de faixas centrais, baseado em visão computacional para sistemas embarcados convencionais, operando com imagens em alta definição (1280×720 píxeis) de uma câmera monocular para realizar a detecção de vias com e sem curvatura. O método proposto, adota uma estratégia iterativa de detecção de faixas contínuas e descontínuas, através de uma cadeia de processamento que tem como base a segmentação das sinalizações das vias e a estimação dos píxeis pertencentes às faixas de trânsito.

O método em questão possui uma arquitetura heterogênea, tendo parte do seu método executado de forma sequencial em uma CPU (*Central Processsing Unit*) e o processamento complexo é feito de forma paralela em uma GPU (*Graphics Processing Unit*) embarcada.

Não obstante, a principal contribuição deste trabalho é a proposta de um método de detecção de faixas centrais, baseado em visão monocular capaz de operar em um sistema embarcado convencional. Isto é, um método que realize a detecção de faixas em tempo real e mantenha o compromisso de baixa complexidade computacional e alta taxa de detecção, bem como, a proposta de uma arquitetura de software e hardware para desempenhar o trabalho de detecção.

Para atingir os objetivos principais, é proposto um conjunto de otimizações de computação paralela, que visam otimizar a utilização dos recursos limitados do hardware, viabilizando a solução deste tipo de aplicação em um sistema embarcado.

Os objetivos secundários do trabalho são:

- especificação de um sistema embarcado para realização da tarefa proposta;
- desenvolvimento de uma solução completa embarcada maximizando os recursos do hardware através de implementações paralelas otimizadas;
- aplicação do método em cenários reais baseados em diferentes base de dados;
- levantamento da performance do método em diferentes situações e métricas.

Desta forma, a arquitetura do projeto é composta por três elementos: um sistema de geração de imagens de vias, um sistema embarcado e um computador suporte.

A geração de imagens é responsável por fornecer as informações de entrada ao sistema embarcado. Esta pode ser realizada de duas formas: através de um sensor de imagem

embarcado ou de imagens pré-capturadas de uma base de dados. O sistema embarcado por sua vez, deve desempenhar as tarefas de aquisição e processamento das imagens, retornando as informações obtidas via registro de dados ou telemetria, fornecendo um retorno das faixas detectadas além de outras métricas.

1.3 Trabalhos Relacionados

A detecção de faixas baseadas em sensores óticos utilizam imagens bidimensionais capturadas por uma câmera. Geralmente esta é montada no pára-brisas do veículo e suas imagens são processadas por métodos de visão computacional, cujas informações são repassadas para o controle do veículo. Tal processamento pode ser categorizado em duas principais classes: os baseados em recursos e os em modelos.

A abordagem por modelos propõem um modelo matemático parametrizado que descreve a estrutura das faixas (ZHOU et al., 2010). Esta técnica apresenta alta robustez a ruídos, contudo sua implementação possui limitações, tais como o elevado custo computacional e o conhecimento prévio sobre os parâmetros geométricos da faixas.

Na metodologia baseada em recursos (LEE; MOON, 2018; WANG; TEOH; SHEN, 2004; BORKAR; HAYES; SMITH, 2012a; LI, X. et al., 2014; SON; YOO et al., 2014) a análise da imagem busca detectar os gradientes, os padrões de cores, e outras informações presentes nos píxeis da imagem, afim de reconhecer as sinalizações das faixas de trânsito.

Em Lee e Moon (2018) foi proposto um algoritmo de detecção através do uso de duas regiões de interesse, uma retangular e uma em formato de Λ, contornando os efeitos de ruídos externos e diminuindo o tempo de execução do algoritmo. Este sistema utiliza um filtro de Kalman e uma aproximação de movimento linear para rastrear as faixas, ao mesmo tempo que faz as detecções.

Em Wang, Teoh e Shen (2004) a região de interesse é dividida em seções finitas tornando mais simples o rastreamento de faixas de trânsito. Segmentos de linha extraídos pela transformada de Hough interpolam cada seção através do método *B-Snake*. O método é aprimorado em Chuanxiang (2014) que faz uso de um filtro de Kalman para estimar e refinar o rastreamento das faixas de trânsito.

Borkar, Hayes e Smith (2012a) introduzem um método de detecção com base na operação IPM (*Inverse Perspective Mapping*) e uma técnica de limiar adaptativo é utilizada

para produzir imagens binárias. Além disso, são utilizados formatos pré-determinados para selecionar as possíveis faixas, para eliminar as detecções destoantes é utilizado o algoritmo de RANSAC (*Random Sample Consensus*).

Para garantir uma maior robustez com as varições de luminosidade, Son, Yoo et al. (2014) apresentam um método de detecção de faixas que busca manter a iluminação das faixas constante através de filtros em diferentes espaços de cores.

Focando em obter resultados mais factíveis em aplicações reais, que sejam capazes de atender os requisitos de tempo real, alguns trabalhos discutem técnicas que possibilitam melhorar a performance dos algoritmos. Em Aydin, Samet e Bay (2017) é apresentado através de uma aplicação de segmentação de imagem o uso de técnicas e tecnologias para processamento paralelo, desenvolvendo o sistema com OpenMP em uma CPU com múltiplos núcleos e CUDA (*Compute Unified Device Architecture*) em uma GPU (*Graphics Processing Unit*). Seus experimentos mostram que o uso de GPU com CUDA tem grande capacidade de melhorar e aumentar a performance de aplicações de processamento de imagem em tempo real.

Em Jincheng Li et al. (2019) é proposta uma implementação heterogênea baseada em um algoritmo que executa parte em *multi-threading* em uma CPU e parte paralela em uma GPU, para apresentar um método de Visual SLAM com múltiplas câmeras que atenda aos requisitos de tempo real em uma plataforma embarcada. A implementação mostra os ganhos de velocidade gerados pela paralelização do algoritmo utilizando CUDA, bem como, trata o balanceamento de cargas da GPU, através do *multi-threading* na CPU, resultando em um método eficiente que permite a implementação em aplicações de tempo real.

Outros trabalhos como Afif, Said e Atri (2020) e (ZHI et al., 2019) salientam os ganhos proporcionados pela paralelização das operações e o uso de computação heterogênea para a solução de problemas de processamento de imagem em aplicações de tempo real utilizando CUDA.

1.4 Organização do Trabalho

O presente capítulo apresenta, de forma geral, a proposta do trabalho e seus objetivos específicos, situando a localização deste projeto no cenário atual, são também introduzidos os principais conceitos envolvidos na problemática da detecção de faixas de trânsito e as propostas de implementação e otimização.

O Capítulo 2 expõe através de uma revisão do estado da arte, os principais trabalhos que atacam o problema de detecção de faixas de trânsito. Não obstante, são analisadas as principais contribuições e limitações de cada método considerando o cenário do presente trabalho. Os métodos utilizados são apresentados de forma funcional visando apresentar os principais motivos que levaram a escolha dos algoritmos, seus benefícios e limitações.

O Capítulo 3 discorre os principais conceitos acerca da computação heterogênea situando, de forma didática, o ambiente de desenvolvimento do projeto. São apresentados as características da programação de e arquitetura das GPUs, o seu modelo de programação e as bases hierárquicas deste tipo de sistema. Por fim, é discutido o processamento de imagens utilizando CUDA, apontando as principais otimizações que possibilitam extrair de forma eficiente os recursos do hardware embarcado.

O Capítulo 4 apresenta de forma direta a arquitetura do hardware embarcado proposto para o experimento, discriminando os principais componentes e subsistemas da plataforma embarcada, além disso, discute as implementações e algoritmos propostos neste trabalho. São apresentados os algoritmos, suas implementações e as discussões pertinentes de cada implementação. O método proposto é examinado de forma minuciosa as etapas de inicialização e pré-processamento, extração de características e estimação das faixas, garantindo um detalhamento preciso da arquitetura do software.

O Capítulo 5 expõe os resultados experimentais obtidos no trabalho. Em um primeiro momento é descrito o ambiente de testes utilizados, a base de dados adotada e outras características e métricas do experimento. Então, são analisadas as performances qualitativas e quantitativas do método e as contribuições e resultados obtidos.

O Capítulo 6 sumariza as etapas concluídas do projeto, e os resultados práticos do sistema completo, as métricas de tempo real obtidas, bem como os índices e méritos de desempenho do sistema. São apresentados também, as principais conclusões sobre a metodologia heterogênea otimizada para a viabilização deste tipo de aplicação em um cenário real.

Capítulo 2

Detecção de Faixas

Os algoritmos de detecção de faixas de trânsito seguem normalmente um fluxo comum baseado em três etapas: o pré-processamento da imagem, a extração de características das sinalizações e o refinamento de candidatos às faixas encontradas, como ilustrado no diagrama da Figura 2.1. Na primeira etapa são aplicados filtros e transformadas na imagem de entrada com o intuito de obter uma versão da imagem que contenha pouco ruído e apresente apenas informações relevantes para a detecção. Então, esta nova imagem é processada por métodos de extração de características que buscam identificar pontos na imagem que representem as faixas de trânsito. Por fim, estes pontos encontrados formam um conjunto de candidatos que são analisados e refinados através de estimadores, para obter versões confiáveis das faixas detectadas (BORKAR; HAYES; SMITH, 2012b; LEE; MOON, 2018; VAJAK et al., 2019; BAR HILLEL et al., 2014b).

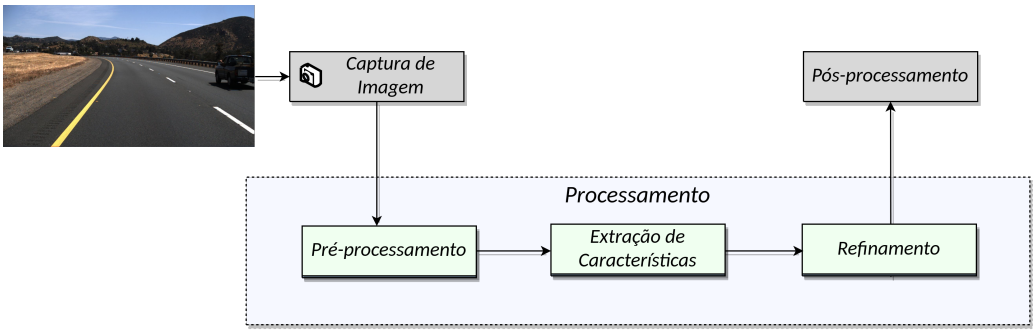


Figura 2.1: Diagrama simplificado do fluxo de processamento de um método de detecção de faixas.

Dito isso, antes de entrar em detalhes do processo de implementação, neste capítulo são apresentados os fundamentos matemáticos dos algoritmos, que serão utilizados para a construção do método de detecção de faixas, bem como, o processos que justificam suas es-

colhas, alguns dos métodos apresentados não são discutidos na etapa de resultado, pois não participaram da implementação final.

2.1 Pré-processamento

Há um número ilimitado de cenários em que os veículos podem ser conduzidos. Cada cena, pode variar com base em combinações de paisagem, tais como, posição do Sol, clima, oclusões, sombras, névoas, texturas da estrada, coloração das vias, etc. Além disso, as imagens podem ser capturadas por câmeras diferentes, de posições, direções e rotações de câmera distintas. Na posição mais comum da câmera, isto é, fixada no pará-brisas do veículo e apontada para frente, há muito conteúdo irrelevante na imagem capturada. Devido a essas variações, uma estratégia muito comum é pré-processar a imagem de entrada e depois aplicar uma técnica de extração de recursos (YENIKAYA; YENIKAYA; DÜVEN, 2013; BAR HILLEL et al., 2014b; NAROTE et al., 2018).

Estratégias de pré-processamento têm sido aplicadas às imagens em vias com o objetivo de:

- reduzir a região a ser processada, evitando erros evidentes ao mesmo tempo em que aumenta o desempenho computacional;
- corrigir a distorção da perspectiva, permitindo algumas técnicas que não funcionariam de outra forma;
- aumentar a confiabilidade dos dados de entrada, assumindo que as marcações da pista não devem mudar abruptamente, em termos de espacialidade, quando comparadas a outros objetos em uma cena.

Além disso, uma conversão para tons de cinza é geralmente aplicada afim de reduzir a quantidade de dados a serem processados, bem como filtros para suavizar a imagem e remover ruídos (NAROTE et al., 2018).

O diagrama da Figura 2.2 ilustra um fluxo de pré-processamento básico adotado para as estratégias de detecção de faixas. Cada um dos componentes deste método serão melhor explicados nas próximas seções.

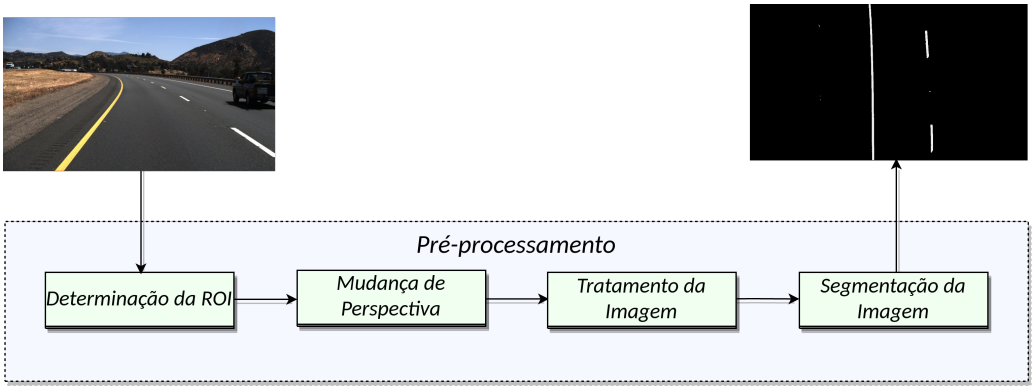


Figura 2.2: Diagrama simplificado do fluxo de pré-processamento de um método de detecção de faixas.

2.1.1 Determinação de uma Região de Interesse (ROI)

A Figura 2.3[1] apresenta um exemplo de imagem adquirida por uma câmera montada no veículo. Como mencionado, espera-se que uma dada imagem de entrada possui uma ROI (*Region of Interest*) que compreende principalmente a estrada, descartando céu, edifícios, partes do próprio carro e outros conteúdos menos relevantes. A definição de uma ROI permite aumentar o desempenho computacional, pois haverá menos dados para processar, e evita erros evidentes, pois objetos irrelevantes podem representar ou adicionar ruído que será processado posteriormente. A Figura 2.4 apresenta na sua região mais clara, interior do retângulo amarelo, uma possível região de interesse para o problema de detecção de faixas de trânsito. Esta ROI busca estabelecer a maior quantidade de informação pertinente, enquanto mitiga as informações externas, tais como: céu, montanhas e regiões que não compreende as vias.

Figura 2.3: Exemplo de imagem utilizada como entrada para os métodos de detecção de faixas (TUSIMPLE, 2018).

[1] as imagens de vias utilizadas para a apresentação deste capítulo são extraídas da base de dados TuSimple (2018)

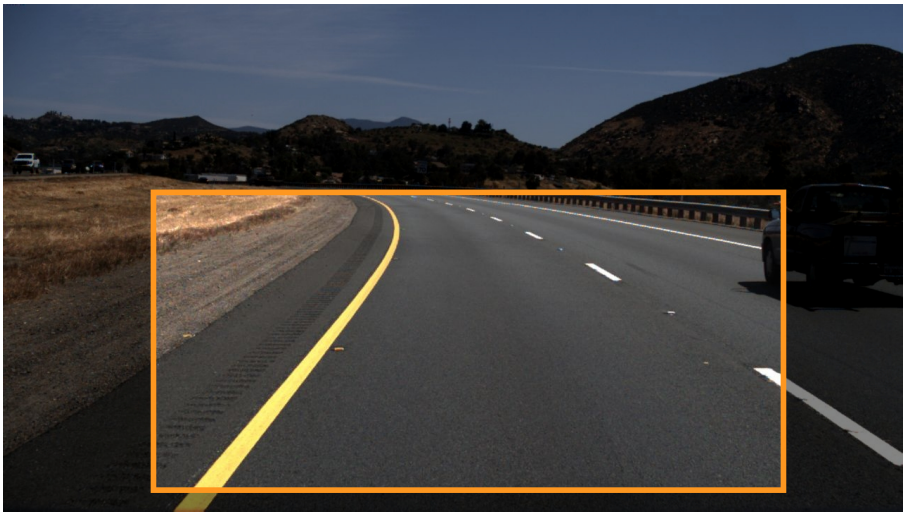


Figura 2.4: Exemplo de uma ROI para a detecção de faixas.

A seleção da ROI pode ser feita estaticamente (LI, Q. et al., 2014) ou dinamicamente (SATZODA; TRIVEDI, 2013). No primeiro caso, há um conhecimento prévio da imagem capturada pela câmera (ou da posição da câmera), então a região é definida a partir desse conhecimento. No segundo, técnicas de processamento são aplicadas afim de determinar essa região. Essas técnicas variam de detecção do céu (ZUO; YAO, 2013), análise de horizonte (YUE; DINGGANG; EAM KHWANG, 2000), estimativa de ponto de fuga (RASMUSSEN, 2004) e outros.

Embora possa parecer melhor ter uma seleção dinâmica, o tempo de processamento gasto nesta tarefa pode ser proibitivo para sistemas de tempo real, especialmente para operações embarcadas, onde os recursos são escassos.

Além disso, em um sistema real, espera-se que a posição da câmera seja conhecida (embora uma calibração possa ser necessária após sua instalação ou possa ser definida no processo de fabricação). Portanto, definir a região de interesse com antecedência pode ser feito facilmente e com baixo custo. Desta forma, apenas determinação da ROI de forma estática será abordada neste trabalho, uma vez que o projeto tem como objetivo a detecção de faixas em uma plataforma embarcada.

2.1.2 Transformação de Perspectiva (IPM)

Muitas vezes a distorção de perspectiva é considerada um ruído adicional que deve ser removido ou atenuado. A imagem capturada por uma câmera voltada para frente sofre inerentemente de uma distorção de perspectiva. Essa distorção é responsável pela quebra de uma variedade de suposições: linhas paralelas (como marcações de pista) não aparecem

paralelas, elementos de largura constante (como largura das sinalizações da pista) apresentam uma largura variável proporcional à distância da câmera entre outros efeitos (YU; JO, 2018).

Muitos dos algoritmos de detecção de faixas funcionam melhor assumindo uma estrada plana, com faixas paralelas e de largura constante e portanto aplicam transformações na imagem buscando obter uma versão de perspectiva que favoreça a detecção. Com base nesta suposição, um IPM é aplicado para obter uma imagem em uma perspectiva que favoreça as características mencionadas (SIVARAMAN; TRIVEDI, 2013; LI, Q. et al., 2014; ALY, 2008; DENG; HAN, 2013; MUTHALAGU; BOLIMERA; KALAICHELVI, 2020). Um exemplo de resultado é apresentado na Figura 2.5, onde a operação IPM foi aplicada à imagem da Figura 2.3 afim de obter uma melhor perspectiva.

A homografia desejada é uma transformada projetiva com oito graus de liberdade, que pode ser restringida por uma variedade de características da imagem (HARTLEY; ZISSERMAN, 2004). Os oito graus de liberdade podem ser mais facilmente restringidos por correspondências de quatro pontos, sendo três deste não colineares, permitindo que a homografia desejada seja resolvida diretamente (HARTLEY; ZISSERMAN, 2004).

Figura 2.5: Exemplo de imagem em BEV obtida pela operação IPM na Figura 2.3.

O IPM é um mapeamento aplicado a imagem e, na maioria dos casos, assume uma estrada plana fixa. O resultado do IPM é frequentemente chamado de "visão panorâmica"(ou *Bird's Eye View* - BEV). Para aplicar o IPM, uma matriz de homografia H de terceira ordem deve ser encontrada. Esta mapeia as coordenadas do plano de estrada (x, y) para os pontos da imagem (u, v) em coordenadas homogêneas, conforme a Equação 2.1.

$$\begin{bmatrix} \lambda u \\ \lambda v \\ \lambda \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.1}$$

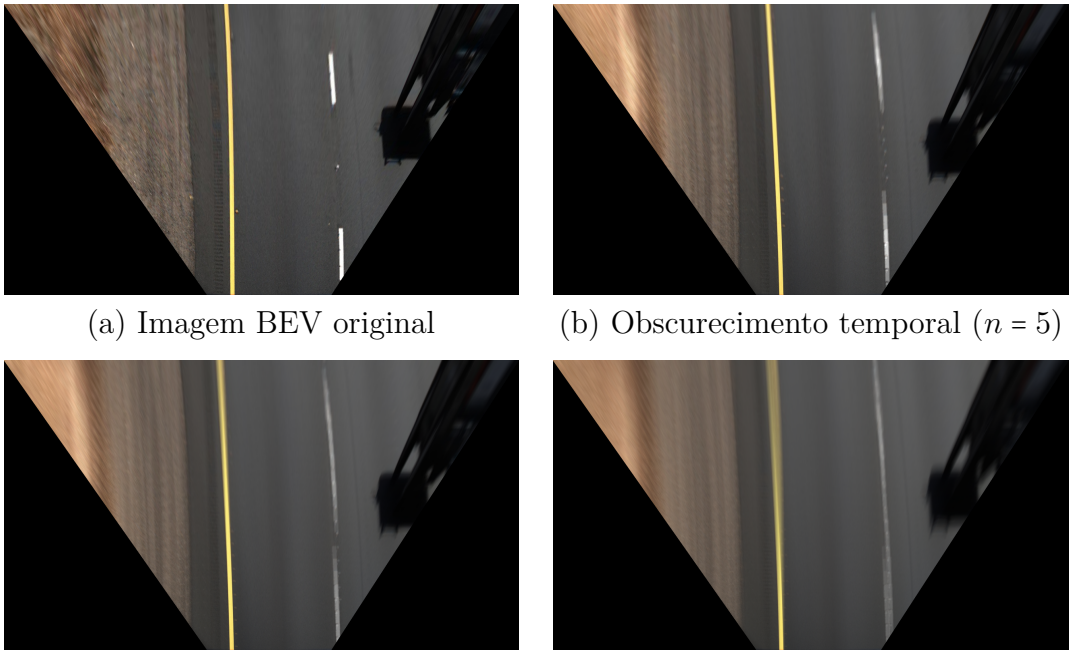
onde $H = [h_{ij}]$ representa a homografia e λ é um escalar. Na literatura, alguns autores (BODIS-SZOMORU; DABOCZI; FAZEKAS, 2007; HUANG et al., 2018; MUTHALAGU; BOLIMERA; KALAICHELVI, 2020) fazem o cálculo de H de maneira offline (fora da rotina de processamento), baseado apenas em uma calibração simples, este método é descrito na Seção 2.1.5. Outros, tais como Bodis-Szomoru, Daboczi e Fazekas (2007) calculam a matriz H em tempo de execução. A obtenção desta matriz ainda em tempo de execução, demanda um processamento elevado, o que não é desejável para sistemas que devem operar com requisitos de tempo real. A calibração automática que permite o cálculo correto de H tem sido explorada por de Paula e Jung (2015), mas exige uma sequência de acionamento específica, que pode nem sempre estar disponível ou possível de ser adquirida.

Neste trabalho, é utilizada uma matriz H constante para cada um dos cenários de teste. A obtenção desta matriz e seus valores são melhor descritos na Seção 2.1.5.

2.1.3 Obscurecimento Temporal

Nesta etapa é calculada a média de um conjunto de quadros anteriores para obter uma versão mais confiável das marcações da pista. A Figura 2.6 apresenta o resultado deste método para diferentes quantidades de quadros, definido por n.

Como as marcações das faixas não mudam abruptamente em um curto espaço de tempo, principalmente quando comparadas a outros objetos (outros veículos, pedestres, etc.), uma quantidade razoável de quadros anteriores são integrados para aumentar a confiança dessas sinalizações. Espera-se que este procedimento aumente as marcações de pista e mitigue objetos em movimento, diminuindo o efeito de possíveis fontes de ruído no processo posterior. Da mesma forma, esta técnica também é aplicada afim de dar às marcações de faixa tracejada a aparência de uma linha longa e contínua (BORKAR; HAYES; SMITH; PANKANTI, 2009). A taxa de quadros de entrada e a velocidade do veículo podem variar, portanto, seu impacto deve ser considerado para integração temporal. Esta técnica também chamada de obscurecimento temporal, permite melhorar a qualidade de detecção para situações onde as sinalizações são muito desgastadas e degradadas (SON; LEE; KUM, 2018a).



(a) Imagem BEV original (b) Obscurecimento temporal ($n = 5$)

(c) Obscurecimento temporal ($n = 10$) (d) Obscurecimento temporal ($n = 20$)

Figura 2.6: Exemplos da operação de obscurecimento temporal sobre um conjunto de imagens em perspectiva BEV para diferentes valores de n.

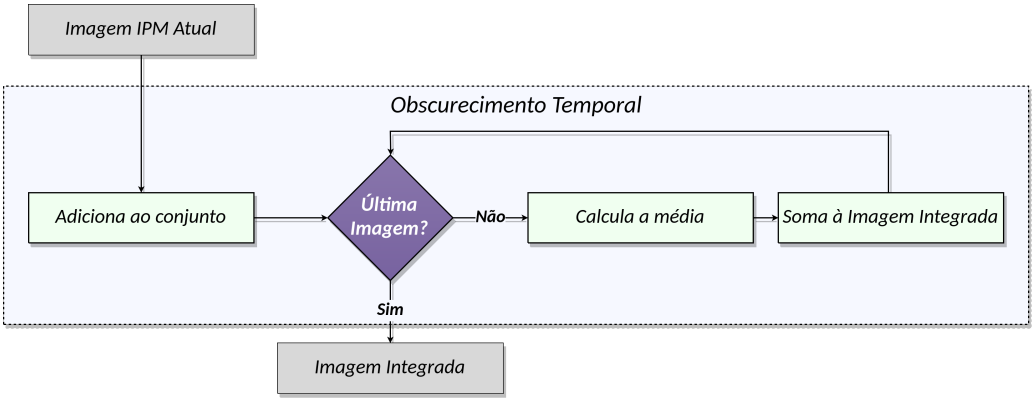


Figura 2.7: Diagrama da operação de obscurecimento temporal.

Para elucidar o método de obscurecimento temporal seu fluxograma é apresentado no diagrama da Figura 2.7, onde a operação é realizada para um número n pré-determinado de quadros. Além disto, é possível alterar também a taxa de obscurecimento de cada quadro, alterando o peso com que os quadros mais antigos impactam a imagem atual.

2.1.4 Tratamento da Imagem

Neste trabalho, optou-se por utilizar técnicas de pré-processamento estático. Para melhorar o resultado obtido nesta etapa e nas seguintes, são necessários tratamentos na imagem que maximizem as informações desejadas, mitigando ruído externo e reduzindo a quantidade

de informação desnecessária. Então, são discutidos outras formas de pré-processamento tais como, a mudança de espaços de cores, redimensionamento e filtros para remoção de ruído.

Conversão para Escala de Cinza

O objetivo desta etapa é encontrar as sinalizações das faixas e mais especificamente as marcações da pista atual. Todas as imagens do espaço de cores RGB são convertidas para tons de cinza com o único objetivo de simplificar o problema e o algoritmo, uma vez que a quantidade de informação armazenada e processada para cada imagem é reduzida (ZHICHENG ZHANG, 2019; HUANG et al., 2018; LI, W. et al., 2018; NAROTE et al., 2018). Devido a essa conversão, as imagens utilizadas nas próximas etapas possuem um único canal e 8 bits por píxel.

Quando a câmera captura uma imagem colorida, ela contém três componentes de cores R, G e B, (Figura 2.8) que são facilmente interferidos pelo brilho externo, além de ocupar três vezes mais espaço que um único canal. A imagem em tons de cinza possui apenas um componente e ocupa um terço do espaço de uma imagem colorida. A Equação 2.2 apresenta o equacionamento genérico para a obtenção de uma imagem a partir de uma fonte RGB

$$I(x,y) = R(x,y) \times w_R + G(x,y) \times w_G + B(x,y) \times w_B. \tag{2.2}$$

São atribuídos diferentes valores de peso (w_R, w_G e w_B) aos três canais originais ($R(x,y)$, $G(x,y)$, $B(x,y)$), como apresentado na Equação 2.3 (ZHICHENG ZHANG, 2019). Isto possibilita a obtenção de uma imagem em tons de cinza

$$I(x,y) = R(x,y) \times 0,299 + G(x,y) \times 0,587 + B(x,y) \times 0,114, \tag{2.3}$$

onde $R(x,y)$, $G(x,y)$, $B(x,y)$ são os três canais da imagem original e suas componentes são escaladas e somadas par obter uma imagem em tons de cinza de um único canal, conforme apresentado na Figura 2.8b.

Outra forma de obter uma imagem em tons de cinza e realçar as marcações das faixas de trânsito, é proposta por Yoo, Yang e Sohn (2013) e também utilizada em Wu, Wang e Wang (2019). Este método faz uso do espaço de cores YCbCr ao invés do espaço RGB convencional, portanto é necessário antes, realizar a conversão entre os espaços de cores, conforme descrito na Equação 2.4)

(a) Imagem original (RGB) (b) Imagem em escala de cinza

Figura 2.8: Exemplos da operação de conversão do espaço de cores RGB para escala de cinza.

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0,299 & 0,587 & 0,144 \\ -0,169 & -0,331 & 0,500 \\ 0,500 & -0,419 & -0,081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}. \tag{2.4}$$

Então, a Equação 2.5 apresenta a conversão para a escala de cinza com realçamento das sinalizações das faixas (YOO; YANG; SOHN, 2013; WU; WANG; WANG, 2019).

$$I(x,y) = Y(x,y) \times 0,9996 + (C_b(x,y) - 128) \times -0,5608 + (C_r(x,y) - 128) \times 0,4127 \tag{2.5}$$

onde $Y(x,y)$, $C_b(x,y)$ e $C_r(x,y)$ representam respectivamente a luminância, a crominância (C_b é a diferença do azul e C_r do vermelho) no píxel (x,y). Convertendo os parâmetros para a escala RGB encontra-se os valores apresentados na Equação 2.6 (WU; WANG; WANG, 2019).

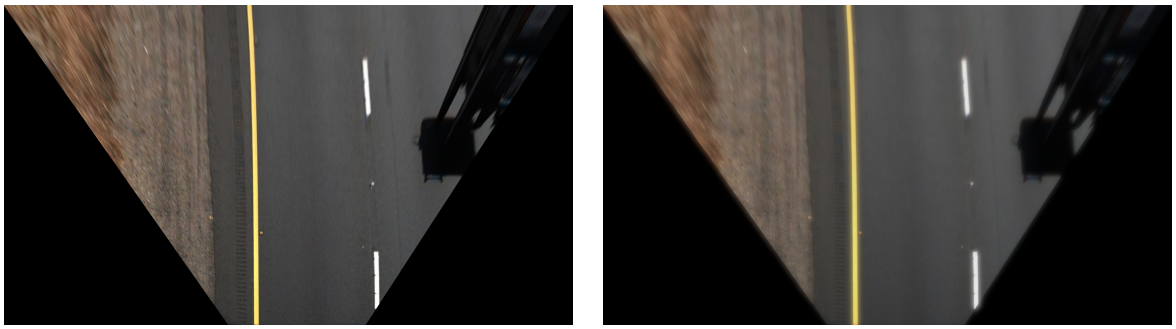
$$I(x,y) = R(x,y) \times 0,6 + G(x,y) \times 0,6 + B(x,y) \times -0,2 \tag{2.6}$$

Filtro Gaussiano Progressivo

Como a superfície da estrada não é perfeitamente lisa e o asfalto costuma apresentar irregularidades, há muitos detalhes inúteis na imagem da região das faixas de trânsito. No entanto, não é necessário tal nível de detalhamento, apenas uma representação simplifica do que a superfície da estrada contém é suficiente e mais adequado. Não é desejável detectar linhas que tenham uma representação mais fina do que, por exemplo, metade da largura da faixa de trânsito. Os detalhes capturados tornam-se ruídos para o algoritmo, então para eliminar a maior parte deles pode ser aplicado um filtro gaussiano à imagem. Como os detalhes são mais

nítidos à medida que se aproxima da câmera, o desfoque não precisa se aplicado forma idêntica a toda a imagem, portanto, adotado-se um tamanho do núcleo que diminui conforme a distância, iniciando com um núcleo 9×9, depois 7×7, 5×5 e finalmente, 3×3 para a zona onde nenhuma linha interessante deve ser encontrada (ZHICHENG ZHANG, 2019).

Na Figura 2.9, é mostrado um exemplo de aplicação desse desfoque gaussiano progressivo.



(a) Imagem original (BEV) (b) Imagem em com desfoque gaussiano

Figura 2.9: Exemplo da aplicação do desfoque progressivo por filtro gaussiano em uma imagem BEV.

2.1.5 Calibração Offline

A presente seção sumariza todos os fundamentos teóricos apresentados neste capítulo, fazendo a associação destes em uma ordem funcional, como a apresentada na Figura 2.10. Apesar de não apresentar novos conceitos, há o aprofundamento em algumas operações, em especial a obtenção da ROI através da IPM estática.

Desta forma, a imagem de entrada é convertida para tons de cinza, uma vez que apenas as intensidades de píxeis são analisadas para gerá-los. Em segundo lugar, uma ROI é definida para remover partes irrelevantes da imagem transformando-a em uma perspectiva BEV mais adequada para a operação de detecção das faixas. São também aplicadas as demais operações apresentadas.

Como mencionado nas seções anteriores, optou-se por fazer diversas operações de forma estática (ou offline). O diagrama da Figura 2.10 ilustra este processo de calibração offline executada para uma dada imagem de entrada.

Para obter a imagem em tons de cinza são utilizadas as conversões de escala de cinza apresentadas na Seção 2.1.4. A IPM é aplicada afim de reduzir a distorção da perspectiva, resultando em uma imagem BEV. Para encontrar a matriz de homografia, uma calibração

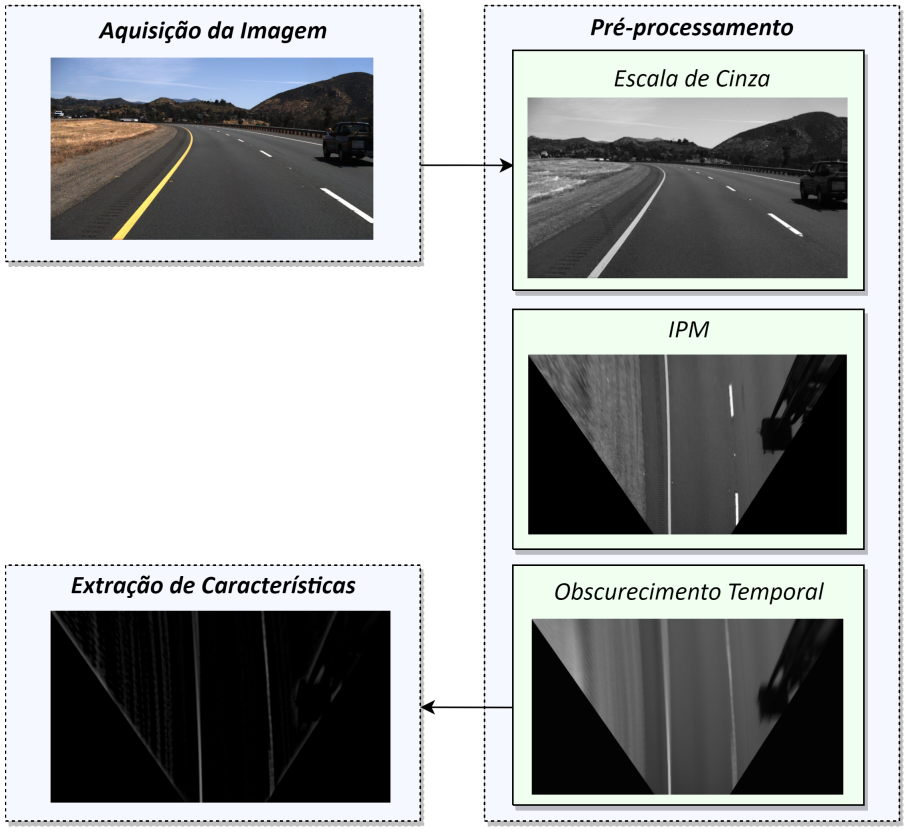


Figura 2.10: Diagrama simplificado da operação de calibração offline.

offline é realizada. Para aplicar o IPM, assume-se um plano constante ao longo dos quadros de entrada, usando uma matriz de homografia estática, ou seja, uma matriz contante previamente obtida.

Na imagem BEV as marcações da pista tendem a ter largura constante. No entanto, devido à inclinação da estrada e solavancos casuais, a constância da largura das marcações da faixa pode ser temporariamente perdida, pois a suposição do plano de fixo pode variar.

Para encontrar a matriz de homografia é utilizada uma imagem onde o carro está no centro da pista (para cada base de dados e resolução é escolhida uma imagem). Nesta, quatro pontos são selecionados: eles compreendem os quatro pontos onde os limites da pista (ou suas extensões de linha, no caso de uma imagem com marcações de pista tracejada) cruzam a região de interesse. A matriz de homografia é calculada de forma que esses quatro pontos sejam mapeados em um quadrilátero (plano da pista). Os oito pontos descritos podem ser melhor visualizados na Figura 2.11, os pontos em questão são os vértices dos planos roxo e verde respectivamente.

A Tabela 2.1 apresenta o mapeamento das coordenadas para a transformação de perspectiva. O objetivo é obter uma imagem BEV como a apresentada na Figura 2.5. Para

Figura 2.11: Em roxo o plano do solo fixo e em verde o plano desejado da transformação IPM.

obter a matriz da perspectiva é realizado o cálculo através da Equação 2.7 originada da Equação 2.1 (MUTHALAGU; BOLIMERA; KALAICHELVI, 2020; HUANG et al., 2018),

Tabela 2.1: Mapeamento dos pares de coordenadas para a transformação de perspectiva

Imagem Origem	Imagem Destino
(150, 719)	(540, 719)
(540, 350)	(540, 1)
(770, 350)	(770, 1)
(1100,719)	(770, 719)

$$\begin{bmatrix} \lambda u_i \\ \lambda v_i \\ \lambda \end{bmatrix} = H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}, \tag{2.7}$$

onde (u_i, v_i) com $i = 0, 1, 2, 3$ são as coordenadas dos vértices dos quadriláteros na imagem destino. Ao passo que, (x_i, y_i) com $i = 0, 1, 2, 3$ são as coordenadas dos quadriláteros da imagem origem. Então, calcula-se a matriz H para a transformação da imagem, conforme a Equação 2.8

$$I(x, y) = O\left(\frac{H_{11}x + H_{12}y + H_{13}, H_{21}x + H_{22}y + H_{23}}{H_{31}x + H_{32}y + H_{33}, H_{31}x + H_{32}y + H_{33}}\right), \tag{2.8}$$

onde $I(x, y)$ são as coordenadas dos píxeis na imagem transformada e $O(x, y)$ são as coordenadas dos píxeis na imagem origem.

Desta forma, operando a Equação 2.8 com os oito pontos (plano da pista e plano desejado) é possível obter uma matriz de homografia H. Na Equação 2.9 é apresentado um

exemplo de matriz de homografia obtida com os pontos da Tabela 2.1. Esta matriz, é a utilizada na geração da imagem BEV da Figura 2.5,

$$H = \begin{bmatrix} -0,05 & -2,863 & 1002 \\ 0 & -4,0855 & 1429 \\ 0 & -0,0043 & 1 \end{bmatrix}. \tag{2.9}$$

Após a geração de uma imagem BEV é possível realizar o aprimoramento das marcações de faixas e a remoção de algumas imperfeições. Um dos métodos de realizar este aprimoramento é a utilização da técnica de integração de imagem e obscurecimento temporal descrito na Seção 2.1.3.

Assim, tendo uma imagem em perspectiva BEV com as sinalizações bem delimitadas e um espaço de cor adequado para a operação, é possível realizar procedimentos de tratamento de imagem como os descritos na Seção 2.1.4. O quadro obtido depois desta etapa de pré-processamento estará adequado para a extração de características e criação dos mapas de características.

2.2 Extração de Características

Após a etapa de pré-processamento, geralmente há uma etapa de extração de características. Nesta fase, o objetivo é extrair pistas visuais que provavelmente ajudarão um determinado algoritmo a detectar as faixas na imagem. Normalmente, o quadro pré-processado é segmentado com base nas características extraídas. Para isso, podem ser consideradas diversas características, tais como as bordas, cores, texturas, estruturas, etc.

As bordas são definidas por mudanças na função de intensidade na imagem (MARR; HILDRETH, 1980). As marcações da pista devem ter bordas claras, porque são projetadas para serem facilmente distinguidas pela visão humana. O problema surge no mundo real, onde as marcações da pista ficam desbotadas depois de anos, os carros podem obstruí-las, outros objetos podem gerar bordas e as sombras podem reduzir o contraste esperado. No entanto, algoritmos de detecção de bordas têm sido amplamente utilizados e, como pode ser visto em alguns trabalhos, trata-se de uma técnica convencional para a tarefa de detecção de faixas (BAR HILLEL et al., 2014b).

Os algoritmos convencionais de detecção de bordas, tais como o filtro de Sobel (WANG; DAHNOUN; ACHIM, 2012), filtro de Canny (YAN; LI, 2017), filtros direcionáveis (GUO; MITA, 2009) e filtros manuais (NIETO; CORTES et al., 2012) são comumente utilizados para a detecção de faixas e se baseiam no padrão de intensidade DLD gerado pela diferença na coloração do asfalto (escura) e das marcações das faixas (clara).

Outra abordagem é a utilização dos padrões de cores para a obtenção de informações e características da posição das faixas. Contudo, como discutido anteriormente, este tipo de abordagem demanda processamento extra devido a quantidade mais elevada de dados adicionais, isto em geral, limita sua aplicação em sistemas embarcadas de tempo real. É importante dizer que essa abordagem tem se mostrado eficaz em estradas rurais, onde a cor da estrada é uma característica discriminativa mais eficiente do que em ambientes urbanos. Mais comumente, as informações de cores são usadas para agrupar píxeis para classificação posterior. Alguns autores utilizam algoritmos supervisionados, como redes neurais baseadas em histogramas RGB (LEE; MOON, 2018) e máquinas de vetores de suporte (SVM) baseadas em píxeis RGB (ZHANG; HOU; ZHOU, 2005). Outros autores usam métodos não supervisionados, como o algoritmo ISODATA baseado em histogramas de matiz e saturação (SOQUET; AUBERT; HAUTIÈRE, 2007), um UNSCARF modificado denominado RSURD baseado sobre os valores do HSV (GAO; QIJUN; MOLI, 2007) entre outros.

Muitas abordagens combinam conjuntos de características como Chuanxiang (2014) e Satzoda e Trivedi (2015). As abordagens baseadas em textura não são comuns para estradas urbanas e estruturadas, mas junto com as baseadas em cores, são bastante usadas para ambientes rurais e não estruturados. Filtros Gabor multi-escala foram utilizados em Rasmussen (2004) para encontrar o ponto de fuga. Redes neurais foram empregadas por Neven et al. (2018) para classificação e segmentação das faixas.

Após a extração de características, a segmentação geralmente é realizada, principalmente por meio de uma técnica de binarização, resultando no que é chamado de mapa de características como em Berriel et al. (2015). Os limiares são distribuídos por todos esses métodos e seu ajuste fino é uma limitação conhecida. Limiares geralmente restringem esses sistemas para funcionar sob condições específicas (como aqueles com sombras fortes, mudança abrupta de iluminação, marcações de faixa desbotadas, etc.) exigindo ajuste adicional ou recalibração. Técnicas de limiar adaptativo também foram exploradas (WU; LIN; CHEN, 2009; WU; WANG; WANG, 2019). Mais recentemente, o aprendizado de máquina e aprendizado profundo também

estão sendo utilizados para extrair características e realizar segmentação de pista (HUVAL et al., 2015; NEVEN et al., 2018), e pentender a cena fornecendo um retorno para o sistema de controle (CHEN et al., 2015).

Neste trabalho, diferentes extratores de características são utilizados para gerar vários mapas de evidências. Esses mapas são então combinados, sob demanda, por cada módulo de processamento, de acordo com as combinações de características que mais contribuem na tarefa de detecção de faixas. São apresentados os mapas I^{ALT} (*Adaptive Luminance Threshold Map*), I^{SRF} (*Step Row Filter Map*), I^{DOG} (*Difference of Gaussians Map*), I^{UCF} (*Unilateral Correlation Filter Map*), e I^{DLD} (*DLD Filter Map*).

2.2.1 Mapa de Limiar Adaptativo de Luminância (I^{ALT})

Devido a variação de ambientes, a coloração do asfalto e das faixas variam como apresentado nas Figuras 2.12.

Figura 2.12: Exemplos de coloração de asfalto e das faixas da base de dados TuSimple (TUSIMPLE, 2018).

Assim, limiares fixos para a determinação de um mapa de característica não é suficiente, não desempenhando bem na identificação das sinalizações. Para resolver isto, é proposto um limiar adaptativo pela média de iluminação na ROI, extraído através da Equação 2.10 (WU; WANG; WANG, 2019),

$$L_{med} = \frac{\sum_{x=0}^{W_{ROI-1}} \sum_{y=0}^{H_{ROI-1}}}{W_{ROI} H_{ROI}} \tag{2.10}$$

onde W_{ROI} e H_{ROI} são respectivamente os valores da largura e altura da ROI em píxeis. Com base no valor de L_{med} os píxeis que são associados as sinalizações das faixas de trânsito são extraídos utilizando a Equação 2.11.

$$I(x,y) = \begin{cases} I(x,y) & \text{se } T_L \leq I(x,y) \leq T_U \\ 0 & \text{caso contrário} \end{cases} \tag{2.11}$$

onde T_L e T_U são os limiares sobre quatro condições de brilho e são especificados nas Equações 2.12 e 2.13, respectivamente.

$$L_T = \begin{cases} 60, & \text{se } 0 \leq L_{med} \leq 25 \\ 115, & \text{se } 25 < L_{med} \leq 40 \\ 125, & \text{se } 40 < L_{med} \leq 70 \\ 135, & \text{se } 70 < L_{med} \leq 100 \\ 145, & \text{caso contrário} \end{cases} \tag{2.12}$$

$$T_U = \begin{cases} 220, & \text{se } 0 \leq L_{med} \leq 25 \\ 235, & \text{se } 25 < L_{med} \leq 40 \\ 240, & \text{se } 40 < L_{med} \leq 70 \\ 250, & \text{se } 70 < L_{med} \leq 100 \\ 255, & \text{caso contrário} \end{cases} \tag{2.13}$$

A Figura 2.13 apresenta o resultado do mapa I^{ALT} para diferentes marcações de faixas e coloração de vias.

2.2.2 Mapa do Filtro de Linhas (I^{SRF})

As sinalizações das faixas são geralmente áreas mais claras do que seus arredores (asfalto). Neste mapa, características das sinalizações de faixa são detectadas na imagem original em tons de cinza através de uma filtragem de linhas definido pela Equação 2.14, conforme apresentado em Nieto, Arrospide e Salgado (2011),

$$y_i = 2x_i - (x_{i-\tau} + x_{i+\tau}) - |x_{i-\tau} + x_{i+\tau}|, \tag{2.14}$$

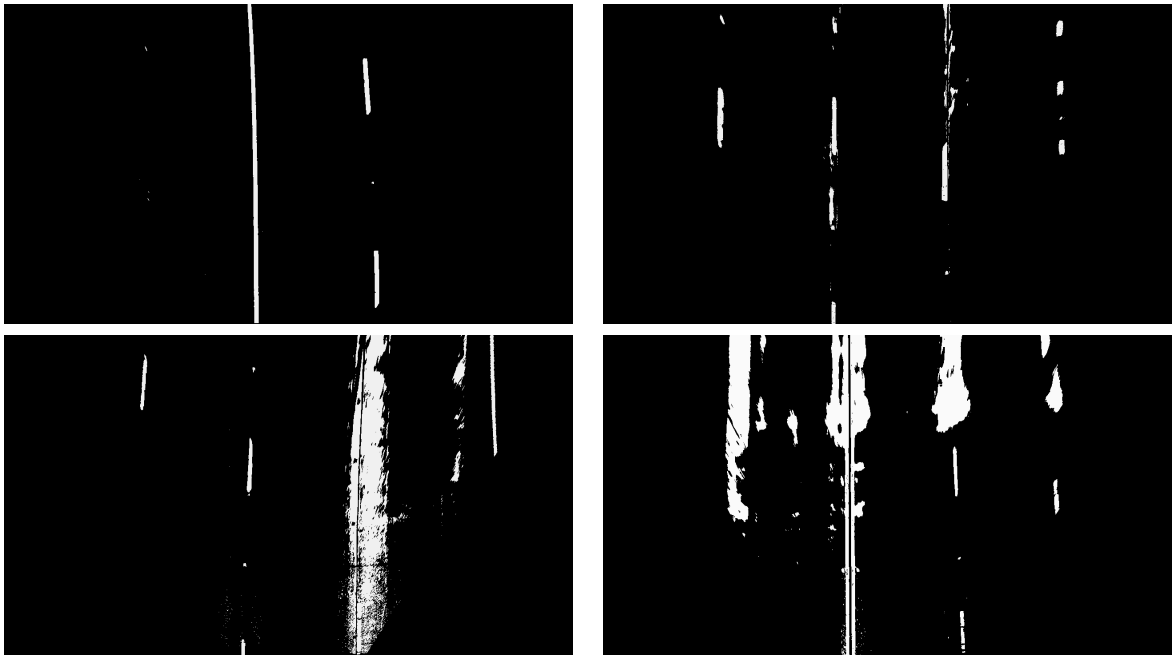


Figura 2.13: Resultados obtidos através do mapeamento de limiar adaptativo de iluminação (mapa I^{ALT}).

τ é considerado grande o suficiente para ultrapassar o tamanho das marcações de faixa dupla. Adicionalmente, este filtro é aplicado na imagem original, ajustando linearmente τ de acordo com o eixo vertical. Como resultado, espera-se que este mapa capture as sinalizações das faixas, além de ser robusto contra sombras, buracos e recapeamento de asfalto. É importante notar que dependendo de τ alguns objetos, por exemplo carros brancos, podem gerar características indesejáveis neste mapa.

2.2.3 Mapa da Diferença de Gaussianas (I^{DOG})

Com base na mesma suposição (as sinalizações das faixas são mais claras do que o asfalto), neste mapa, as evidências são encontradas usando uma diferença horizontal de gaussianas (MARR; HILDRETH, 1980) (DOG), equivalente a um chapéu mexicano horizontal com abertura central e média igual a largura da faixa. A operação de DOG é aplicado na imagem em tons de cinza em perspectiva BEV aproveitando a invariabilidade da largura da faixa de acordo com um limiar. Para este mapa de características, espera-se capturar as marcações de faixa na imagem IPM, mas também se espera ruído de sombras, carros, meio-fios e outros. Mesmo assim, espera-se que este mapa de recursos aumente a robustez geral quando combinado com outros mapas de recursos (HUANG et al., 2018).

2.2.4 Mapa do Filtro de Correlação Unilateral (I^{UCF})

Com o intuito de reduzir a complexidade computacional, os operadores de detecção de bordas convencionais tais como Sobel (WANG; DAHNOUN; ACHIM, 2012), Canny (YAN; LI, 2017) não são utilizados neste trabalho, devido a carga computacional elevada exigida no processamento. Desta forma é utilizado um filtro de correlação para extração unilateral proposto em Zhicheng Zhang (2019). O algoritmo de filtragem de correlação pode ser expresso conforme a Equação 2.15

$$g = f * h, \tag{2.15}$$

onde f é a imagem de entrada e h o núcleo do filtro e g a imagem de saída. Portanto, tem-se a Equação 2.16

$$f * h(\tau) = \int_{-\infty}^{\infty} f(\tau)h(t+\tau)dt, \tag{2.16}$$

que é expressa em uma notação mais intuitiva para imagens na Equação 2.17

$$g(i,j) = \sum_{k,l} f(i+k, j_l) \cdot h(k,l). \tag{2.17}$$

Então, é obtido um filtro de correção 3×3 (Equação 2.18) para o processamento da imagem.

$$K = \begin{bmatrix} -3 & 0 & 3 \\ -3 & 0 & 3 \\ -3 & 0 & 3 \end{bmatrix} \tag{2.18}$$

O filtro de correção da Equação 2.18 é utilizado para extrair a imagem em uma única borda, onde as amplitudes são tomadas como -3 e 3 respectivamente aprimorando a linha das bordas.

2.2.5 Mapa do Filtro de Transição DLD (I^{DLD})

A ideia vem da observação das sinalizações das faixas. É possível notar que uma faixa é uma zona clara cercada por uma área mais escura (suposição também adotada nos outros extratores). Este padrão é denominado DLD (*Dark-Light-Dark*) e é a característica principal

e mais visível que distingue uma linha do resto da superfície da estrada (ZHICHENG ZHANG, 2019).

A implementação do algoritmo que extrai as transições DLD é baseada em Felisa e Zani (2010). Ele foi modificado para se adaptar ao cenário do projeto, ou seja, imagens frontais BEV. A partir da imagem inicial, obtêm-se uma nova, definida conforme a Equação 2.19

$$DLD(i,j) = max\left[min\left[I(i,j) - I(i,\frac{ks}{2}), I(i,j) + I(i,\frac{ks}{2})\right], 0\right]. \tag{2.19}$$

O tamanho do núcleo DLD, ks, é constante e este fato é essencial, uma vez que a dimensão do núcleo deve ser o mais próximo possível do tamanho em píxeis da linha que deve ser encontrada. Como mencionado, a imagem BEV busca exatamente esta característica de largura fixa para as faixas de trânsito.

2.3 Estimação de Faixas

A estimação de faixas é o processo de ajustar um modelo de faixa em uma entrada (por exemplo, a imagem de uma câmera monocular). Na literatura, o termo estimação de faixas pode ser encontrado com dois significados distintos: um mais amplo, que se refere a todo o processo de pré-processamento, extração de características e ajuste do modelo (SATZODA; TRIVEDI, 2014); e um mais restrito, referindo-se apenas ao ajuste do modelo. Neste trabalho, a estimação de faixas faz referência a seu significado mais restrito, o ajuste de modelo. De forma intercambiável, a detecção de faixa tem sido utilizada no mesmo sentido de estimativa de faixa na literatura (WANG; TEOH; SHEN, 2004).

Nas etapas anteriores, a imagem foi pré-processada e as características foram extraídas formando um mapa de características. Em geral, a estimativa da faixa é geralmente realizada com base no mapa final (mapa obtido da combinação dos outros). Este processo pode ser feito na perspectiva original ou no mapa de características baseado em IPM. Como o modelo de ajuste na perspectiva BEV tem se mostrado uma solução eficaz, ele tem sido amplamente utilizado (YENIKAYA; YENIKAYA; DÜVEN, 2013).

Em Reichenbach et al. (2018) são analisadas diversas formas de realizar a estimação de faixas de trânsito. Em especial, são analisadas os desempenhos destes métodos em arquiteturas de hardwares embarcados. A análise foi feita com os quatro principais métodos de estimação de faixas utilizados para sistemas embarcados. Os resultados apresentados em Rei-

chenbach et al. (2018), indicam que a abordagem de janelas deslizantes e RL (*Random Lines*) são as abordagens com maior precisão e acurácia, contudo, no requisito de tempo de processamento a estratégia de janelas deslizantes apresentou um baixo custo computacional, sendo a de execução mais rápida em relação a outros métodos para estimação de faixas.

2.3.1 Detecção de Faixas através de Janelas Deslizantes

Para este algoritmo, um histograma de coluna da imagem de entrada binária é calculado. Os dois pontos na direção horizontal com as maiores intensidades no histograma são usados como pontos de partida para os dois primeiros sensores virtuais (representados respectivamente pelos retângulos azul e verde na parte inferior da Figura 2.14). Dentro do sensor atual (definida por uma altura e largura constantes) as coordenadas horizontais de todos os píxeis brancos são calculadas, e a densidade de píxel é obtida de um novo histograma de coluna e a posição do sensor seguinte é determinada. A coordenada vertical é simplesmente deslocada para cima pela altura de uma janela. Os pontos de maior densidade de cada sensor (círculos internos aos retângulos na Figura 2.14) formam um conjunto de pontos de maior probabilidade de representar as faixas de trânsito (CAO et al., 2019a; CHAN; LIN; CHEN, 2019; REICHENBACH et al., 2018).

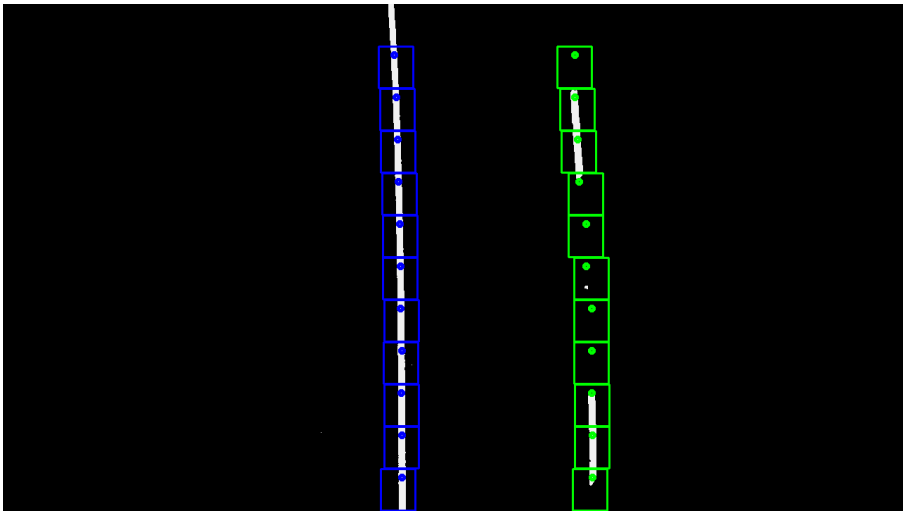


Figura 2.14: Exemplo da operação do algoritmo de janelas deslizantes na estimação de faixas de trânsito em um mapa de características.

Desta forma, o algoritmo de janelas deslizantes é o adotado para a realização da estimação de faixas de trânsito no atual projeto. Para melhor entendimento o método é subdividido em três etapas: histograma de coluna, sensores virtuais e reposicionamento.

Histograma de Coluna

O histograma de coluna é a operação central do método de estimação de faixas de trânsito através de janelas deslizantes. Esta operação consiste em realizar, para cada coluna de uma imagem, o somatório dos elementos que pertencem a esta coluna (CAO et al., 2019b; REICHENBACH et al., 2018). Ao final da iteração ao longo da imagem é obtido um vetor que possui dimensão igual a largura da imagem, em que cada elemento i possui a soma de todos os elementos da coluna i da imagem. Esta operação é melhor ilustrada através da Figura 2.15, onde $i = 0, 1, 2, 3, 4, 5$.

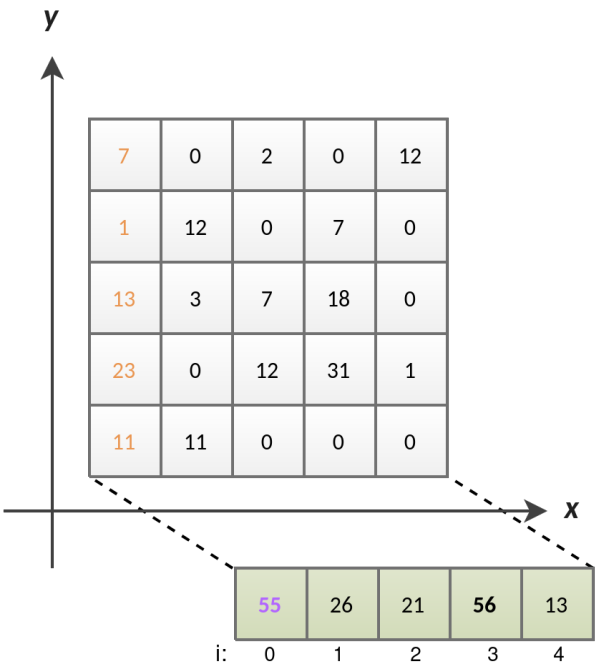


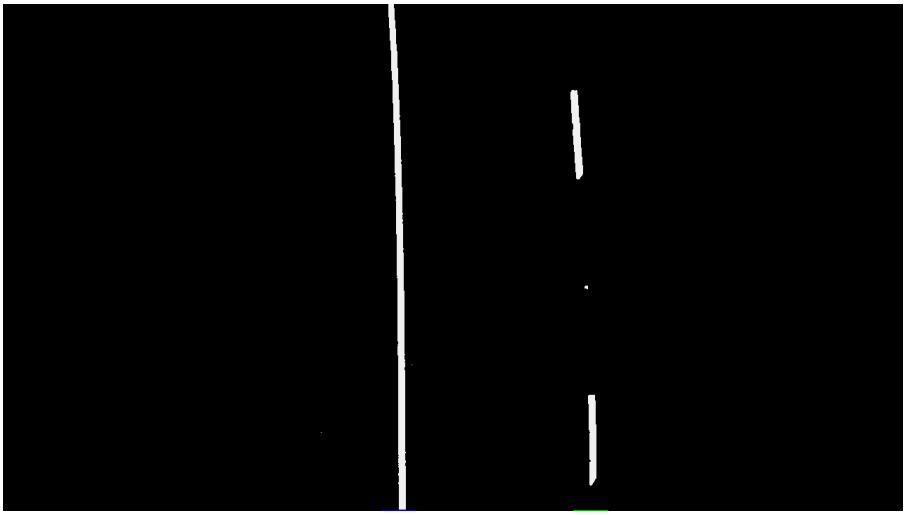
Figura 2.15: Exemplo da operação de histograma de coluna em uma matriz.

A Figura 2.15 apresenta uma matriz 5×5 que é utilizada para facilitar a visualização da operação. Cada elemento da matriz (quadrados cinzas) deve ser iterado para formar o vetor de intensidade (quadrados em verde). Um exemplo de iteração é apresentada na primeira coluna da matriz, as intensidades (em laranja) dos píxeis são somados um a um e resultam no primeiro elemento do vetor de intensidades (valor em lilas). Esta operação é realizada ao longo de toda a matriz resultando no vetor da última linha que contêm as intensidades de cada coluna.

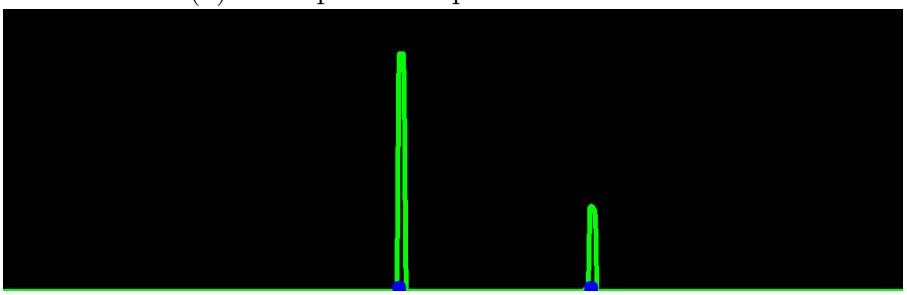
A primeira etapa do método de janelas deslizantes utiliza o histograma de coluna para extrair as regiões de maior probabilidade de conter as faixas de trânsito (MUTHALAGU; BOLIMERA; KALAICHELVI, 2020). Para tal, é utilizada como entrada, um mapa de característica (imagem BEV binária que contém apenas informações das sinalizações das faixas) obtido como

resultado da etapa de extração de características. O histograma de coluna extrai o vetor de intensidades, detectando assim os dois principais picos de intensidade nesta imagem. Na Figura 2.15, estes dois picos são encontrados (em negrito), respectivamente, na primeira e quarta posição do vetor de intensidades ($i = 0$ e $i = 3$).

Estes picos indicam as duas regiões de maior probabilidade de se encontrar as faixas, uma vez que na etapa de pré-processamento, é realizada a mitigação de qualquer informação que não seja pertencente a faixa de trânsito. Estas duas regiões são utilizadas para iniciar os sensores virtuais.



(a) Exemplo de mapa de característica



(b) Histograma de coluna inicial da Figura 2.16(a).

Figura 2.16: Exemplo da operação do histograma de coluna em um mapa de características.

Seguindo este princípio, a Figura 2.16(b) apresenta o histograma de coluna inicial da Figura 2.16(a), em verde é apresentado as intensidades de cada coluna, ao passo que as posições horizontais dos dois picos são apresentados pelos círculos azuis.

Sensores Virtuais

Os sensores virtuais são retângulos de dimensão fixa ou variável para cada cenário de operação. Estes sensores são utilizados para realizar a estimação de pontos na imagem que

pertencem as sinalizações das faixas de trânsito. São utilizados em conjunto com o método de histograma de coluna. Inicialmente, os dois primeiros sensores são posicionados através do histograma de colunas inicial realizado sobre a imagem (MUTHALAGU; BOLIMERA; KALAICHELVI, 2020). Em seguida, cada uma das iterações na etapa de reposicionamento acarreta uma nova operação de histograma de coluna sobre a área de um sensor virtual, esta operação é melhor visualizada na Figura 2.17.

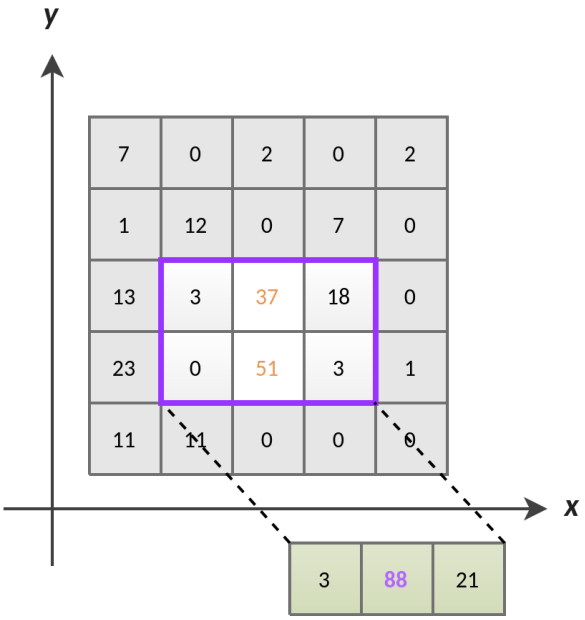


Figura 2.17: Exemplo da operação de histograma de coluna em um sensor virtual.

O diagrama da Figura 2.17 apresenta um exemplo da operação de histograma de coluna realizada sobre um sensor virtual. A matriz de elementos cinzas representa uma porção da imagem, ao passo que o retângulo em lilás representa um sensor virtual 2×3. De forma similar à Figura 2.15, o vetor verde representa as intensidades de cada uma das colunas do sensor virtual e em lilás é apresentado o valor de pico deste sensor.

Reposicionamento

Uma vez que os dois primeiros sensores foram posicionados sobre a imagem, estes devem iterar por toda área da imagem para encontrar as posições da maior probabilidade de conter as sinalizações das faixas de trânsito. Uma forma, seria varrer a imagem inteira para obter o resultado. Contudo, esta tarefa demanda alto custo computacional e itera sobre regiões onde não há presença das sinalizações. Assim, uma alternativa é iterar os sensores de forma vertical, da borda inferior até a superior da imagem. Percorrendo apenas as regiões de maior

probabilidade, determinada pelos dois picos do vetor de intensidades obtidos no histograma de colunas inicial (REICHENBACH et al., 2018; CAO et al., 2019a; MUTHALAGU; BOLIMERA; KALAICHELVI, 2020).

Porém, esta abordagem falha em cenários onde há curvatura das faixas de trânsito, pois estas não estão totalmente verticais. Desta forma, é necessário o reposicionamento dos sensores, para que os mesmos fiquem centralizados sobre as faixas. Esta tarefa, é realizada reposicionando horizontalmente cada um dos sensores sobre os pontos de maior densidade do sensor anterior e deslocando-os verticalmente ao longo da imagem (REICHENBACH et al., 2018; CAO et al., 2019a; MUTHALAGU; BOLIMERA; KALAICHELVI, 2020). A operação de reposicionamento e consequentemente o método de janelas deslizantes podem ser melhor visualizados nas imagens apresentadas na Figura 2.18.

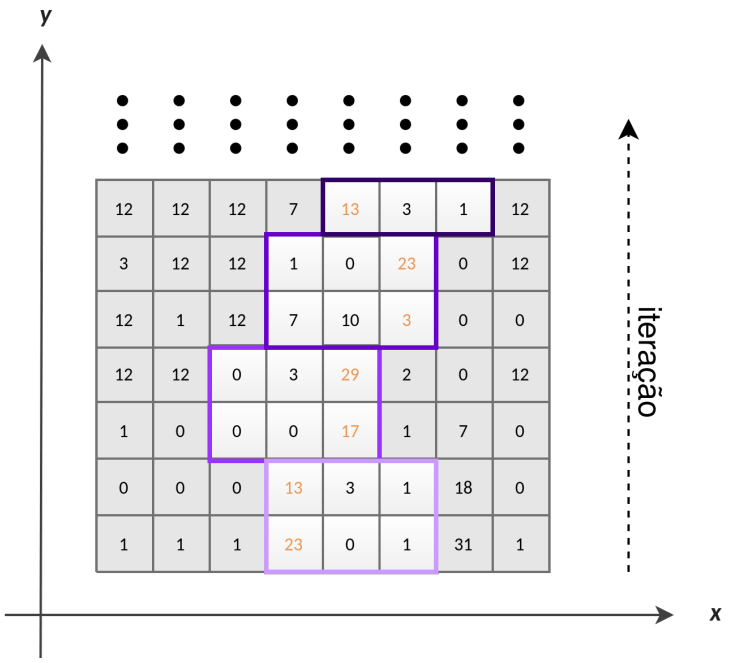


Figura 2.18: Exemplo da operação de reposicionamento dos sensores no método de janelas deslizantes.

Na Figura 2.18 são apresentadas as iterações do método de janelas deslizantes. Inicialmente é estabelecido o sensor virtual na borda inferior da imagem, representado pelo retângulo lilás claro ao passo que em laranja são ressaltados os valores da coluna de maior intensidade. Realizando a operação de histograma de coluna neste sensor, é obtido seu vetor de intensidades e o pico ocorre na primeira posição. Em seguida, o sensor virtual é iterado, sendo deslocado verticalmente para cima e horizontalmente posicionado sobre o ponto de maior densidade do sensor anterior e assim sucessivamente. Desta forma, os sensores virtuais buscam sempre se

manter alinhados a posição de maior intensidade, que no contexto são as sinalizações de faixas de trânsito.

2.3.2 Ajuste da Faixas de Trânsito

Diversos modelos para o ajuste de faixas de trânsito têm sido explorados na literatura. Em Kuk et al. (2010), foi utilizado um modelo puramente linear. A maioria dos modelos lineares são baseados na transformada de Hough (BAR HILLEL et al., 2014b). As técnicas com base em modelos lineares têm a vantagem de geralmente serem menos complexos e exploram a suposição de que a faixa próxima ao carro é aproximadamente linear, mesmo quando há curvatura. Como a transformação de Hough às vezes é vista como um algoritmo de alto custo computacional (SATZODA; SATHYANARAYANA; SRIKANTHAN, 2009) outros modelos estão sendo adotados. Além disso, outra desvantagem dos modelos lineares é o fato de que a complexidade das pistas no mundo real não pode ser simplificada ao ponto de sua adoção. Embora modelar uma faixa de trânsito com uma linha única seja muito proibitiva para o ajuste em cenas complexas, uma combinação de segmentos de linha, como em Lipski et al. (2008), foi usada com sucesso. Isso pode ser feito porque as sinalizações de faixas são consideradas localmente lineares, pelo menos aproximadamente.

Em Yu, Han e Hahn (2009), uma função quadrática é usada para modelar a pista. Os modelos parabólicos são mais flexíveis que lineares e podem ser tão rápidos quanto. No entanto, esses modelos ainda são restritivos em cenas complexas. Muitos outros modelos foram projetados com base nessas limitações. Um modelo de forma circular foi proposto por Ma, Lakshmanan e Hero (2000), partindo do pressuposto de que os limites das pistas são dispostos em círculos concêntricos, pelo menos sobre pequenos segmentos. Embora os autores afirmem que esses modelos são melhores do que os modelos parabólicos, eles não realizaram experimentação para apoiar essa afirmação. A mesma suposição é usada por modelos baseados em arcos circulares, como em Kreucher, Lakshmanan e Kluge (1998) e Cheng et al. (2006).

Ao final das iterações do método de janelas deslizantes é obtido um conjunto de pontos para cada candidato à faixa de trânsito. Isto é, cada um dos centros dos sensores representa um ponto (coordenadas x e y) deste conjunto. Assim, cada conjunto pode ser ajustado através de um polinômio ou de forma não paramétrica e encontrar uma curva que se ajusta a este conjunto de pontos, estabelecendo assim os candidatos às faixas de trânsito de uma dada imagem (SON; LEE; KUM, 2018b).

Para encontrar as faixas são utilizadas as coordenadas, horizontais e verticais, de cada uma das posições de maior probabilidade fazendo uma regressão para um polinômio de segunda ordem, uma vez que as curvaturas das vias são quase constantes e o polinômio quadrático é suficientemente preciso para o ajuste (SON; LEE; KUM, 2018b; SIVARAMAN; TRIVEDI, 2013; PROCHAZKA, 2013). O modelo quadrático pode calcular o deslocamento lateral, o ângulo de direção do veículo e a curvatura da estrada. A fórmula do modelo é apresentada na Equação 2.20

$$y = a_0 x^2 + a_1 x + a_2, \tag{2.20}$$

onde x e y são as coordenadas obtidas pelo algoritmo de janelas deslizantes. Estes são os parâmetros do modelo quadrático, que são usados para calcular o desvio lateral Δy, a inclinação da pista μ e a curvatura da estrada κ, como mostrado na sequência da Equação 2.21.

$$\Delta y = -a_2 \tag{2.21}$$

$$\mu = -\arctan(a_1) \tag{2.22}$$

$$\kappa = 2a_0/(a_1^2 + 1)^{\frac{3}{2}}. \tag{2.23}$$

2.4 Conclusões do Capítulo

As Seções 2.1, 2.2 e 2.3 apresentam os fundamentos matemáticos e os algoritmos relacionados ao método proposto, ao passo que faz uma revisão teórica dos principais trabalhos que formam a base deste projeto. Além disso, alguns métodos de processamento de imagem, como filtros básicos para remoção de ruídos e correção de imagem, filtros de limiar adaptativo e os extratores de característica são introduzidos de forma teórica. São expostos os conceitos básicos que envolvem o processo de estimação e detecção das faixas, partindo dos métodos de extração e geração de mapas de características combinados até os algoritmos de estimação de faixas via janelas deslizantes e ajuste polinomial dos candidatos. Embora diversos métodos de pré-processamento e extração de características tenham sido apresentado no presente capítulo, apenas alguns desses foram discutidos durante os experimentos, sendo descartados nas implementações finais.

Capítulo 3

Computação Heterogênea

A presente seção tem como objetivo fornecer uma visão direta das principais características das GPUs, destacando os elementos básicos, as diferenças dos processadores convencionais e os ganhos em relação ao desenvolvimento sequencial. É apresentado também, os fundamentos da arquitetura heterogênea, definindo formalmente o esquema adotado para a elaboração do projeto e também o modelo de programação para a implementação do sistema. Por fim, são apresentadas as principais otimizações propostas para construção do projeto utilizando de forma eficiente todos os recursos disponíveis pelo hardware.

Esta seção é fortemente baseada nas literaturas Kirk e Hwu (2016) e NVIDIA (2021a, 2020, 2014) e tem como principal objetivo apresentar as características de um sistema heterogêneo, em especial, apresentar as características de um chip gráfico e sua forma de programação e otimização.

3.1 Características de GPUs

A principal motivação para o desenvolvimento dos chips gráficos foi a aplicação como um processador específico e otimizado para acelerar a computação gráfica, ou seja, atender as demandas e cargas computacionais necessárias para processamento de tempo real como renderizadores e gráficos tri-dimensionais. As principais aplicações deste tipo de hardware se dão no desenvolvimento de jogos, indústria cinematográfica, efeitos visuais e problemas de computação paralela. Em especial, a aplicação deste tipo de hardware é vastamente utilizada para solução de processamentos de imagem e visão computacional, como o discutido neste trabalho, além de ser o padrão adotado para treinamento e simulações de redes neurais.

As GPUs apresentam uma capacidade de vazão de instruções e banda de memória superior a uma CPU com um custo e potência similar. A diferença nas capacidades entre a GPU e a CPU existem devido a forma como foram projetadas e os objetivos de cada um dos dispositivos. Enquanto a CPU é desenvolvida para executar operações sequenciais (uma thread) o mais rápido possível e podendo executar apenas algumas *threads* em paralelo, a GPU é desenvolvida para executar uma quantidade enorme de *threads* em paralelo, alcançando uma vazão de dados maior.

A GPU é especializada para cálculos altamente paralelos e, portanto, projetada de forma que mais transistores sejam dedicados ao processamento de dados em vez de armazenamento em cache de dados e controle de fluxo. Um exemplo do projeto de arquitetura de uma CPU e uma GPU é apresentado na Figura 3.2.

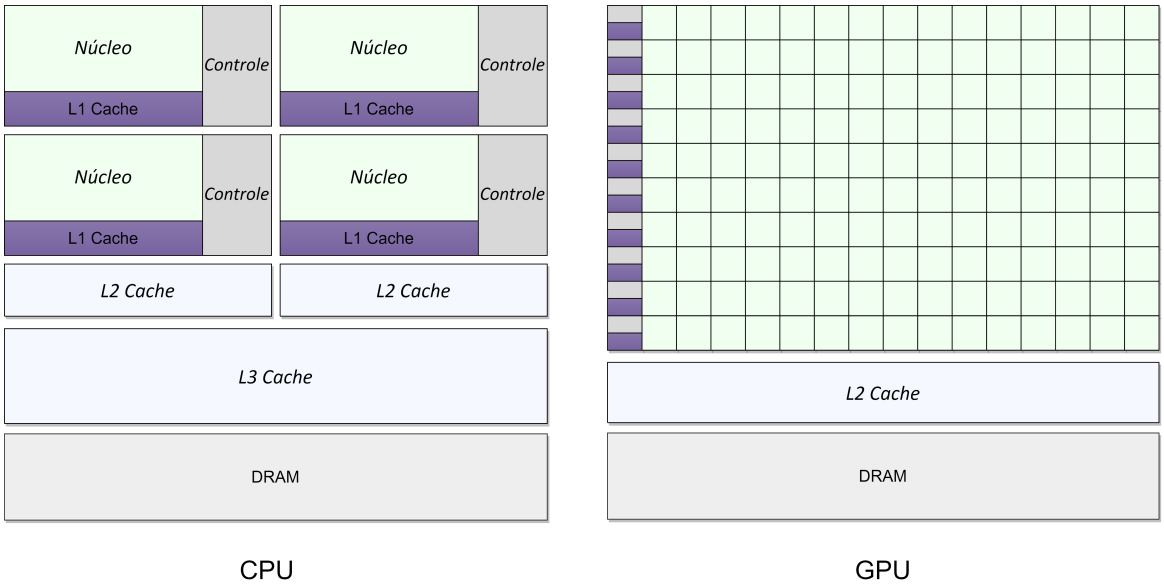


Figura 3.1: Diagrama ilustrativo dos elementos básicos da arquitetura de uma CPU e de uma GPU.

É notável que a GPU aloca mais transistores para o processamento de dados, significando, por exemplo, mais poder para cálculos de ponto flutuante e processamento paralelizado. Além disso, as GPUs buscam reduzir as latências de acesso à memória através da computação de quantidades massivas de dados ao invés de depender de grandes caches de dados e controles de fluxo complexos, ambas funcionalidades com custo elevado em termos de transistores.

Em geral, aplicações que buscam uma solução otimizada utilizando tanto a CPU quanto a GPU são chamadas de implementações heterogêneas, ou seja, há a combinação de partes de execução paralelas e sequenciais. Esta categoria de sistemas é projetada com uma combinação de GPUs e CPUs para maximizar o desempenho geral. Os aplicativos com alto

grau de paralelismo podem explorar essa natureza massivamente paralela da GPU para obter um desempenho mais alto do que na CPU.

3.1.1 Arquitetura Heterogênea

Como mencionado, um sistema heterogêneo possui parte da execução desempenhada em forma sequencial pela CPU e parte executada de forma paralela em uma GPU. A Figura 3.2 apresenta um exemplo de arquitetura convencional para este tipo de solução.

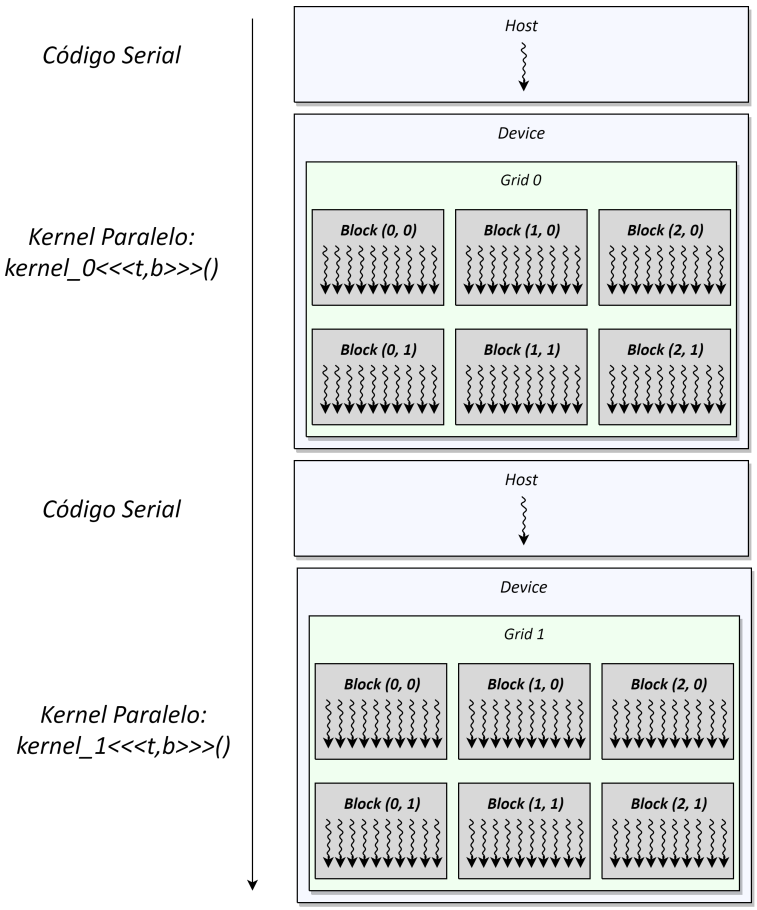


Figura 3.2: Diagrama ilustrativo da arquitetura básica de uma CPU e uma GPU.

A Figura 3.2 apresenta quatro seções de execução, duas seriais e duas paralelas. As etapas seriais são denotadas pela representação de execução em *Host*, nomenclatura adotada pela NVIDIA para descrever uma execução na CPU, e que consequentemente não é realizada no *Device*, que representa o chip gráfico.

Outro ponto descrito no diagrama da Figura 3.2 é a relação e organização das *threads* do sistema. Enquanto na parte serial há poucas *threads* em execução, representado pela seta ondulada, na paralela há a presença de um conjunto massivo dessas em execução. Não obstante,

é possível observar que há um arranjo estrutural de *threads*, sendo separadas em blocos que pertencem a uma determinada grade.

Por fim, é apresentado na linha de execução proposta, uma notação adotada para sinalizar uma função que roda em um conjunto de *threads* simultâneas da GPU, este artefato é chamado de *kernel*. Ele é constituído pelo nome do *kernel*, a quantidade de *threads* `t` e blocos `b` lançados para a execução (argumentos delimitados pelos símbolos ⋘ e ⋙) e os argumentos do método do *kernel*, como comumente utilizado em linguagem C/C++.

Este conjunto de características para a programação heterogênea descritas nesta seção e abordados ao longo do trabalho, dizem respeito a interface de programação CUDA, fornecida pela NVIDIA para o desenvolvimento de códigos heterogêneos em chips gráficos.

3.2 Modelo de Programação

Esta seção apresenta os conceitos principais da programação heterogênea utilizando CUDA. São discutidos os conceitos básicos do modelo de programação e seus elementos. Os principais pontos discutidos são o método de execução de uma função, alguns tipos de instruções, hierarquia de memórias e principalmente as otimizações permitidas neste tipo de sistema.

A interface de programação CUDA C++ estende a linguagem de programação C++ para possibilitar o desenvolvimento de funções que podem ser executadas nas GPUs, os chamados *kernels*. Além disso, é fornecida uma série de primitivas de programação otimizadas, como as operações SIMT (*Single Instruction Multiple Thread*) e SIMD (*Single Instruction Multiple Data*). Este grupo de operações permitem performance elevadas do ponto de vista de eficiência de instruções, pois em geral, performam uma mesma operação com um conjunto reduzido de instruções. Ademais, o modelo de programação é constituído de diversas outras características e algumas das principais são descritas nas seções seguintes.

A Figura 3.3 apresenta de forma resumida o fluxo de operação no modelo de programação heterogênea.

A primeira etapa deste modelo de programação constitui-se em elaborar os dois códigos que compõe um sistema heterogêneo: o código serial executado na CPU e o código paralelo da GPU. A segunda etapa é relacionada com a transferência dos dados que serão utilizados nos cálculos paralelos. Esta transferência, é em geral, realizada copiando dados da memória do *Host* para o *Device*.

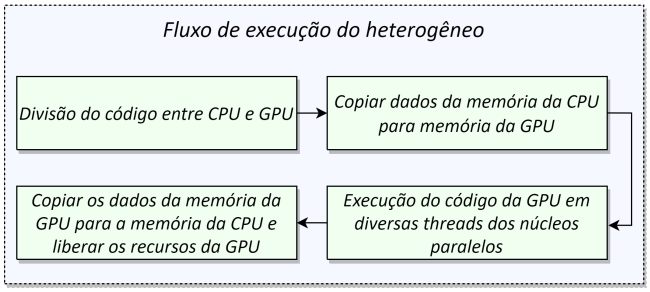


Figura 3.3: Diagrama simplificado do fluxo de execução de um sistema heterogêneo.

Após a cópia dos dados e a invocação do *kernel*, é feita a execução de forma paralela das operações programadas no código da GPU. Então, os resultados obtidos nesta etapa são transferidos de volta para a memória da CPU, possibilitando o uso destes dados pelas outras partes do sistema e também possibilitando novas interações entre CPU e GPU.

3.2.1 Kernels

O elemento básico da computação paralela, como já mencionado, é o *kernel*. Um *kernel* quando executado em um dispositivo, realiza a operação de forma paralela no hardware. Isto é, são executados N vezes em paralelo em N diferentes *threads*/núcleos CUDA do chip gráfico, diferentemente das funções em C++ convencionais que são, em geral, *monothread* e executada de forma sequencial na CPU.

Como mencionado o CUDA C++ possibilita a criação de *kernels*, que são funções em C++ que executam em paralelo nos núcleos da GPU. A declaração de um *kernel* é similar a declaração de funções regulares da linguagem C++, como mencionado anteriormente.

De forma simplificada, um *kernel* é definido pela palavra reservada `__global__` que especifica a declaração do *kernel*. Diferentemente dos métodos tradicionais do C++, é especificado o arranjo de *threads* CUDA que são lançadas na execução do *kernel*. Este argumento é definido pela quantidade de *threads* de cada bloco e a quantidade total de blocos. O arranjo é declarado utilizando ⋘blocos, *threads*⋙ que é uma extensão da sintaxe de execução.

Cada uma das *threads* em execução é identificada por um índice único chamado de `threadIdx`. Similarmente, cada bloco é identificado pelos índices `blockIdx`. Essas palavras reservadas permitem que em tempo de execução cada uma das *threads* e blocos sejam identificados, tornando possível executar operações distintas em blocos e *threads* distintas, como por exemplo, divergência de fluxo.

O Código 3.1 apresenta um exemplo da declaração e utilização de um *kernel* em CUDA.

```
1/* Declaracao do kernel de soma de vetores. */
2__global__ void soma_vetores_kernel(
3  int * vetor_a,
4  int * vetor_b,
5  int * vetor_c)
6{
7  int idx = threadIdx.x;
8  vetor_c[idx] = vetor_a[idx] + vetor_b[idx];
9}
10
11int main()
12{
13  ...
14  /* Invocacao do kernel com N threads. */
15  soma_vetores_kernel<<<1,N>>>(vetor_a, vetor_b, vetor_c);
16  ...
17}
```

Código 3.1: Exemplo de declaração de um kernel em CUDA.

É declarado um *kernel* com o nome `soma_vetores_kernel` que realiza o cálculo da soma de dois vetores, `vetor_a` e `vetor_b`, e registra em um terceiro vetor `vetor_c`.

O *kernel* em questão é executado de forma paralela em N *threads* CUDA que pode ser constatado pelo argumento no arranjo de *threads* de lançamento, que possui um único bloco de N *threads*. A execução em paralelo desta soma é garantida pelo uso da variável de controle `idx` que utiliza os índices únicos das *threads* para garantir que cada *threads* execute a soma dos mesmos índices. Ou seja, neste formato, cada uma das N *threads* executa a soma de um par (os elementos `idx` de `vetor_a` e `vetor_b`) de forma paralela.

Para elucidar o funcionamento do *kernel* e a relação com as *threads* lançadas, é possível supor que cada um dos vetores tivessem 1024 elementos. Em uma execução serial, seriam necessários 1024 iterações para obter a soma. Contudo, utilizando $N = 1024$ é possível obter, em uma mesma execução o cálculo dos 1024 elementos.

As *threads* e blocos são os elementos que viabilizam a paralelização em massa da execução dos *kernels* na GPU. Entender a hierarquia e as estruturas desses elementos é importante para a compreender as formas de se otimizar o uso dos recursos do hardware.

3.2.2 Hierarquia de *threads*

Como mencionado, uma thread é identificada através de um índice único, o `threadIdx`. Este índice é composto por um vetor de três componentes (x, y e z), sendo possível formar blocos de *threads* em uma, duas ou três dimensões. Cada uma dessas dimensões pode ser acessada através dos índices `threadIdx.x`, `threadIdx.y` e `threadIdx.z`. Assim, é possível lançar blo-

cos de *threads* para processar elementos nos três domínios espaciais, vetor, matriz ou volume, conforme ilustrado na Figura 3.4.

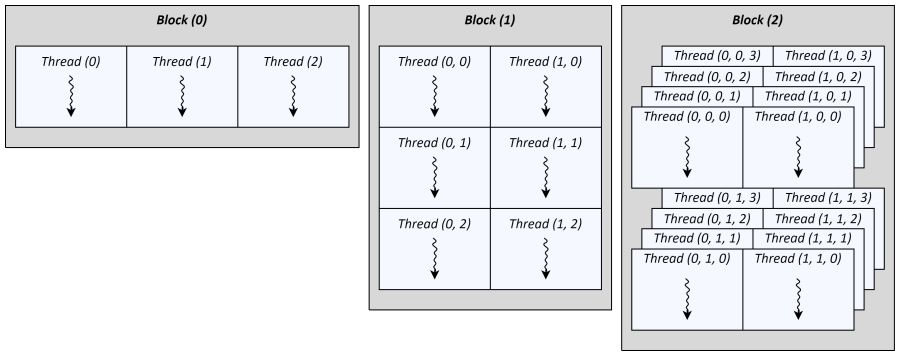


Figura 3.4: Diagrama ilustrativo para exemplificar as três possível dimensões de um bloco de *threads*.

A Figura 3.4 apresenta três blocos distintos para exemplificar os três arranjos possíveis. O primeiro bloco apresenta um arranjo unidimensional, equivalente a um lançamento do tipo ⋘1, 3⋙, ou seja, uma grade com um único bloco de três *threads*. Para acessar cada uma das *threads* o índice único de identificação seria `threadIdx.x = 0, 1, 2`.

No bloco do meio, é ilustrado um bloco bidimensional onde a indexação das *threads* é feita através do elementos `threadIdx.x` e `threadIdx.y`, conforme apresentado nas figuras das *threads*. Por fim, o terceiro bloco apresenta um arranjo tridimensional, onde há o acréscimo do índice `threadIdx.z`, responsável por garantir um acesso "volumétrico".

Para obter o índice global da thread é utilizado um arranjo linear, ou seja, para um bloco unidimensional é utilizado o próprio índice. Para um bloco bidimensional de dimensões (D_x, D_y), a thread de índices (x, y) é $(x+y\times D_x)$. De forma similar para um bloco tridimensional de dimensões (D_x, D_y, D_z) o identificador da thread de índice (x, y, z) é $(x + y \times D_x + z \times D_y)$.

Para elucidar o acesso dos arranjos de blocos multi-dimensionais possíveis, o Código 3.2 apresenta a implementação de um bloco bidimensional de $N \times N$ *threads* que executa a soma de duas matrizes de ordem N.

```
1  /* Declaracao do kernel de soma de matrizes. */
2  __global__ void soma_matrizes_kernel(
3  int matriz_a[N][N],
4  int matriz_b[N][N],
5  int matriz_c[N][N])
6  {
7    int i = threadIdx.x;
8    int j = threadIdx.y;
9    matriz_c[i][j] = matriz_a[i][j] + matriz_b[i][j];
10 }
11
12 int main()
13 {
14   ...
15   /* Invocacao do kernel com um bloco de N*N*1 threads. */
16   int n_blocos = 1;
17   dim3 threads_por_bloco(N, N);
18   soma_matriz_kernel<<<n_blocos, threads_por_bloco>>>(matriz_a, matriz_b, matriz_c);
19   ...
20 }
```

Código 3.2: Exemplo de declaração de um kernel com bloco bidimensional em CUDA.

O Código 3.2 é similar ao apresentado anteriormente no Código 3.1, contudo, é possível notar a mudança no acesso através dos índices `threadIdx.x` e `threadIdx.y` decorrentes da escolha bidimensional de lançamento dos blocos.

As threads são recursos limitados dentro de uma GPU, não obstante, existe também um limite de *threads* por blocos. Isto porque cada thread possui seus próprios recursos e também recursos compartilhados com os núcleos do chip gráfico, como memórias e registradores. Assim, existe uma quantidade finita e limitada de *threads* em um mesmo bloco.

Entretanto, cada *kernel* pode ser executado por um arranjo de múltiplos blocos de mesmo formato, resultando em uma quantidade de *threads* total, igual ao número de blocos multiplicado pela quantidade de *threads* em cada um dos blocos, como apresentado anteriormente na indexação das *threads*.

Assim como as *threads* de um bloco, os blocos de uma grade também são organizados em composições de até três dimensões. Portanto, é possível se ter uma grade com blocos unidimensionais, bidimensionais e tridimensionais. A Figura 3.5 exemplifica a estrutura de uma grade.

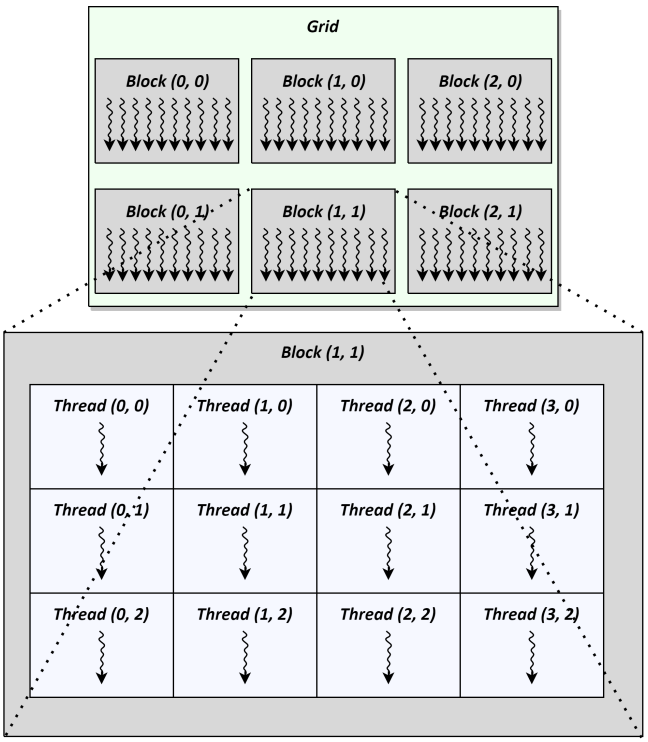


Figura 3.5: Diagrama ilustrativo dos diferentes elementos da hierarquia de *threads* de uma GPU.

A grade apresentada possui duas dimensões e seis blocos de *threads* e cada bloco é bidimensional possuindo doze *threads*. Assim, um *kernel* lançado com esta grade possui um total de $1 \times 6 \times 12$ *threads*.

De forma intuitiva, a quantidade de *threads* lançadas em um determinado *kernel* depende do tamanho dos dados que serão processados e também de seu arranjo. Em geral, são lançadas mais *threads* do que o número de núcleos disponíveis no sistema.

Similarmente a indexação das *threads*, um bloco de uma grade pode ser identificado por um índice único unidimensional, bidimensional ou tridimensional através das variáveis de sistema `blockIdx`, exatamente como é feito com as *threads*. Além disso, é possível também acessar a dimensão, em cada um dos eixos de um bloco, através da variável `blockDim`.

Para exemplificar o lançamento e acesso de blocos multidimensionais, é apresentado o Código 3.3, que re-implementa a soma de matrizes utilizando mais de um bloco.

```
1  /* Declaracao do kernel de soma de matrizes. */
2  __global__ void soma_matrizes_kernel(
3  int matriz_a[N][N],
4  int matriz_b[N][N],
5  int matriz_c[N][N])
6  {
7    int i = blockIdx * blockDim.x + threadIdx.x;
8    int j = blockIdy * blockDim.y + threadIdx.x;
9    if (i < N && j < N)
10     matriz_c[i][j] = matriz_a[i][j] + matriz_b[i][j];
11 }
12
13 int main()
14 {
15   ...
16   /* Invocacao do kernel com um bloco de N*N*1 threads. */
17   dim3 threads(16, 16);
18   dim3 blocos(N/thread.x, N/thread.y);
19   soma_matriz_kernel<<<blocos, threads>>>(matriz_a, matriz_b, matriz_c);
20   ...
21 }
```

Código 3.3: Exemplo de declaração de um kernel com um grid de blocos bidimensionais em CUDA.

O Código 3.3 apresenta o lançamento de blocos de 256 *threads* em um arranjo bidimensional de 16×16 *threads*. A quantidade de blocos é definida de tal forma a cobrir todos os elementos da matriz.

Quando o *kernel* é lançado, não existe uma ordem correta ou determinística para a execução dos blocos, cada um é executado de forma independe. É possível executar em qualquer ordem, em paralelo ou mesmo em série. Esta independência na execução, permite que os blocos de *threads* sejam escalonados em qualquer ordem, permitindo escalar facilmente o código com o número de núcleos disponíveis no sistema.

Como mencionado anteriormente, as *threads* de um bloco compartilham diversos recursos e podem cooperar entre si, transferindo informações através da SM (*Shared Memory*) e das primitivas de sincronização, que permitem acesso coordenado às regiões de memória.

Um exemplo desta cooperação entre *threads* é a utilização de primitivas de sincronização como a função `__syncthreads()` fornecida pela API CUDA. Ela funciona como um ponto de sincronização entre as *threads*, atuando como uma barreira que retém todas as *threads* de um bloco, que só serão liberadas quando todas estiverem na mesma instrução. Esta função utiliza poucos recursos do sistema e não gera sobrecarga. Outro recurso vastamente utilizado nas otimizações, é a SM, que permite o acesso de dados desalinhados entre as *threads* com baixa latência de acesso, tal como uma memória cache L1 de um processador.

3.2.3 Hierarquia de Memórias

Existem alguns tipos de memórias disponíveis para as *threads* em CUDA, em especial, cada thread tem uma memória privada local. Cada bloco de thread possui uma SM que é visível e acessível por todas as *threads*, que compartilham o mesmo bloco e também possuí o mesmo ciclo de vida do bloco. Além disso, existe uma memória global, que todas as *threads* do sistema podem acessar, sendo visível por todos os blocos de *threads*.

Para ilustrar esta distribuição hierárquica das memórias e a sua relação com as *threads* e os blocos, foi elaborado o diagrama da Figura 3.6, que apresenta a relação entre esses elementos.

A SM, definida pela palavra reservada `__shared__`, é um dos recursos que viabiliza grandes otimizações, em especial, fornece uma forma de acesso e transferência de dados entre as *threads* do bloco com rapidez. É uma memória de baixa latência, que permite acesso desordenado com eficiência, ao contrário da memória global. A SM fornece um gerenciamento de cache via software, o que minimiza os acessos à memória global pelas *threads* do bloco. Os usos da SM para a otimização da computação dos *kernels* são abordados nas seções seguintes.

3.3 Otimizações em CUDA

Visando uma execução em tempo real no sistema embarcado, é importante garantir que os algoritmos executem da melhor forma possível, utilizando todos os recursos do hardware,

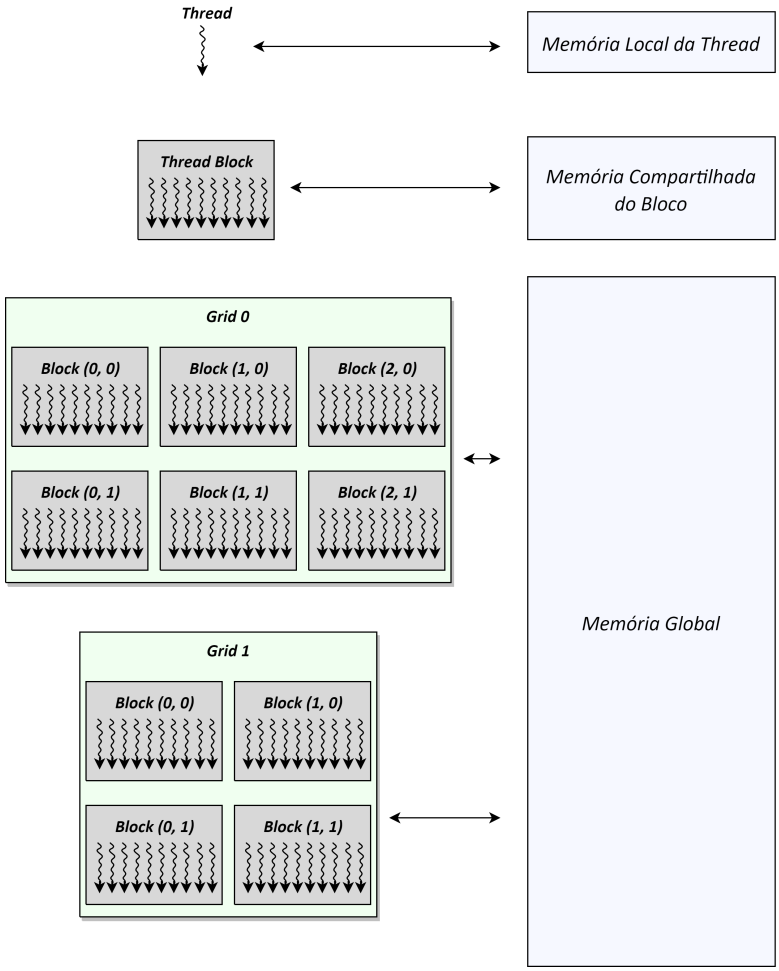


Figura 3.6: Diagrama ilustrativo dos diferentes elementos da hierarquia de memórias de uma GPU.

uma vez que estes são limitados. Neste sentido, é possível fazer diversas otimizações em CUDA e garantir que os algoritmos tenham alta eficiência de execução.

Diversas otimizações podem ser feitas para atingir maior eficiência na execução, e estas foram aplicadas em grande parte das implementações propostas. Desta forma, esta seção sumariza as principais otimizações utilizadas nas elaborações dos *kernels* que compõe o método proposto. Otimizações específicas de cada um dos métodos são detalhados na Seção 4.4. Onde cada um dos algoritmos e suas implementações são apresentados de forma mais profunda.

O principal objetivo quando se trata de otimizações, é encontrar quais são os fatores limitantes dos algoritmos, que podem ser de divididos em basicamente três categorias:

- limitações na banda de memória;
- limitações relacionadas a latência;
- limitações na taxa de transferência/rendimento das instruções.

O fluxo padrão para obter um sistema otimizado, é baseado em produzir uma versão inicial do *kernel* e então através de análises, identificar quais são os limitantes dessa execução. A partir disso, é possível propor novas otimizações e este ciclo de implementação, análise e otimização é mantido até obter uma versão onde a limitação seja o recurso do hardware e não o software desenvolvido.

Para identificar essas limitações pode se utilizar ferramentas que auxiliam na medição dos parâmetros doS códigos, como ferramentas de *profiling*. Durante a elaboração do trabalho, foi utilizado o VisualProfiler (NVIDIA, 2021b) fornecido pela NVIDIA para a análise de execuções CUDA. Este programa é capaz de analisar a execução das implementações em CUDA e fornecer uma série de métricas, tais como, o tempo de execução, o número de registradores utilizados, a quantidade de memória, a relação dos tipos memórias, e a qualidade dos acessos, a demanda de unidades aritméticas, etc. Além disso, é apresentado também a interação entre os diversos *kernels*, mostrando a latência de lançamento entre eles, e como eles estão organizados nos diferentes fluxos de execução.

O conteúdo abordado nesta seção te m como principais referências os trabalhos Kirk e Hwu (2016) e NVIDIA (2021a, 2020, 2014).

3.3.1 Otimizações de Memória

Como mencionado anteriormente, existem alguns tipos de memórias nos dispositivos embarcados e cada uma delas com suas próprias especificações. As características mais importantes do ponto de vista de execução, são as capacidades de armazenamento e a velocidade de acesso. A primeira, diz respeito a quantidade de informação que é possível alocar neste dispositivo simultaneamente, ao passo que a velocidade de acesso, diz respeito ao tempo de leitura e gravação de uma informação na memória.

Usualmente, nos sistemas heterogêneos são utilizados dois tipos principais de memórias, a memória global, geralmente mais lenta e com maior capacidade de armazenamento, é uma memória interna menor, com taxas de transferências mais elevadas. Além disso, uma característica inerente deste tipo de recurso é o tipo de acesso, em geral, existem arranjos organizados e estruturados, que conseguem garantir maior eficiência na transferência de informação.

O diagrama da Figura 3.7 apresenta a relação entre os diferentes tipos de memória e a arquitetura heterogênea. O diagrama representa uma visão simplificada da execução de um *kernel*, onde uma grade de dois blocos é lançada.

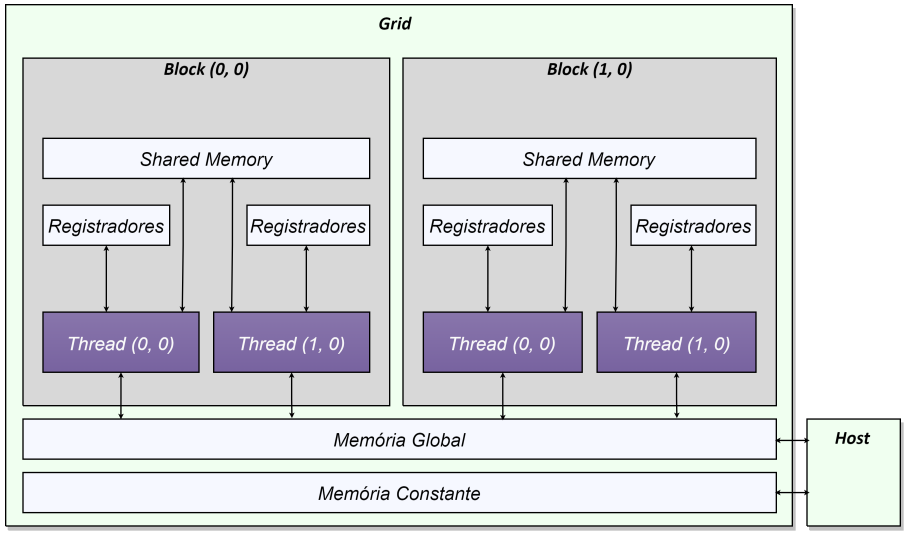


Figura 3.7: Diagrama ilustrativo da interação dos diferentes tipos de memórias do sistema heterogêneo.

Nota-se que a troca de informação entre os dados do *Host* e o chip gráfico, são feitos via memória global. Esta é a principal memória do sistema, ela serve como ponte entre as informações da GPU e da CPU, sendo responsável por inserir e coletar os resultados dos dados processados.

A memória global possui a característica de ser visível por todas as *threads* do sistema, representado pelas setas bidirecionais entre as entidades de *threads* e a memória. Além disso, por ser a memória principal do sistema, possui alta capacidade de armazenamento, na ordem de GB, e consequentemente um tipo de memória menos complexa, quando comparada com memórias do tipo cache.

Outra característica inerente deste tipo de memória é que, como possui um armazenamento elevado, e faz a ponte entre os dois sistemas, é uma memória mais lenta. Isso se deve a dois fatores, a posição física do chip, geralmente externa à GPU e o tamanho do armazenamento, implicando em uma memória mais simples. A taxa de transferência desse tipo de recurso é mais lento e seu acesso demanda cerca de 400 à 600 ciclos de *clock*.

A SM é o oposto da memória global, ela possui baixa latência e velocidade elevada, contudo, possui um tamanho significativamente menor, na ordem de kB. Estes atributos são inerentes, tanto da posição física, interna à GPU, quanto ao tipo da memória. Ela esta naturalmente associada a um tipo de memória cache interna de cada *warp*, e o seu tamanho é limitado por processador. Não sendo é acessível pelas outras partes do sistema, apenas pelas *threads* do *kernel* em execução, sendo acessível as *threads* de um mesmo bloco. A velocidade

desta memória é uma centena de vezes maior que a global, demandando na ordem de quatro ciclos de *clock*.

Os registradores são outra forma de armazenar informações internas as *threads*. Na realidade, seu uso é restrito as *threads*, pois o ciclo de vida dos registradores é alinhado com o ciclo de vida da thread. Não obstante, cada *thread* possui acesso somente aos seus registradores, não sendo possível compartilhar informações por este tipo de memória, ao contrário da SM, que viabiliza essas transferências internas em um bloco. Contudo, assim como a SM, os registradores são um tipo rápido de armazenamento, sendo internos ao chip.

Assim, os principais objetivos deste tipo de otimização, são de garantir uma maior eficiência no acesso à memória, elevando as taxas de transferência e o rendimento deste acesso. A abordagem mais comum é minimizar a quantidade de acessos ás memórias globais, visto que são inerentemente mais lentas. Além disso, sempre que os acessos são necessário, devem ser realizados esforços para reduzir a quantidade de dados transferidos, carregando apenas os dados necessários e sempre com um padrão de acesso adequado, garantindo eficiência máxima transferências.

Para atingir altos rendimentos, uma técnica emprega na implementação dos *kernels*, é o uso da SM para acessos redundantes e ou sem *coalescing*. Isto significa, que sempre que os dados são requisitados múltiplas vezes no algoritmo ou quando o acesso é desestruturado e não possui alinhamento, é recomendado o uso da SM como uma memória cache, facilitando e otimizados estas situações. Esse tipo de abordagem garante uma melhor taxa de transferência de dados, pois a SM funciona como uma memória cache de baixa latência, demandando poucos ciclos de *clock* e alcançando rendimentos elevados no acesso aos dados.

Um exemplo concreto da utilização deste recurso, pode ser observado na otimização de um núcleo de convolução. Esta operação, consiste na multiplicação de uma matriz por um conjunto de píxeis. Os dados utilizados nesta operação são limitados e são reutilizados diversas vezes, devido a operação de multiplicação. Além disso, a forma que esses dados são consumidos durante a execução, não possibilita um arranjo otimizado para o acesso. Portanto, o uso desta abordagem com SM pode propiciar ganhos de desempenho para esta operação, do ponto de vista de acesso à memória.

Como mencionado, sempre que é necessário a leitura de dados da memória global, é preferível que se tenha um padrão de acesso alinhado, garantindo uma leitura com *coalescing* perfeito dos dados. Neste sentido, algumas operações, como a transposição das estruturas de

dados de *Array of Struct (AoS)* para *Struct of Array (SoA)* se faz necessário, bem como a aplicação de *padding* e mudanças no esquema de paralelização, que evitam carregamento de dados desnecessários.

Um exemplo prático da utilização deste tipo de recurso é um *kernel* de conversão de espaço de cores, tal como, a conversão de uma imagem RGB para escala de cinza discutida na Seção **??**. Neste caso, os mesmos comentários sobre a SM são válidos, porém é necessário realizar uma transposição dos dados entre leitura e processamento das informações.

3.3.2 Otimizações de Latência

O principal problema relacionado a latência para a execução eficiente dos *kernels* é a geração de tempo ocioso durante as execuções dos métodos. Isto pois, os tempos de execução das ações são distintos, e cada recurso tem suas próprias latências inerentes, como o acesso à memória, a alocação de recursos, os cálculos aritméticos, etc.

O objetivo da otimização de latência, é garantir que exista concorrência suficiente nas *threads* em execução, de tal forma, que a latência seja ocultada ou mascarada no processo. Em outras palavras, busca-se sincronizar a quantidade de operações com taxa de aquisição de dados, garantindo que sempre que um conjunto de *threads* estejam aptos a executar seus dados também estejam disponíveis e prontos, viabilizando a execução imediata.

Uma técnica empregada, é a escolha de tamanhos de blocos múltiplos da dimensão do *warp*, garantido que sempre um *warp* inteiro esteja executando a mesma operação, isto significa que a maior quantidade de *threads* disponível esta em execução e consequentemente uma maior ocupância.

Com o auxílio das ferramentas de *profiling*, algumas vezes, uma abordagem heurística é aplicada para encontrar tamanhos ótimos de grades para o lançamento dos *kernels*, maximizando a ocupância e mascarando a latência.

3.3.3 Otimizações de Instruções

Além de limitações por banda de memória e latência, existem alguns algoritmos que possuem alta taxa de computação, e portanto são limitados pelo rendimento das instruções. Ou seja, existem dados suficientes para a execução da operação, mas sua complexidade computacional é tão elevada, que demanda muito tempo para a realização dos cálculos. Neste tipo

de *kernel* é necessário organizar e otimizar as operações que devem ser feitas, para evitar que o gargalo de execução seja as instruções.

Desta forma, o principal objetivo deste tipo de otimização é reduzir a quantidade de instruções totais do método. Utilizar menos instruções para realizar a mesma tarefa, significa que, o mesmo resultado de um método é obtido utilizando menos recurso e consequentemente, menos ciclos de *clock*.

Uma técnica comum para a solução deste tipo de limitante é a utilização de instruções com alto rendimento, que são operações que executam a mesma função, porém com uma quantidade superior de dados. Um exemplo deste tipo de operação são as primitivas SIMD (*Single-Instruction, Multiple-Data*) e SIMT (*Single-Instruction, Multiple-Threads*) fornecidas pela API CUDA.

Uma operação SIMD é definida como uma instrução que performa de forma vetorial os dados, isso significa que ela consome uma quantidade maior de dados em uma mesma instrução. Um exemplo, discutido na Seção 4.4.2, deste tipo de instrução é o `__vsetminu4` fornecida pela API CUDA. Esta operação recebe como parâmetro não um inteiro, mas um conjunto de quatro inteiros, e computa a mesma operação utilizando esses dados, ou seja, o rendimento desta operação seria quatro vezes maior que a execução de quatro instruções que operam em um só inteiro.

Diversas operações foram vetorizadas em GSL utilizando os arranjos dos *bytes* para garantir a maior eficiência de execução, buscando sempre evitar a divergência de fluxo nos *warps* e reduzindo os conflitos nos bancos de memória.

3.4 Conclusões do Capítulo

O presente capítulo apresenta de forma direta as principais características da computação em um sistema heterogêneo. São abordados os aspectos da arquitetura deste tipo de sistema, as principais características das GPUs e as diferenças em relação à computação convencional. São abordados também, as principais etapas do fluxo de execução de um sistema heterogêneo, a relação entre o código executado no *Host* (CPU) e no *Device* (GPU), bem como os pontos mais pertinentes da linguagem e API que fornecem as diretrizes e interfaces para a execução do código.

Na Seção 3.2 são discutidos os elementos principais de um sistema heterogêneo, em especial, os presentes na placa embarcada NVIDIA Jetson Nano. Isto é, os elementos que compõe o modelo de programação dos chips gráficos, como *kernels*, a hierarquia de *threads*, e a hierarquia de memórias. Esta seção, é a base para o entendimento do modelo de programação abordado no trabalho, fornecendo o conhecimento necessário para a produção de uma implementação eficiente.

Neste sentido, a Seção 3.3 sumariza, de forma precisa, as principais otimizações utilizadas na elaboração do projeto. Em um escopo geral, são discutidos os principais pontos de otimização e também as técnicas que permitem identificar e corrigir os gargalos de execução dos algoritmos heterogêneos. Esta seção, serve como suporte para entender todas as otimizações que possibilitam a execução dos métodos com o uso otimizado do hardware disponível, que no caso, possui recursos limitados por se tratar de um sistema embarcado convencional. Essas otimizações são necessárias para garantir que as implementações propostas, e discutidas nos capítulos seguintes, atendam os requisitos para a execução em tempo real.

Capítulo 4

Sistema Embarcado

Nos Capítulos 2 e 3 são apresentados e discutidos os conceitos básicos e os algoritmos propostos para a solução do problema de detecção de faixas de trânsito, bem como as principais otimizações utilizadas na implementação. Ao passo que na Seção 4.1 são analisadas as características do sistema embarcado adotado para o projeto. Além disso, é apresentado o software embarcado proposto, expondo as implementações dos algoritmos e as interações do sistema embarcado, sumarizando de forma minuciosa toda a arquitetura do sistema embarcado proposto para a solução do problema de detecções de faixas centrais em tempo real.

Assim, são apresentadas as arquiteturas e implementações da solução sequencial do problema, bem como a arquitetura da abordagem heterogênea e suas principais características. Por fim, são feitas algumas ponderações sobre a versão final da solução, apresentando todo o fluxo de processamento.

4.1 Arquitetura do Hardware Embarcado

Os computadores de placa única, como os sistemas embarcados convencionais se tornaram muito populares entre os entusiastas e pesquisadores, com os avanços no poder de processamento e principalmente com o baixo custo dos processadores e circuitos integrados (Li, J. et al., 2019). Atualmente, a quantidade deste tipo de hardware disponível no mercado é enorme e, portanto, é inviável comparar todos os modelos disponíveis. Foi realizada uma análise comparativa entre diversos sistemas embarcados de prateleira. Alguns fatores foram levados em consideração, tais como: marca e arquitetura do processador, quantidade e velocidade do

núcleo, tipo da GPU, tamanho e tecnologia de memória, quantidade e tipo de periféricos entre outros parâmetros.

Dito isso, afim de selecionar um hardware adequado e acessível para desenvolver o projeto, uma breve comparação de seis diferentes placas foi feita. Os sistemas embarcados selecionados são: ASUS TinkerBoard S (ASUS, 2019), ODROID XU4Q (HARDKERNEL, 2020b), ODROID N2 (HARDKERNEL, 2020a), VIM3 Pro (KHADAS, 2020), NVIDIA Jetson Nano (NVIDIA, 2019b) e RaspberryPi 4B (FOUNDATION, 2019).

As Tabelas 4.1 e 4.2 apresentam os dados coletados para os hardwares avaliados. A primeira tabela, apresenta os dados com relação as características da CPU e da memória de cada sistema embarcado. Ao passo, que a segunda tabela apresenta as principais características da GPU do sistema.

Tabela 4.1: Lista de comparação dos diferentes hardwares embarcados - Características de CPU e Memória

Sistema Embarcado	**Arquitetura CPU**	**Clock (GHz)**	**Num. de Núcleos**	**L2 Cache (MB)**	**Memória (MB)**	**Tipo da Memória**
ODROID XU4Q	ARM Cortex A15+A7	2,0 + 1,4	4+4	2,0+0,512	2048	LPDDR3
ODROID N2+	ARM Cortex A73+A53	2,4 + 2,0	4+2	2,0+0,512	4096	LPDDR4
VIM3 Pro	ARM Cortex A73+A53	2,2 + 1,8	4+2	2,0+0,512	2048	LPDDR4
Jetson Nano	ARM Cortex A57	1,43	4	2	4096	LPDDR4
Raspberry Pi 4	ARM Cortex A72	1,5	4	1	4096	LPDDR4
ASUS TinkerBoard S	ARM Cortex A17	1,8	4	1	2048	LPDDR3

Nota: O símbolo "+"indica que existem dois processadores no sistema embarcado. Ao passo que valores separados por "+"são relacionados a cada um dos processadores.

Tabela 4.2: Lista de comparação dos diferentes hardwares embarcados - Características da GPU Embarcada

Sistema Embarcado	**Arquitetura GPU**	**Clock (MHz)**	**Num. de Núcleos**	**Principais APIs Suportadas**
ODROID XU4Q	ARM Mali-T628	600	6	OpenGL ES 3.1; OpenCL 1.2
ODROID N2+	ARM Mali-G52	846	6	OpenGL ES 3.1; OpenCL 2.1
VIM3 Pro	ARM Mali-T820MP3	800	6	OpenGL ES 3.1; OpenCL 2.1
Jetson Nano	128-core NVIDIA Maxwell	921	128	CUDA 10; Open GL ES 3.2
"Raspberry Pi 4	VideoCore VI 3D	500	1	OpenGL ES 3.1; OpenCL 2.1
ASUS TinkerBoard S	ARM Mali T764	600	4	OpenGL ES 3.1; OpenCL 1.2

Todos os hardwares avaliados estão na mesma categoria de preço e cumprem a mesma função, que é fornecer um sistema heterogêneo embarcado, dotados de processadores com múltiplos núcleos e também de chips gráficos integrados.

Como o principal objetivo do projeto, e por consequência o maior requisito do sistema embarcado, é a execução do algoritmo paralelo na chip gráfico, o requisito de maior impacto é a capacidade da GPU. Relacionando os dados das duas tabelas, constatou-se que o

hardware mais adequado para o projeto seria a placa NVIDIA Jetson Nano. Ela possui um processador com quatro núcleos de 1,43 GHz e possui um chip de processamento gráfico superior aos outros, bem como o suporte a API CUDA 10, que é utilizada para a implementação das operações gráficas na GPU. Conforme mencionado anteriormente, é proposta uma implementação que faz uso da GPU para a computação paralela. Assim, de acordo com a Tabela 4.2, nota-se que a NVIDIA Jetson Nano possui a GPU mais potente entre os exibidos, apresentando 0,5 TFLOPS (NVIDIA, 2019a).

Não obstante, concluí-se que a melhor escolha de hardware para realização do projeto proposto é a NVIDIA Jetson Nano e, portanto, foi o escolhido já que possui custo e poder de processamento similar aos outros modelos e uma GPU com a maior capacidade entre as analisadas.

A Figura 4.1 ilustra o diagrama de blocos geral da NVIDIA Jetson Nano, enquanto a Figura 4.2 mostra suas características mecânicas. O objetivo de ambas as imagens é apresentar a disposição espacial dos componentes internos e as interfaces presentes no sistema embarcado adotado.

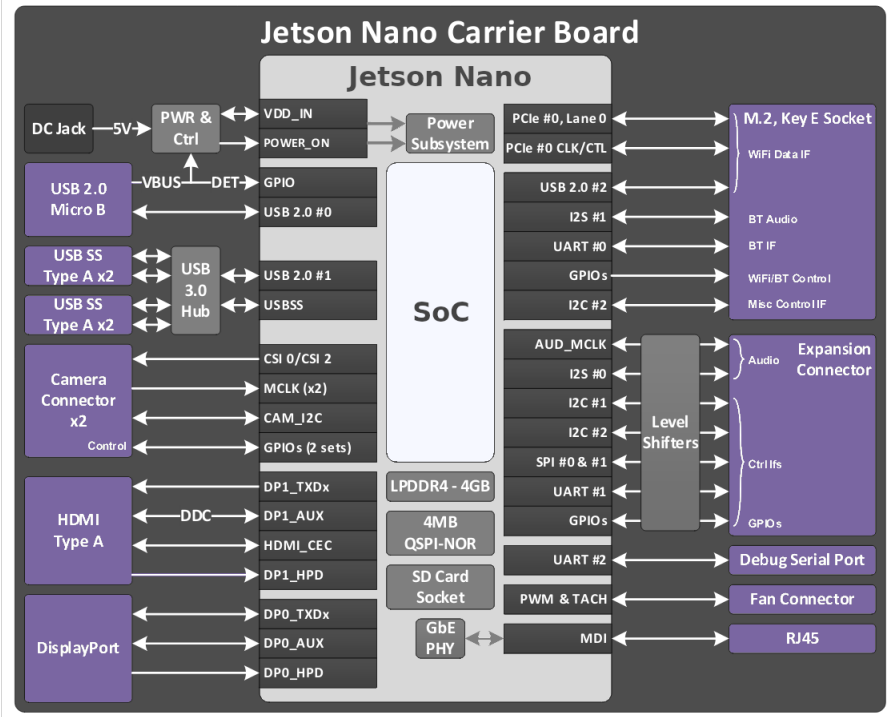


Figura 4.1: Diagrama de blocos simplificado da Nvidia Jetson Nano.

Ressaltando as principais características e periféricos da placa de desenvolvimento disponível na Figura 4.1, é possível observar a presença da memória RAM de 4 GB LPDDR4, o suporte ao SD Card onde o sistema operacional e os dados serão armazenados, a interface

de comunicação UART para a comunicação entre o sistema embarcado e o computador de desenvolvimento, a interface USB SS 3.0 para a comunicação com a câmera externa, a presença de um conector Ethernet que é utilizado para uma conexão SSH para o desenvolvimento e depuração.

A Figura 4.2 busca apresentar as característica mecânicas do sistema embarcado e são sinalizadas os principais recursos. Em amarelo são apresentados os pontos visíveis da placa de desenvolvimento, ao passo que em vermelho estão destacados os módulos que estão, de certa forma, sobrepostos por outros componentes, como o chip Tegra X1 e o conector M.2.

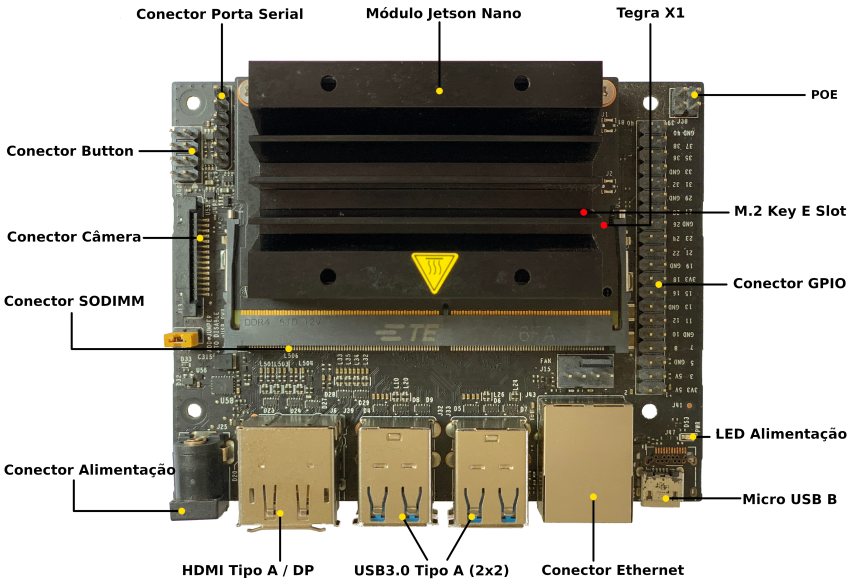


Figura 4.2: Detalhes físicos da Nvidia Jetson Nano.

Por fim, esta seção destaca os recursos do *Tegra X1 System-On-Chip (SoC)* presentes na NVIDIA Jetson Nano. Os quatro núcleos ARM são projetados com o conjunto de instruções da arquitetura ARMv8 e, de acordo com NVIDIA (2019a), algumas de suas principais características são:

- taxa de clock da CPU de até 1,43 GHz para os núcleos ARM Cortex-A57;
- predição dinâmica com *Branch Target Buffer* (BTB) e Buffer de Histórico Global da memória RAM, um retorno de pilha e um preditor indireto;
- caches de dados e instrução fixos de 32KB L1;
- um cache compartilhado de 2 MB L2;

- tamanho de linha fixado de 64 bytes para L2;
- suporte de ECC (*Error Correction Code*);
- uma unidade de ponto flutuante vetorial versão 4 (VFPv4), totalmente compatível com o padrão IEEE 754.

Por fim, todas as informações relevantes sobre o sistema embarcado estão reunidas na Tabela 4.3.

Tabela 4.3: Jetson Nano - Especificações

CPU	ARM Cortex A57 Quad-Core 1,43 GHz
GPU	NVIDIA 128 Core Maxwell @ 472 GFLOPs (FP16) OpenGL 4.6, Open GL ES 3.2, Vulkan 1.1, DirectX 12 e CUDA 10
Memória	4 GB 64 bit LPDDR4 25,6 GB/s
Armazenamento	32 GB eMMC 5.0 (externo)
Periféricos	1-x1/2/4 PCIE; 4-USB3.0; 1-HDIM; 1-Ethernet; 1-MicroSD slot;
Interfaces	1-SDIO; 2-SPI; 5-SysIO; 13-GPIO; 6-I2C
Alimentação	5V / 4A
Tamanho e Peso	59,6 x 45 mm, 140 g

Como mencionado anteriormente, os principais recursos e especificações observadas no sistema são em relação a suas capacidades computacionais, tanto no desempenho do processador com quatro núcleos, quanto na GPU de 0,5 TFLOPS. As características físicas do hardware também são adequadas ao projeto. Para uma lista completa das especificações e recursos está disponível no manual do usuário (NVIDIA, 2019a).

4.2 Arquitetura do Software Embarcado

Conforme apresentado na Seção 1.2, o principal objetivo deste trabalho, é o desenvolvimento de um sistema capaz de realizar a detecção de faixas centrais em tempo real, de forma embarcada a partir de imagens fornecidas por uma câmera (ou pré-capturadas). Além disso, o software deve registrar todos os dados gerados em cada ciclo de execução e enviá-los ou armazená-los como telemetria. A Figura 4.3 apresenta um diagrama esquemático geral para um sistema de processamento de imagem e telemetria, em especial é apresentado o fluxo geral do projeto baseado nas etapas de pré-processamento, extração de características, e estimação de faixas (SCARAMAL, 2017; LEE; MOON, 2018; ALMAGAMBETOV; VELIPASALAR; CASARES, 2015).

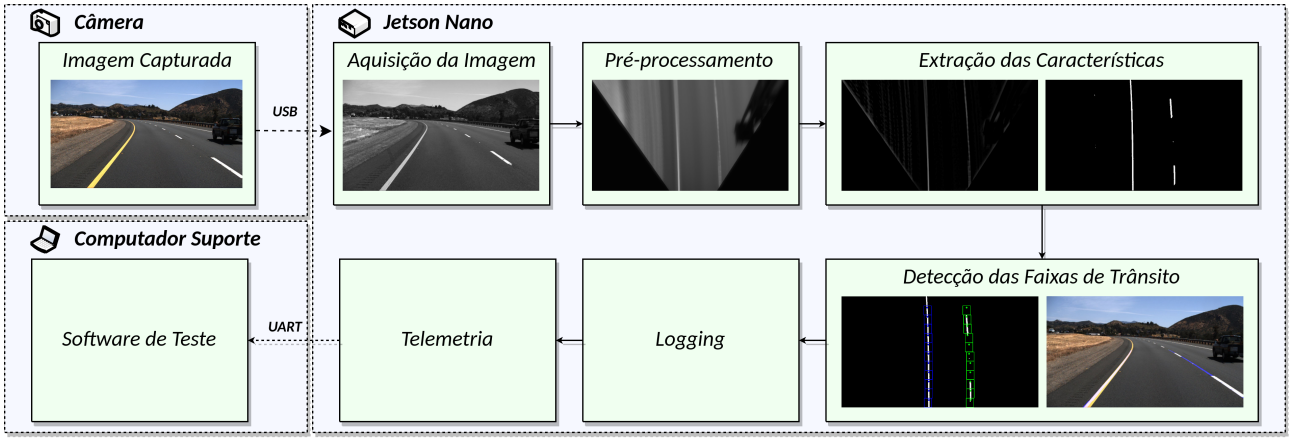


Figura 4.3: Diagrama simplificado do software completo.

Inicialmente, no diagrama da Figura 4.3, uma imagem é capturada por uma câmera e enviada para o sistema embarcado (NVIDIA Jetson Nano) [1]. Então, este quadro é convertido em uma imagem em escala de cinza e pré-processada, sendo então inserida no algoritmo de extração de características, com o intuito de criar uma imagem binária que contenha apenas as faixas. Após a detecção, a imagem binária é então processada pelo método de estimação das faixas, que com base no mapa de características detecta os candidatos as faixas de trânsito. Por fim, as faixas de trânsito resultantes, são salvas no armazenamento interno do sistema embarcado e empacotadas.

Visando proporcionar uma implementação eficiente, o software foi desenvolvido em linguagem de programação C++ na versão 7.5.0 e na API CUDA 10.2 (NVIDIA, 2020)[2], o uso de bibliotecas externas foi evitado sempre que possível, buscando minimizar o tempo de execução e a complexidade do programa. Contudo, para os experimentos discutidos na Seção 5.2 são implementados três programas distintos, um totalmente serial utilizando C++, outro utilizando uma versão compilada para GPU da biblioteca de processamento de imagens OpenCV e outra versão paralela utilizando CUDA.

4.3 Algoritmo - Abordagem Serial (CPU)

O software serial proposto é baseado na iteração de um laço principal, as etapas de pré-processamento, extração de características e estimação de faixas são realizadas utili-

[1] na versão baseada em imagens pré-capturadas da base de dados, o processo de aquisição dos quadros é feita através da leitura da base de dados já armazenada do dispositivo

[2] a API CUDA foi utilizada apenas na versão paralela da implementação.

zando a imagem atual de entrada e as informações armazenadas previamente no sistema, como constantes e outras imagens. Antes desta etapa iterativa, são armazenados os parâmetros e constantes reutilizadas ao longo do processo em um etapa de inicialização. Estes parâmetros são por exemplo, as matrizes de transformação, o primeiro quadro para o método de integração temporal, a alocação de memórias dinâmicas, entre outros recursos.

Um diagrama funcional desta etapa é exibido na Figura 4.4. São apresentadas as principais funções do Loop Principal, como as condicionais de execução e as etapas do método de detecção de faixas centrais.

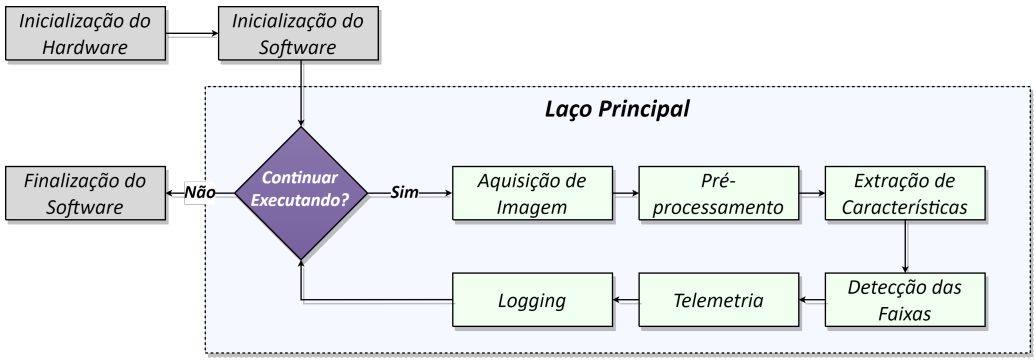


Figura 4.4: Diagrama simplificado da execução do software embarcado proposto.

O laço principal é controlado por uma variável inteira, que retém um valor positivo a menos que uma tecla seja pressionada [3], o que zera a variável e inicia os processos de finalização de software e hardware. O software foi inicialmente projetado para consumir informações da base de dados local, utilizando descritores para realizar a leitura de forma contínua. Similarmente, para a utilização de uma câmera externa é necessário que a interface gere uma forma de gatilho, como uma interrupção detectada pelo software para então consumir um novo quadro.

4.3.1 Aquisição de Imagens

A etapa de aquisição de imagens pode ser realizada de duas formas, através de uma câmera externa ou de imagens pré-gravadas na base de dados. Do ponto de vista de execução do algoritmo, não há nenhuma diferença. A Figura 4.5 apresenta um diagrama funcional da etapa de aquisição de imagens, mostrando em especial, a condicional de seleção do método de entrada de dados.

[3] interrupções na aquisição da imagem também provocam a saída do método.

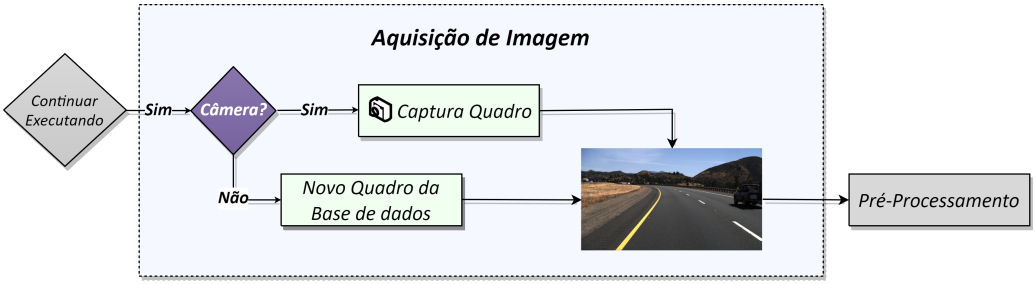


Figura 4.5: Diagrama simplificado do processo de aquisição de imagem no sistema embarcado.

Em ambos os casos, câmera externa e base de dados, as imagens são adquiridas através de funções pré-programadas da biblioteca de manipulação de imagens OpenCV nas versões seriais ou NVJPEG na versão paralela, fato que será discutido ao longo do texto.

Além disso, a biblioteca OpenCV utiliza uma API de captura de imagem que inclui apenas um pequeno conjunto de funções para captura. Em especial, a biblioteca OpenCV é programada para converter toda imagem para o formato BGR [4], não possibilitando a leitura dos vídeos em outros formato de píxeis, sendo necessário uma conversão para a escala de cinza (OPENCV, 2020), o que não ocorre com a NVJPEG.

4.3.2 Pré-processamento

Esta etapa tem como principal objetivo realizar o tratamento da imagem recém aquisitada, garantindo que a mesma esteja nas condições mais corretas para a etapa de extração de características. Isto é, uma imagem em escala de cinza, em perspectiva BEV e integrada temporalmente, obtendo uma melhor qualidade das marcações das faixas de trânsito.

A etapa de pré-processamento é descrita de forma simplificada através do diagrama da Figura 4.6. É apresentado de forma funcional os três principais métodos desta etapa: *i)* a conversão para escala de cinza; *ii)* a transformação inversa de perspectiva; *iii)* e o desvanecimento temporal.

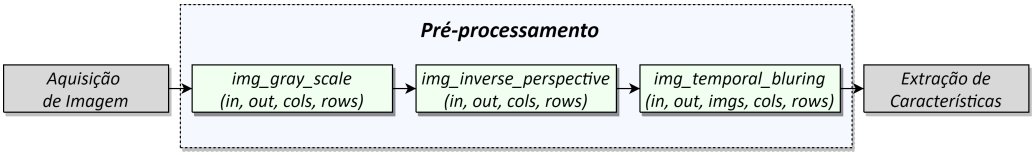


Figura 4.6: Diagrama simplificado da etapa de pré-processamento no sistema embarcado.

[4] espaço de cores padrão (*Blue-Green-Red*)

Conversão para Escala de Cinza

A primeira tarefa da etapa de pré-processamento é tratar a imagem de entrada para um formato adequado para a operação. Como mencionado, a aquisição das imagens são feitas no espaço de cores BGR, contudo, deseja-se que as imagens sejam convertidas para a escala de cinza, como apresentado na Seção 2.1.4.

Assim, através da Equação 2.6 é realizada a conversão da imagem original para a escala de cinza. O Código 4.1 apresenta a implementação do método de conversão de escala BGR para cinza, proposta na versão serial, executada na CPU embarcada.

```
1 void img_gray_scale(const unsigned char *data_in, unsigned char *data_out, int cols, int
      rows)
2 {
3   int index = 0;
4   for (int row = 0; row < rows; ++row)
5   {
6     for (int col = 0; col < cols*3; col+=3)
7     {
8       data_out[index] = 0.299 * data_in[row * cols + col] +
9                 0.587 * data_in[row * cols + col + 1] +
10                0.114 * data_in[row * cols + col + 2];
11      index++;
12    }
13  }
14 }
```

Código 4.1: Conversão de BGR para escala de cinza (serial)

Transformação de Perspectiva (IPM)

De posse da imagem em escala de cinza, é necessário extrair a ROI adequada para o método. Esta região é obtida pela transformação para a perspectiva BEV, através do cálculo da IPM, conforme apresentado na Seção 2.1.5. Para isto, são previamente armazenados os oito pares de coordenadas utilizadas na transformação. Na Inicialização do Software são calculadas as matrizes de transformação, estas são estáticas (constantes para todas as imagens) e portanto podem ser escritas diretamente na memória. Uma matriz de transformação de exemplo é apresentada por `m` no Código 4.2 que apresenta a implementação serial proposta para ser executada na CPU embarcada.

A implementação apresentada no Código 4.2 realiza para cada píxel da imagem de entrada, o cálculo da IPM (Equação 2.8) utilizando como base os valores pré-computados da matriz `m`. As conversões obtidas em cada uma das iterações são então escritas em um novo píxel de saída, formando a imagem BEV.

```
void img_inverse_perspective(const unsigned char *data_in, unsigned char *data_out,
    unsigned char *outMin, unsigned char *outMax, int rows, int cols)
{
  /** Matriz de transformacao. */
  const float m[] =
  {
    -5.0780827e-01f, -2.8630407e+00f, 1.0020643e+03f,
    -8.7075156e-16f, -4.0855622e+00f, 1.4294390e+03f,
    -9.8639541e-19f, -4.3080235e-03f, 1.0000000e+00f
  };

  /** Coordenadas 3D*/
  float x1 = 0.0, y1 = 0.0, z1 = 0.0;

  /** Coordenadas 2D transformadas. */
  int i2 = 0, j2 = 0;

  for (int j = 0; j < rows; ++j)
  {
    for (int i = 0; i < cols; i++)
    {
      /** Calculo das coordenadas 3D. */
      x1 = ((m[0]*i) + (m[1]*j)) + m[2];
      y1 = ((m[3]*i) + (m[4]*j)) + m[5];
      z1 = ((m[6]*i) + (m[7]*j)) + m[8];

      /** Indeces para a nova matriz (2D). */
      i2 = (int)(x1 / z1 + 0.5f);
      j2 = (int)(y1 / z1 + 0.5f);

      /** Montagem da nova matriz (2D). */
      if (i2 >= 0 && i2 < cols && j2 >= 0 && j2 < rows)
      {
        if (data_out_min[j2 * cols + i2] > data_in[j * cols + i])
        {
          data_out_min[j2 * cols + i2] = data_in[j * cols + i];
        }

        if (data_out_max[j2 * cols + i2] < data_in[j * cols + i])
        {
          data_out_max[j2 * cols + i2] = data_in[j * cols + i];
        }

        data_out[j2 * cols + i2] = data_in[j * cols + i];
      }
    }
  }
}
```

Código 4.2: Função para a transformação inversa de perspectiva (serial)

Desvanecimento Temporal

Após obter a imagem em perspectiva BEV em escala de cinza é realizada uma operação de desvanecimento temporal, isto é, a imagem atual é integrada com um conjunto de quadros adquiridos nas iterações anteriores. Este processo é detalhado na Seção 2.1.3 e uma visualização simplificada do método é apresentada na forma de um diagrama na Figura 4.7.

Esta etapa tem como principal objetivo aprimorar a qualidade das sinalizações das faixas de trânsito e remover possíveis interferências, provendo uma imagem mais adequada para a extração de características. É utilizada uma lista de quadros de tamanho fixo, onde são armazenadas as últimas imagens processadas. Esta lista, possibilita remover a contribuição mais antiga, sempre que uma nova imagem é processada, evitando saturar a imagem e dando a característica de janela temporal para o algoritmo.

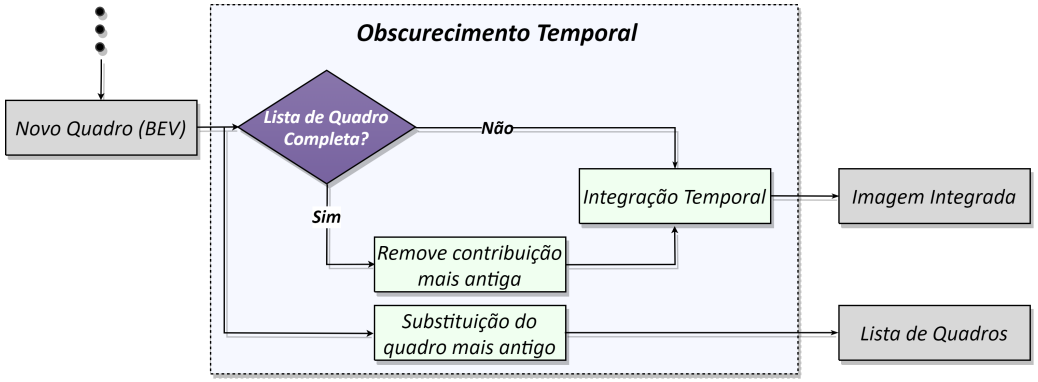


Figura 4.7: Diagrama lógico da implementação do algoritmo de obscurecimento temporal proposto.

O Código 4.3 apresenta a implementação serial do método de desvanecimento temporal.

A primeira etapa do algoritmo corresponde ao preenchimento do arranjo de quadros, onde cada imagem é acumulada até que um valor mínimo de quadros seja atingido, neste caso definido por `NUM_TEMPORAL`. Assim, são integrados os quadros sem a realização da remoção do mais antigo, ao passo, que quando existem quadros suficientes no arranjo, os mais antigos são removidos, como apresentado na segunda condicional do código. Essa remoção, além de criar uma janela temporal fixa, evitando que imagens muito antigas interfiram nas novas, evita também a saturação da imagem final.

4.3.3 Extração de Características

Esta etapa tem como objetivo elaborar os mapas de características e combiná-los para obter uma imagem ideal, para que o método de estimação das faixas, realize as detecções. São combinados os dois mapas de características e então transmitidos para a etapa de estimação das faixas. O diagrama da Figura 4.8 apresenta de forma funcional a operação de extração de características. Os dois algoritmos escolhidos a implementação final são um exemplo de possíveis combinação de mapas utilizados nesta etapa.

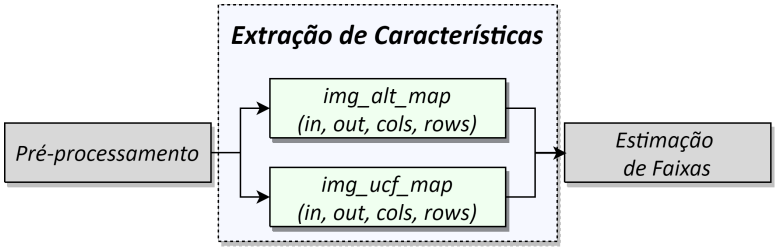


Figura 4.8: Diagrama simplificado de execução da etapa de extração de características.

```
1 void img_temporal_blurring_serial(const cv::Mat &input, cv::Mat &output, cv::Mat *imgs, int
      num_frame)
2 {
3   if (num_frame <= NUM_TEMPORAL)
4   {
5     imgs[num_frame-1] = input;
6
7     /* Integracao da nova contribuicao na imagem final. */
8     for (int i = 0; i<input.cols; i++)
9     {
10      for (int j = 0; j<input.rows; j++)
11      {
12        output.data[j* input.step + i] +=
13          input.data[j * input.step + i]/NUM_TEMPORAL;
14      }
15    }
16  }
17  else
18  {
19    /* Remocao da contribuicao mais antiga da imagem final. */
20    for (int i = 0; i<input.cols; i++)
21    {
22      for (int j = 0; j<input.rows; j++)
23      {
24        int idx = j * imgs[(num_frame-1)%NUM_TEMPORAL].step + i;
25        output.data[j* output.step + i] -=
26          imgs[(num_frame-1)%NUM_TEMPORAL].data[idx]/NUM_TEMPORAL;
27      }
28    }
29    /* Integracao da nova contribuicao na imagem final. */
30    for (int i = 0; i<input.cols; i++)
31    {
32      for (int j = 0; j<input.rows; j++)
33      {
34        output.data[j* output.step + i] +=
35          input.data[j * input.step + i]/NUM_TEMPORAL;
36      }
37    }
38
39    /* Atualizacao do arranjo de imagens. */
40    imgs[(num_frame-1)%NUM_TEMPORAL] = input;
41  }
42 }
```

Código 4.3: Método de desvanecimento temporal recursivo (serial)

Obtenção do Mapa de Características I^{ALT}

A imagem obtida na etapa de pré-processamento é utilizada como base para a elaboração do mapa. É obtido o valor de luminância média, L_{med}, que é utilizado para ajustar os valores dos limiares adaptativos do filtro, conforme discutido na Seção 2.2.1. O Código 4.4 apresenta a função utilizada para a geração deste mapa de característica.

De forma simplificada, a primeira etapa da implementação obtém o valor de luminância média acumulando os valores de cada píxel da imagem. De posse deste valor médio são obtidos os limiares bases para o filtro de binarização. Nesta etapa, cada píxel da imagem é validado segundo os limiares e então é binarizado.

Obtenção do Mapa de Características I^{UCF}

Os detectores de borda convencionais não são adequados para implementação de sistema embarcado em tempo real, devido ao seu custo computacional, como observado na

```
1 void img_alt_map_serial(const cv::Mat &input, cv::Mat &output)
2 {
3   int tu, tl;
4
5   long int ll_avg = 0;
6
7   unsigned char *data_in = (unsigned char *)(input.data);
8   unsigned char *data_out = (unsigned char *)(output.data);
9
10  /* Acumula a luminancia da imagem. */
11  for (int i = 0; i < input.cols; i++)
12  {
13    for (int j = 0; j < input.rows; j++)
14      ll_avg += data_in[i + j*input.cols];
15  }
16
17  /* Calcula a media da luminancia da imagem. */
18  float l_avg = (float) ll_avg / (input.cols*input.rows);
19
20  /* Obtem o valor tabelado dos limiares. */
21  get_threshold_value(l_avg, &tl, &tu);
22
23  /* Filtro de binarizacao da imagem. */
24  for (int i = 0; i < input.cols; i++)
25  {
26    for (int j = 0; j < input.rows; j++)
27    {
28      unsigned char data = data_in[i + j*input.cols];
29
30      if ( tl < data && tu > data)
31        data_out[i + j*input.cols] =  1;
32      else
33        data_out[i + j*input.cols] =  0;
34    }
35  }
36 }
```

Código 4.4: Função auxiliar para a determinação dos valores de limiar

implementação inicial realizada em (SILVA et al., 2020). Portanto, uma forma de minimizar este gargalo e extrair essas características salientadas na etapa de pré-processamento, é proposto um filtro de correlação unilateral (ZHICHENG ZHANG, 2019; ZENG et al., 2015) como solução. Este tipo de filtro, possui menor complexidade computacional e tem como base a convolução da imagem com um núcleo de separável.

O filtro implementado é uma matriz de terceira ordem, que extrai um único tipo de borda na imagem, no caso, a borda vertical, causada pela variação abrupta de intensidade de luz entre o asfalto e a marcação da faixa. Como as marcações das faixas são quase verticais na imagem BEV e tendem a ser contínuas pela integração temporal, o filtro é projetado para extrair bordas verticais, detectando variações horizontais nos píxeis. O Código 4.5 apresenta a implementação serial do extrator de características para a geração deste mapa.

A variável `k`, no Código 4.5, é responsável por armazenar o núcleo da convolução utilizada para a extração de bordas, ao passo que o laço de iteração percorre a imagem completa, realizando o cálculo da convolução, como descrito na Seção 2.2.

```
1 void img_ucf_map_serial(const cv::Mat &input, cv::Mat &output)
2 {
3   const int k[] = {1,2,1,0,0,0,-1,-2,-1};
4
5   unsigned char *data_in = (unsigned char *)(input.data);
6   unsigned char *data_out = (unsigned char *)(output.data);
7
8   int step = input.cols;
9
10  for (int i = 1; i < input.cols-1; i++)
11  {
12    for (int j = 1; j < input.rows-1; j++)
13    {
14      int val =   (k[0] * data_in[(j-1)*step + (i-1)]) +
15            (k[1] * data_in[(j+0)*step + (i-1)]) +
16            (k[2] * data_in[(j+1)*step + (i-1)]) +
17            (k[3] * data_in[(j-1)*step + (i+0)]) +
18            (k[4] * data_in[(j+0)*step + (i+0)]) +
19            (k[5] * data_in[(j+1)*step + (i+0)]) +
20            (k[6] * data_in[(j-1)*step + (i+1)]) +
21            (k[7] * data_in[(j+0)*step + (i+1)]) +
22            (k[8] * data_in[(j+1)*step + (i+1)]);
23
24      if(val < 0) val = 0;
25      if(val > 255) val = 255;
26
27      data_out[j*step + i] = val;
28    }
29  }
30 }
```

Código 4.5: Implementação serial do extrator de bordas para geração do mapa I^{UCF}

4.3.4 Estimação de Faixas

Após a geração dos mapas de características é iniciado o processo de estimação das faixas, que busca encontrar os píxeis que pertencem as sinalizações no asfalto e obter suas coordenadas e então, utilizá-las para obter um ajuste polinomial que se adeque a curvatura da faixa detectada.

Para tal, é proposto o método de janelas deslizantes, como apresentado na Seção 2.3. O método em questão é subdivido em duas etapas, a detecção dos picos iniciais, através do método de histograma no mapa de características e o método iterativo para a detecção das regiões pertencentes as faixas. Este método iterativo é também baseado na análise via histograma de coluna, porém é restringido as regiões delimitadas pelos sensores virtuais e é realizado o cálculo da densidade de píxeis, ao invés dos picos.

Para melhor visualização desta etapa, foi elaborado um diagrama simplificado da operação conforme apresentado na Figura 4.9. O fluxo apresenta de forma funcional a implementação do método de estimação das faixas, através do algoritmo de janelas deslizantes e da obtenção dos pontos iniciais através do método de histograma de colunas.

Uma visão mais profunda da operação pode ser detalhada nos Códigos 4.6 e 4.7. O primeiro apresenta a implementação proposta para a solução do problema de detecção dos

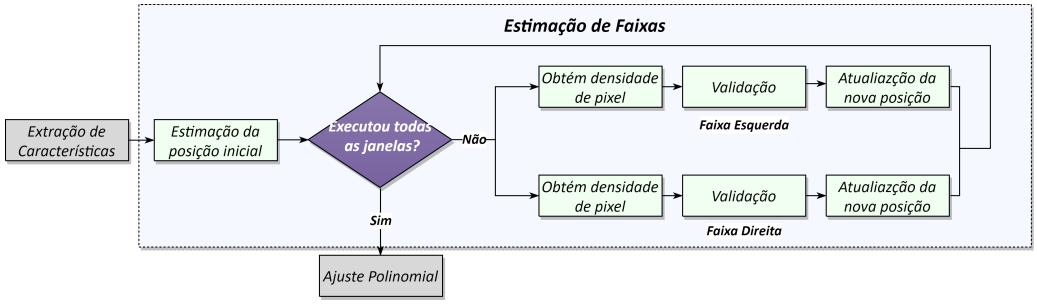


Figura 4.9: Diagrama simplificado da operação proposta para a estimação da posição das faixas de trânsito.

pontos iniciais das faixas, ao passo que o segundo, apresenta a implementação do método de estimação das posições das faixas centrais ao longo da imagem.

Estimação da Posição Inicial - Histograma de Colunas

O Código 4.6 apresenta o algoritmo ilustrado no primeiro bloco 'Estimação da posição inicial' da Figura 4.9. Esta etapa é crucial para a detecção das faixas centrais, pois gera os pontos iniciais da entrada do método de estimação. Esses são, de forma simplificada, as duas posições horizontais de maior probabilidade de conter as faixas centrais da via.

```
1 void img_histogram_serial(const cv::Mat &input, uint32_t *p1, uint32_t *p2)
2 {
3   uint32_t pos_left = 0, pos_right = 0, val_left = 0, val_right = 0;
4   uint32_t histogram[input.cols] = {0};
5
6   /* Calculo do valor acumulado de cada coluna. */
7   for (uint32_t i = 0; i < input.cols; i++)
8   {
9     for (uint32_t j = input.rows / 2; j < input.rows; j++)
10      histogram[i] += input.data[j * input.step + i];
11  }
12
13  /* Obtencao dos valores de posicao e intensidade de cada pico desejado. */
14  for (uint32_t i = 0; i < input.cols; i++)
15  {
16    if (val_left < histogram[i] && i < input.cols / 2)
17    {
18      pos_left = i;
19      val_left = histogram[i];
20    }
21    else if (val_right < histogram[i] && i >= input.cols / 2)
22    {
23      pos_right = i;
24      val_right = histogram[i];
25    }
26  }
27
28  /* Atualizacao das posicoes iniciais. */
29  *p1 = pos_left;
30  *p2 = pos_right;
31 }
```

Código 4.6: Detecção dos picos de intensidade no histograma inicial sequencial utilizando OpenCV

Para a obtenção destes valores, uma ROI da imagem é selecionada, no caso, a metade inferior da imagem. Esta escolha é feita por três motivos, o primeiro é uma maior precisão na posição da faixa, visto que curvas na parte superior da imagem teriam impacto da posição estimada, causando um viés nas detecções nessa situação. Além disto, devido as características

da imagem BEV, a região inferior, contém uma presença menor das faixas laterais, salientando apenas as centrais. O último motivo é a redução da carga computacional, uma vez que são realizadas apenas metade das iterações que seriam necessárias, já que o método processa todos os píxeis da ROI, que neste caso corresponde a metade da dimensão da imagem.

O método acumula os valores de cada píxel de uma determinada coluna no vetor `histogram`, que possui dimensão igual ao número de colunas da imagem. Posteriormente, o vetor é dividido ao meio, onde cada uma das metades representam um lado da imagem e então obtém-se o ponto de máximo de cada uma, resultando na posição inicial das faixas. No algoritmo proposto, este método é utilizado para detectar as faixas centrais e remover as laterais, contudo, o mesmo pode ser aplicado para a detecção de múltiplas faixas.

Estimação da Posição das Faixas - Janelas Deslizantes

A segunda etapa da estimação das faixas, definida pelos blocos seguintes à condicional do diagrama da Figura 4.9, representam o método de detecção por janelas deslizantes. Como comentado na Seção 2.3 este algoritmo apresenta um tempo de execução adequado para sistemas embarcados e uma boa precisão. Além disso, todo pré-processamento e extração de características possibilitam extrair alta eficiência na detecção.

```
1 void img_sliding_windows_serial(const cv::Mat &input, int *lanes_left, int *lanes_right,
      uint32_t leftx, uint32_t rightx)
2 {
3   uint32_t lanes_height[NUM_WINDOW+1] = {0};
4
5   int left_low = 0, right_low = 0, y_low = 0;
6   int sum_x, int nonzero;
7
8   for (int w = 0; w < NUM_WINDOW+1; w++)
9   {
10    uint8_t hist_right[input.cols] = {0};
11    uint8_t hist_left[input.cols]  = {0};
12    y_low = input.rows - w * HEIGHT_WINDOW(input.rows) - 1;
13    left_low  = (leftx  >= WIDTH_WINDOW) ? leftx  - WIDTH_WINDOW : 0;
14    right_low = (rightx >= WIDTH_WINDOW) ? rightx - WIDTH_WINDOW : 0;
15    sum_x = 0, nonzero = 0;
16    /* Faixa da esquerda */
17    for (int i = left_low; i < left_low + 2 * WIDTH_WINDOW; i++)
18    {
19      for (int j = y_low - HEIGHT_WINDOW(input.rows)+1; j <= y_low; j++)
20      hist_left[i] += input.data[j * input.step + i];
21      if (hist_left[i] > 0)
22      {
23        sum_x += i;
24        nonzero += 1;
25      }
26    }
27    /* Update da posicao central. */
28    if (nonzero > MIN_WHITE_PIXELS)
29    {
30      uint16_t newl = (uint16_t)(sum_x / nonzero);
31      if (leftx * MAX_HORIZONTAL > newl && leftx * MIN_HORIZONTAL < newl)
32        leftx = newl;
33    }
34    sum_x = 0, nonzero = 0;
35    /* Faixa da direita. */
36    for (int i = right_low; i < right_low + 2 * WIDTH_WINDOW; i++)
37    {
38      for (int j = y_low - HEIGHT_WINDOW(input.rows)+1; j <= y_low; j++)
39      hist_right[i] += input.data[j * input.step + i];
40      if (hist_right[i] > 0)
41      {
```

```
42          sum_x += i;
43          nonzero += 1;
44        }
45      }
46      /* Update da posicao central. */
47      if (nonzero > MIN_WHITE_PIXELS)
48      {
49        int newr = (int)(sum_x / nonzero);
50        if (rightx * MAX_HORIZONTAL > newr && rightx * MIN_HORIZONTAL < newr)
51          rightx = newr;
52      }
53      /* Update da estrutura da faixa. */
54      lanes_left[w]   = leftx;
55      lanes_right[w]  = rightx;
56      lanes_height[w] = (uint16_t)(y_low - HEIGHT_WINDOW(input.rows) / 2);
57    }
58}
```

Código 4.7: Implementação do método de janelas deslizantes serial executado na CPU embarcada

O método de janelas deslizantes, de forma resumida, realiza para cada uma das faixas a análise de pequenas porções da imagem (denominadas janelas ou sensores virtuais) através da detecção de densidade de píxel, obtendo a coluna que possui a maior probabilidade de conter a faixa. Esse método iterativo, consegue percorrer a imagem BEV verticalmente detectando as faixas. Além disso, possibilita também a detecção de mais faixas de trânsito em um cenário com múltiplas faixas, contudo, esta discussão não é abordada neste trabalho.

Este método utiliza como estado inicial as duas posições obtidas na etapa de estimação da posição inicial, que representam as duas regiões da imagem com maior probabilidade de conter as marcações das faixas de trânsito. Para uma análise mais detalhada do algoritmo, o Código 4.7 apresenta a implementação proposta para a detecção de faixas via janelas deslizantes, que é executada na CPU de forma serial.

Como mencionado, o método realiza a iteração sobre o número de janelas, definido no Código 4.7 com `NUM_WINDOW`. Para cada uma das janelas `w` é obtido o valor de pico e a posição de maior densidade de píxeis, esse valor é obtido pela razão entre o valor acumulado dos histogramas e a quantidade de valores não nulos, como calculado nas variáveis `newl` e `newr`.

Ao final da execução do método de janelas deslizantes, os vetores `lanes_left`, `lanes_right` e `las_hneeight`, contêm respectivamente os valores das coordenadas horizontais da faixa da esquerda, da faixa da direita e das posições verticais de cada ponto.

4.3.5 Refinamento de Candidatos

Como mencionado, no método de janelas deslizantes é obtido um conjunto de pontos para cada candidato à faixa de trânsito. Isto é, cada um dos centros dos sensores representam um ponto (coordenadas x e y) deste conjunto. Assim, cada conjunto pode ser ajustado através

de um polinômio quadrático (ou superior) encontrando a curva que se ajusta a este conjunto de pontos, estabelecendo assim os candidatos às faixas de trânsito do quadro atual.

4.3.6 Conclusões sobre a Abordagem Serial (CPU)

As presentes seções apresentam, sobre o ponto de vista prático da implementação, os algoritmos propostos para a detecção de faixas de trânsito executadas na CPU embarcada. São expostos os códigos e diagramas funcionais de cada uma das etapas implementadas na etapa intermediária, isto é, na versão sequencial. Inicialmente, são explorados os aspectos do sistema embarcado, bem como as etapas iniciais do método, tais como, a inicialização dos recursos e a aquisição de imagens. Em seguida, são discutidos os algoritmos de cada uma das etapas subsequentes do processo de detecção de faixas, explorando os métodos de pré-processamento, extração de características e detecção de faixas. São abordadas, de um ponto de vista prático, cada uma das implementações propostas para a solução do problema de detecção de faixas, expondo desta forma os principais componentes da arquitetura do sistema.

4.4 Algoritmo - Abordagem Paralela (GPU)

Esta seção, assim como a Seção 4.3, apresenta a implementação das funções que compõe o método proposto para a detecção de faixas. Contudo, aqui são exibidas de forma sucinta às implementações paralelas de tais algoritmos. Assim, esta seção busca sumarizar as principais contribuições práticas do trabalho, expondo como cada operação foi otimizada ao máximo para atingir a maior eficiência de execução e extrair o máximo de recursos do hardware embarcado.

Como algumas etapas do projeto são feitas de forma serial, ou não têm ganhos substantivos para a execução paralela, estas possuem implementação comum entre as duas versões. Estas etapas são a aquisição de imagem, refinamento de candidatos, telemetria e registro de dados. Deste modo, as outras etapas de processamento são implementadas separadamente para cada uma das versões.

4.4.1 Pré-processamento

A etapa de pré-processamento tem como principal objetivo preparar a imagem para a extração de características. Nesta seção, são apresentadas as implementações heterogêneas

para a conversão para escala de cinza, transformação de perspectiva e integração temporal da imagem de entrada. São discutidas também principais otimizações e característica de cada um dos *kernels* implementados.

Conversão para Escala de Cinza

A operação de conversão para escala de cinza possuí um grande potencial de paralelização, devido a natureza de seus cálculos. Em geral, são consumidos três canais de cores e então é aplicado o cálculo da conversão para um único canal de cor.

Como discutido na seção 3.3.1 este *kernel* é limitado por banda de memória, isto é, o cálculo executado é simples e a taxa de leitura e escrita dos dados é o gargalo da operação. Além disso, a forma como os dados são organizados faz com que o acesso seja realizado de maneira não adequada, sendo necessário a transposição da representação para uma leitura mais eficiente.

Desta forma, é utilizado um GSL para garantir máxima eficiência no acesso e aumentar a ocupância da GPU. Cada passo da iteração é de `blockDim.x * gridDim.x` que é exatamente o número total de *threads* de uma grade, ou seja, a primeira *thread* (`threadIdx=0`) irá computar os elementos $0, 1024, 2048, ...$, a segunda *thread* os elementos $1, 1025, 2049, ...$ e assim por diante, até o último elemento da imagem.

Aliado ao GSL é utilizado um arranjo na SM para permitir o acesso desalinhado na hora de computar a conversão de espaço de cor. Assim, os dados são carregados da memória global utilizando um acesso ordenado com máximo *coalesing* e então alocados na SM, onde são utilizados de forma desordenada sem perda de desempenho. Esse procedimento garante uma eficiência máxima no acesso à memória, extraindo o máximo do hardware embarcado.

São lançados 8 blocos de 256 *threads* cada, para a execução no *kernel* de exemplo, apresentado no Código 4.8.

```
1 __global__ void convert_to_soa_grayscale_kernel(
2     unsigned char * __restrict__ output,
3     const unsigned char * const __restrict__ input,
4     int cols,
5     int rows)
6 {
7
8   /* Alocacao dos tres canais na shared memory. */
9   __shared__ uchar16 sh[3*SOA_THREADS];
10
11  /* Variaveis auxiliares para a transformacao AoS-SoA. */
12  uchar16 i0, i1, i2, g;
13
14  int i, nn, idx;
15  nn = blockDim.x*((cols*rows)/16)/blockDim.x);
16
17  /* Grid-Stride Loop. */
18  for (i = blockIdx.x * blockDim.x + threadIdx.x;
19  i < nn;
20  i += blockDim.x * gridDim.x)
21  {
22    /* Calculo do indice do primeiro elemento da iteracao. */
23    idx = 3*i - 2*threadIdx.x;
```

```
24
25      /* Carregamento dos dados da memoria global para shared memory. */
26      sh[threadIdx.x                ] = ((uchar16*)input)[idx                 ];
27      sh[threadIdx.x +   blockDim.x] = ((uchar16*)input)[idx +   blockDim.x];
28      sh[threadIdx.x + 2*blockDim.x] = ((uchar16*)input)[idx + 2*blockDim.x];
29
30      /* Sincronizacao das threads para garantir que os dados estejam prontos. */
31      __syncthreads();
32
33      i0 = sh[3*threadIdx.x    ];
34      i1 = sh[3*threadIdx.x + 1];
35      i2 = sh[3*threadIdx.x + 2];
36
37      /* Calculo vetorial dos 16 elementos que compoe a conversao para cinza. */
38      g.a = (19595*i0.a + 38470*i0.b + 7471*i0.c + 32768)/65536;
39      g.b = (19595*i0.d + 38470*i0.e + 7471*i0.f + 32768)/65536;
40      g.c = (19595*i0.g + 38470*i0.h + 7471*i0.i + 32768)/65536;
41      g.d = (19595*i0.j + 38470*i0.k + 7471*i0.l + 32768)/65536;
42      g.e = (19595*i0.m + 38470*i0.n + 7471*i0.o + 32768)/65536;
43      g.f = (19595*i0.p + 38470*i1.a + 7471*i1.b + 32768)/65536;
44      g.g = (19595*i1.c + 38470*i1.d + 7471*i1.e + 32768)/65536;
45      g.h = (19595*i1.f + 38470*i1.g + 7471*i1.h + 32768)/65536;
46      g.i = (19595*i1.i + 38470*i1.j + 7471*i1.k + 32768)/65536;
47      g.j = (19595*i1.l + 38470*i1.m + 7471*i1.n + 32768)/65536;
48      g.k = (19595*i1.o + 38470*i1.p + 7471*i2.a + 32768)/65536;
49      g.l = (19595*i2.b + 38470*i2.c + 7471*i2.d + 32768)/65536;
50      g.m = (19595*i2.e + 38470*i2.f + 7471*i2.g + 32768)/65536;
51      g.n = (19595*i2.h + 38470*i2.i + 7471*i2.j + 32768)/65536;
52      g.o = (19595*i2.k + 38470*i2.l + 7471*i2.m + 32768)/65536;
53      g.p = (19595*i2.n + 38470*i2.o + 7471*i2.p + 32768)/65536;
54
55      /* Carregamento dos pixeis computados para a memoria global. */
56      ((uchar16*)output)[i] = g;
57
58      __syncthreads();
59  }
60}
```

Código 4.8: Kernel que implementa a conversão para escala de cinza (realizando AoS to SoA).

São reservados na SM cada um dos três canais de cores, porém para maximizar a banda de leitura, são armazenados no formato de `uchar16` que é uma estrutura que alinha 16 *bytes*, ao invés de utilizar *bytes* separados. A estrutura elaborada para o arranjo `uchar16` é apresentado no Código 4.9 e é utilizado em diversos pontos do código da etapa paralela da implementação, assim como outros arranjos, tais como `int4` e `uint4`.

```
1  /* Arranjo de 16 bytes para leitura/escrita da memoria global. */
2  struct __device_builtin__ __align__(16) uchar16
3  {
4    unsigned char a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p;
5  };
6  typedef __device_builtin__ struct uchar16 uchar16;
```

Código 4.9: Estrutura `uchar16` que permite o arranjo de 16 bytes.

A primeira parte do GSL realiza a leitura dos dados da memória global (`input`) que está no padrão de um AoS e então é convertido para SoA, representado pelas variáveis `i0`, `i1` e `i2`, transformação discutida na Seção 3.3. Como esta primeira operação realiza transferências da memória global, é necessária uma sincronização das *threads*, para garantir que todas estejam alinhadas, uma vez que os dados devem estar prontos na SM para continuar a execução.

Com os dados já inseridos na memória, o que permite acesso desalinhado de baixo custo, são realizados os cálculos de conversão para escala de cinza, tal como ao executado

na implementação serial, contudo, são executados de forma vetorial, garantindo que sejam utilizados mais dados em cada uma das iterações, elevando a taxa de transmissão de dados. Portanto, são computadas as operações de conversão para cada um dos elementos do arranjo `uchar16`. Estes valores são então, novamente movidos para a memória global.

Transformação de Perspectiva (IPM)

O *kernel* para a transformação de perspectiva, possui características bem distintas da conversão para escala de cinza. Neste são computados diversos cálculos e além disso, são inseridas otimizações de instruções para reduzir a quantidade de operações no *kernel*. A implementação completa do *kernel* é apresentada no Código 4.10.

Inicialmente são carregados os valores da matriz de transformação nos registradores internos, através da variável `m_inv` e cada transformação é feita orientada por um píxel da imagem destino, ou seja, para evitar cálculos de elementos indesejáveis, são calculados segundo a imagem IPM.

Para reduzir a quantidade de píxeis analisados é utilizada a estrutura geométrica da transformada de perspectiva, que possibilita a remoção de certa de 45% dos píxeis, que devem ser calculados. Estes, são os píxeis pretos dos dois triângulos laterais gerados na imagem BEV. Como a matriz de transformação é constante, significa que essas áreas também são contantes, portanto, as retas transversais que correspondem as hipotenusas dos triângulos, foram obtidas e são utilizadas para reduzir o número de operações que devem ser realizadas. A Figura 4.10 apresenta de forma mais clara as retas que são utilizadas para essa otimização, ao passo que as condicionais das linhas 35 e 42 do Código 4.10 apresentam a divergência do fluxo mencionado.

Figura 4.10: Imagem ilustrativa da região utilizada na transformação IPM. As retas em verde limitam a região válida. Em amarelo é representado a transformação de um elemento da imagem destino.

Outra otimização de instrução é o reaproveitamento de um conjunto de valores que são mantidos contantes para uma mesma execução, como no caso dos valores de `x1c`, `y1c` e `z1c`, que são calculados e então reutilizados durante as interações, pois `i` é constante visto que depende apenas da *thread* e do bloco que está sendo executado no momento.

```
__global__ void img_inverse_perspective_kernel(
  unsigned char * output,
  const unsigned char * const __restrict__ input,
  int cols,
  int rows)
{
  /* Matriz de transformacao. */
  const float m_inv[] =
  {
    0.99894555e+00f, -0.70077033e+00f, 0.70077033e+00f,
    7.38226219e-17f, -0.24476436e+00f, 349.875707e+00f,
    3.35945421e-19f, -1.05445066e-03f, 1.00000000e+00f
  };

  float x1, y1, z1;
  int i2, j2;

  /* Obtencao do valor da coluna baseado na thread em execucao. */
  int i_min = blockIdx.x * blockDim.x;
  int i_max = i_min + blockDim.x - 1;
  int i = i_min + threadIdx.x;

  /* Retorno precoce. */
  if (i >= cols)
  {
    return;
  }

  int j = threadIdx.y * (rows/blockDim.y);
  int j_max = (threadIdx.y + 1) * (rows/blockDim.y);

  /* Condicionais para evitar calculo fora do trapezio central. */
  if (i_min < 505)
  {
    if (j >= roundf(1.4244317f * i_max + 2.2080764f))
    {
      return;
    }
  }
  else if (i_max > 814)
  {
    if (j >= roundf(-1.5406473f * i_min + 1972.7528f))
    {
      return;
    }
  }

  /* Calculo de valores contantes da transformacao. */
  float x1c = m_inv[0]*i + m_inv[2],
      y1c = m_inv[3]*i + m_inv[5],
      z1c = m_inv[6]*i + m_inv[8];

  /* Iteracao sobre as colunas para o calculo de IPM. */
  for (; j < j_max; j++)
  {
    x1 = x1c + j*m_inv[1];
    y1 = y1c + j*m_inv[4];
    z1 = z1c + j*m_inv[7];

    float z1_inv = __frcp_rn(z1);

    i2 = (int)rintf(x1*z1_inv);
    j2 = (int)rintf(y1*z1_inv);

    /* Escrita para a memoria principal do pixel transformado. */
    if (i2 >= 0 && i2 < cols && j2 >= 0 && j2 < rows)
    {
      output[j*cols + i] = input[j2*cols + i2];
    }
  }
}
```

Código 4.10: Kernel que implementa a conversão de perspectiva da imagem (IPM).

Desvanecimento Temporal

A despeito dos outros métodos já implementados, este utiliza dados pré-existentes da memória da GPU para a execução, pois um dos parâmetros de entrada do *kernel* é uma lista

das imagens já computadas (`d_imgs` no Código 4.11). Como já apresentado anteriormente, são armazenadas um conjunto de imagens anteriores para a realização da integração temporal deste arranjo.

Este *kernel*, performa operações simples de acumulação e portanto sua grande restrição é a banda de memória. Aumentar a taxa de leitura e escrita dos dados é essencial para que este tipo de *kernel* atinja eficiência máxima. Portanto, foi adotado o GSL para garantir maior ocupância na execução do *kernel*. Além do padrão de projeto do laço, o *kernel* realiza a leitura de diversos elementos em cada iteração, aumentando a taxa de leitura da banda de memória, ontudo, diferentemente das implementações anteriores, o arranjo de leitura deste *kernel*, utiliza arranjos de quatro *bytes* (`uchar4`) cada, para vetorização das operações.

Em cada iteração do GSL, são carregados os elementos da memória global e então é feita a integração. Caso ainda não existam imagens o suficiente na coleção, os valores são acumulados e tanto a imagem integrada quanto a atual, são colocadas na memória global, similarmente ao apresentado no diagrama da Figura 4.7. Contudo, quando a coleção está completa, isto é, existem `NUM_TEMPORAL` imagens no arranjo `d_imgs`, uma das imagens precisa ser substituída pela atual e também sua contribuição deve ser removida da integração, evitando a saturação dos píxeis da imagem.

```
__global__ void img_temporal_blurring_kernel(
  unsigned char * __restrict__ output,
  const unsigned char * const __restrict__ input,
  unsigned char * d_imgs,
  uint32_t cols,
  uint32_t rows,
  int num_frame)
{
  uint32_t n = cols*rows;
  uint32_t nn = n/4;

  const uchar4 zero = {0,0,0,0};
  uchar4 data, temp;

  /* Imagem mais antiga que sera removida da integracao. */
  unsigned char * __restrict__ d_img = &d_imgs[n*((num_frame-1)%NUM_TEMPORAL)];

  /* Grid-Strid Loop. */
  for (uint32_t i = blockIdx.x * blockDim.x + threadIdx.x;
  i < nn;
  i += TEMPORAL_THREADS * TEMPORAL_BLOCKS)
  {
    /* Carregamento dos dados da memoria global. */
    data = ((uchar4 *)input)[i];

    temp = (num_frame == 1) ? zero : ((uchar4 *)output)[i];

    /* Escalando o valor de entrada. */
    data.x = data.x / NUM_TEMPORAL;
    data.y = data.y / NUM_TEMPORAL;
    data.z = data.z / NUM_TEMPORAL;
    data.w = data.w / NUM_TEMPORAL;

    /* Remocao da contribuicao da imagem mais antiga na integracao. */
    if (num_frame > NUM_TEMPORAL)
    {
      uchar4 d_img_temp = ((uchar4 *) d_img)[i];

      temp.x = temp.x - d_img_temp.x;
      temp.y = temp.y - d_img_temp.y;
      temp.z = temp.z - d_img_temp.z;
      temp.w = temp.w - d_img_temp.w;
    }
```

```
44
45      /* Adicao da contribuicao da imagem atual na integracao. */
46      temp.x = temp.x + data.x;
47      temp.y = temp.y + data.y;
48      temp.z = temp.z + data.z;
49      temp.w = temp.w + data.w;
50
51      /* Carregamento do resultado para a memoria global. */
52      ((uchar4 *)output)[i] = temp;
53      ((uchar4 *)d_img)[i] = data;
54  }
```

Código 4.11: Kernel que implementa o desvanecimento temporal através da integração dos frames.

4.4.2 Extração de Características

Obtenção do Mapa de Características I^{ALT}

O filtro de limiar adaptativo realiza a extração das características com base na intensidade média dos píxeis da imagem. Na implementação proposta, foram elaborados dois *kernels* distintos para executar esta operação. O primeiro apresentado no Código 4.12 é responsável por calcular o valor da intensidade média de píxeis da imagem, enquanto o Código 4.13 apresenta o *kernel* que realiza a binarização de acordo com o limiar de luminância.

O cálculo do valor de intensidade média de luminância da imagem, consiste basicamente em extrair a média acumulada de todos os píxeis da imagem. Essa operação apresenta baixa complexidade aritmética, sendo principalmente limitada pela taxa de transferência da banda de memória.

```
1 __global__ void img_avg_kernel(
2   const unsigned char * const __restrict__ input,
3   uint32_t* sum_total,
4   uint32_t cols,
5   uint32_t rows)
6 {
7   uint4 data;
8   uint32_t sum = 0;
9   uint32_t n = cols*rows;
10  uint32_t nn = AVG_THREADS*((n/16)/AVG_THREADS);
11
12  /* Grid-Stride Loop. */
13  for (uint32_t i = blockIdx.x * blockDim.x + threadIdx.x;
14    i < nn;
15    i += AVG_THREADS * AVG_BLOCKS)
16  {
17    /* Carregamento da imagem da memoria global. */
18    data = ((uint4*)input)[i];
19
20    /* Separacao dos dados para a computacao vetorial. */
21    uint4 data2;
22    data2.x = data.x & 0x00FF00FF;
23    data2.y = data.y & 0x00FF00FF;
24    data2.z = data.z & 0x00FF00FF;
25    data2.w = data.w & 0x00FF00FF;
26
27    data.x = (data.x >> 8) & 0x00FF00FF;
28    data.y = (data.y >> 8) & 0x00FF00FF;
29    data.z = (data.z >> 8) & 0x00FF00FF;
30    data.w = (data.w >> 8) & 0x00FF00FF;
31
32    /* Somatorio vetorial dos dados. */
33    data.x += data.y;
34    data.z += data.w;
35
36    data2.x += data2.y;
```

```
37      data2.z += data2.w;
38
39      data.x += data.z;
40      data2.x += data2.z;
41
42      data.x += data2.x;
43
44      sum += (data.x & 0xFFFF) + ((data.x >> 16) & 0xFFFF);
45    }
46
47    /* Reducao da soma das threads nesse bloco para um valor unico. */
48    sum += __shfl_down_sync(0xFFFFFFFF, sum, 16);
49    sum += __shfl_down_sync(0x0000FFFF, sum,  8);
50    sum += __shfl_down_sync(0x000000FF, sum,  4);
51    sum += __shfl_down_sync(0x0000000F, sum,  2);
52    sum += __shfl_down_sync(0x00000003, sum,  1);
53
54    /* Soma atomica do valor final acumulado dos pixeis do bloco. */
55    if (threadIdx.x % 32 == 0)
56    {
57      atomicAdd(sum_total, sum);
58    }
59  }
```

Código 4.12: Kernel que implementa a extração da intensidade média de píxeis.

As principais otimizações são em relação a vetorização da operação utilizadas. Primeiramente é carregada uma quantidade elevada de dados da memória global, aumentando a taxa de transferência de dados da memória, em seguida, são utilizadas máscaras para separar os *bytes* e possibilitar uma operação de soma otimizada. Após acumular cada um dos valores, é aplicada uma série de reduções, através de primitivas SIMT. Esta operação realiza uma redução síncrona entre as *threads* de um *warp*, conforme apresentado na Figura 4.11. Desta forma, o acumulador terá a soma dos valores de cada uma das *threads* do bloco, e então basta realizar a soma dos valores desses blocos. Para garantir a consistência da soma final, é utilizado uma operação de soma atômica (`atomicAdd`) disponível na API CUDA.

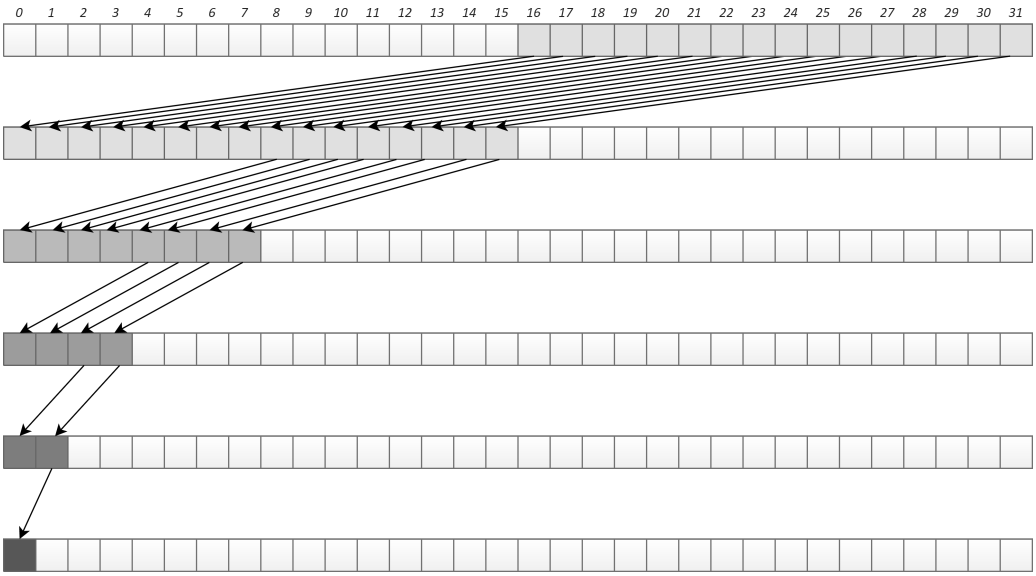


Figura 4.11: Imagem ilustrativa da operação de redução de um *warp* da GPU. O exemplo ilustra a execução de consecutivas chamadas da operação SIMT `__shfl_down_sync` disponível na API CUDA.

O *kernel* que implementa a binarização da imagem é similares aos anteriores, apesar de ter uma computação baseada em operações condicionais. Trata-se de um algoritmo limitado por banda de memória, onde as principais otimizações foram feitas para garantir alta taxa de transferência e otimizar as instruções utilizadas.

Similarmente aos outros *kernels*, um GSL é utilizado visando uma maior ocupância, contudo, diferentemente do *kernel* de extração do valor médio, onde boa parte das operações foram vetorizadas por meio de máscaras binárias. Nesta operação, foram utilizadas instruções SIMD, que garantem uma taxa de transferência superior às operações padrões da linguagem. As instruções SIMD utilizadas, foram `__vminu4`, `__vsetltu4` e `__vsetgtu4` que computam de forma paralela quatro elementos por instrução. As operações realizadas por essas instruções, são respectivamente, a obtenção do mínimo entre dois valores, atribuição 'se menor que' e atribuição 'se maior que'. Estas instruções foram combinadas para realizar o filtro de binarização baseado nos limiares, substituindo as condicionais da versão serial, conforme apresentado nas linhas 27, 28, 29 e 30 do Código 4.13.

```
1 __global__ void img_alt_map_kernel(
2   unsigned char * output,
3   const unsigned char * const __restrict__ input,
4   uint32_t cols,
5   uint32_t rows, u
6   int32_t tl,
7   uint32_t tu)
8 {
9   uint4 data;
10
11   const uint32_t n = cols*rows;
12   const uint32_t nn = ALT_THREADS*((n/16)/ALT_THREADS);
13
14   tl = __byte_perm(tl, tl, 0);
15   tu = __byte_perm(tu, tu, 0);
16
17   /* Grid-Stride Loop. */
18   for (uint32_t i = blockIdx.x * blockDim.x + threadIdx.x;
19     i < nn;
20     i += ALT_THREADS * ALT_BLOCKS)
21   {
22
23     /* Carregamento dos dados da memoria global. */
24     data = ((uint4*)input)[i];
25
26     /* Binarizacao da imagem baseada nos valores de limiar. */
27     data.x = __vminu4(__vsetltu4(tl, data.x), __vsetgtu4(tu, data.x));
28     data.y = __vminu4(__vsetltu4(tl, data.y), __vsetgtu4(tu, data.y));
29     data.z = __vminu4(__vsetltu4(tl, data.z), __vsetgtu4(tu, data.z));
30     data.w = __vminu4(__vsetltu4(tl, data.w), __vsetgtu4(tu, data.w));
31
32     /* Carregamento do dados de volta para a memoria global. */
33     ((uint4*)output)[i] = data;
34   }
35 }
```

Código 4.13: Kernel que implementa a binarização da imagem através do filtro adaptativo.

Obtenção do Mapa de Características I^{UCF}

Como discutido anteriormente nas Seções 2.2 e 4.3.3 esta etapa do método realiza uma convolução, de um núcleo de terceira ordem com a imagem, afim de evidenciar as mudanças horizontais nas intensidades dos píxeis, destacando assim, as bordas verticais (em especial as faixas) da imagem de entrada.

Em processamento de imagem, o núcleo de convolução é um produto escalar, que pode ser realizado em uma operação paralela, que é adequada para uma computação heterogênea por meio de um hardware com alta densidade e capacidade de computação simultânea, como a GPU empregada no trabalho.

Foi então proposto uma implementação paralela para solucionar esta filtragem, o Código 4.14 apresenta de forma direta o *kernel* proposto para a geração do mapa de características I^{UCF}.

```
__global__ void img_ucf_kernel(unsigned char *output, unsigned char * const __restrict__
    input, uint32_t cols,  uint32_t rows)
{
 const int f[] = {-1,-2,-1,0,0,0,1,2,1};

  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  if (x >= cols-1 || y >= rows-1 || x <= 0 || y <= 0) return;

  int dx;
  dx = (f[0] * input[(y-1)*cols + (x-1)]) +
    (f[1] * input[(y+0)*cols + (x-1)]) +
    (f[2] * input[(y+1)*cols + (x-1)]) +
    (f[3] * input[(y-1)*cols + (x+0)]) +
    (f[4] * input[(y+0)*cols + (x+0)]) +
    (f[5] * input[(y+1)*cols + (x+0)]) +
    (f[6] * input[(y-1)*cols + (x+1)]) +
    (f[7] * input[(y+0)*cols + (x+1)]) +
    (f[8] * input[(y+1)*cols + (x+1)]);

  __syncthreads();

  if(dx < 0) dx = 0;
  if(dx > 255) dx = 255;

  output[y*cols + x] = dx;
}
```

Código 4.14: Kernel que implementa a operação de extração de bordas através do filtro de correlação unilateral.

4.4.3 Estimação de Faixas

Histograma de Colunas

A operação de histograma de colunas, consiste em acumular os valores de cada linha de uma coluna, obtendo um valor único de intensidade para cada coluna da imagem. Este algoritmo possuí uma alta complexidade e elevada possibilidade de paralelização, possibilitando diversas otimizações em relação a versão serial. O Código 4.15 apresenta a implementação oti-

mizada proposta para o *kernel* de histograma de colunas, ao passo que o Código 4.16 apresenta uma implementação pré-processada auxiliar, utilizada para realizar as reduções síncronas do algoritmo.

Como o objetivo desta operação é encontrar os dois pontos de maior probabilidade de conter as faixas centrais da via, foi utilizada uma heurística simples, para obter este resultado. Uma das faixas tende a estar no lado esquerdo e outra no lado direito da região de interesse. Aproveitando esta geometria, é proposto um *kernel* com apenas dois blocos de 640 *threads* cada. Um bloco corresponde ao lado esquerdo e outro ao lado direito da imagem, que possui 1280 píxeis de largura. As 640 *threads* são lançadas em um arranjo de 160 na orientação horizontal e 4 *threads* na vertical, uma ilustração simplificada do padrão de lançamento deste *kernel* é apresentado na Figura 4.12.

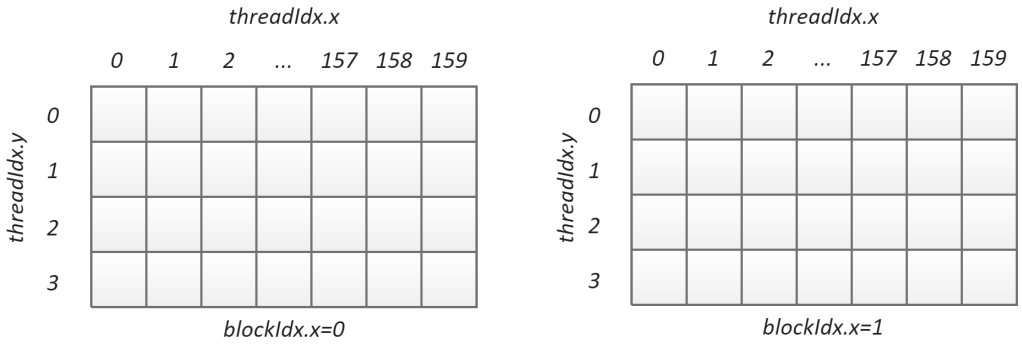


Figura 4.12: Diagrama simplificado do padrão bidimensional de lançamento do *kernel* de histograma.

Neste *kernel* são empregadas várias das otimizações já discutidas nos outros *kernels*. São implementadas para obter o máximo de desempenho nesta operação, em especial, é possível apontar o uso de SM para possibilitar eficiência no acesso da memória, o uso de GSL, uso de instruções com alta taxa de transferência e vetorização e o aproveitamento da geometria do problema.

Inicialmente, este *kernel* faz três alocações na SM: `histogram`, `pos_array` e `val_array`, que representam respectivamente o armazenamento dos elementos utilizados na computação do histograma, uma lista com as posições das colunas e outras com valores das intensidades.

Este *kernel* apresenta diversas semelhanças com o método de obtenção do valor médio das intensidades e portanto, uma abordagem semelhante de vetorização foi utilizada. São realizadas máscaras de *bits* para garantir um processo de soma entre os elementos carregados da memória, como vistos nas atribuições das variáveis `inp1_xz`, `inp2_xz`, `inp1_yw` e `inp2_yw`, que se acumulam através desta operação vetorial, para obter o valor de elemento do histograma.

```
__global__ void img_histogram_kernel(
  const unsigned char * const __restrict__ input,
  int* p1,
  int* p2,
  uint32_t cols,
  uint32_t rows)
{
  /* Alocacao da shared memory para os dados de entrada, valores e posicoes das colunas. */
  __shared__ int histogram[4*HIST_THREADS_X*HIST_THREADS_Y];
  __shared__ int pos_array[(HIST_THREADS_X*HIST_THREADS_Y + 31)/32];
  __shared__ int val_array[(HIST_THREADS_X*HIST_THREADS_Y + 31)/32];

  /* Indices para iteracao interna (warp) e global (imagem). */
  int tid = threadIdx.y * HIST_THREADS_X + threadIdx.x;
  int i = blockIdx.x * HIST_THREADS_X * HIST_THREADS_Y + tid;

  int4 hist = { 0, 0, 0, 0 };
  ((int4*)histogram)[tid] = hist;

  /* Sincronizacao para garantir integridade das memorias. */
  __syncthreads();

  /* Grid-Stride Loop. */
  for (int j = rows/2; j < rows; j += 8)
  {
    /* Carregamento dos dados de entrada. */
    int inp1 = ((int*)input)[(j     + threadIdx.y) * (cols/4)
                + HIST_THREADS_X*blockIdx.x
                + threadIdx.x];
    int inp2 = ((int*)input)[(j + 4 + threadIdx.y) * (cols/4)
                + HIST_THREADS_X*blockIdx.x
                + threadIdx.x];

    /* Vetorizacao das soma das linhas. */
    int inp1_xz = inp1 & 0x00FF00FF;
    int inp2_xz = inp2 & 0x00FF00FF;

    int inp1_yw = (inp1 >> 8) & 0x00FF00FF;
    int inp2_yw = (inp2 >> 8) & 0x00FF00FF;

    inp1_xz += inp2_xz;
    inp1_yw += inp2_yw;

    hist = ((int4*)histogram)[tid];

    hist.x += inp1_xz & 0x000001FF;
    hist.y += inp1_yw & 0x000001FF;
    hist.z += (inp1_xz >> 16) & 0x000001FF;
    hist.w += (inp1_yw >> 16) & 0x000001FF;

    ((int4*)histogram)[tid] = hist;
  }

  /* Sincronizacao para garantir integridade dos dados entre as memorias. */
  __syncthreads();

  histogram[tid] +=
    histogram[  HIST_THREADS_X*HIST_THREADS_Y + tid] +
    histogram[2*HIST_THREADS_X*HIST_THREADS_Y + tid] +
    histogram[3*HIST_THREADS_X*HIST_THREADS_Y + tid];

  /* Sincronizacao para garantir integridade dos dados entre as memorias. */
  __syncthreads();
    int pos = i;
  int val = histogram[tid];
  int pos_shfl, val_shfl;

  /* Reducao para obtencao da posicao e valor do pico (warp). */
  RED_ROUND(0xFFFFFFFF, 16);
  RED_ROUND(0x0000FFFF,  8);
  RED_ROUND(0x000000FF,  4);
  RED_ROUND(0x0000000F,  2);
  RED_ROUND(0x00000003,  1);

  /* Atualiza arranjos com os picos. */
  if ((tid & 0x1F) == 0)
  {
    pos_array[tid/32] = pos;
    val_array[tid/32] = val;
  }

  __syncthreads();

  if (tid >= 32) return;

  if (tid < (HIST_THREADS + 31)/32)
  {
    pos = pos_array[tid];
    val = val_array[tid];
```

```
90   }
91   else
92   {
93     pos = val = -1;
94   }
95
96   /* Rodada de reducao para obter o pico. */
97   RED_ROUND(0xFFFFFFFF, 16);
98   RED_ROUND(0x0000FFFF,  8);
99   RED_ROUND(0x000000FF,  4);
100  RED_ROUND(0x0000000F,  2);
101  RED_ROUND(0x00000003,  1);
102
103  /* Ultimo elemento da reducao. */
104  if (tid == 0)
105  {
106    /* Pico da esquerda. */
107    if (blockIdx.x == 0)
108    *p1 = pos;
109    /* Pico da direita. */
110    else
111    *p2 = pos;
112  }
113}
```

Código 4.15: Kernel que implementa a operação de histograma de colunas (Parte I).

Cada um dos dois blocos de 640 *threads* é lançado em um arranjo bidimensional de 160×4, ou seja, são lançadas 160 *threads* que consomem elementos no eixo horizontal da imagem (colunas) e 4 *threads* no eixo vertical (linhas), conforme apresentado na Figura 4.12. O GSL é responsável por fazer com que esses blocos percorram a imagem completa. Em cada uma dessas iterações, os elementos são acumulados e então é feita a redução síncrona de cada um dos *warps*.

Para uma melhor eficiência no código, as reduções foram programadas em uma estrutura pré-processada, conforme apresentado no Código 4.16.

```
1   #define RED_ROUND(mask, shift)                          \
2   do                                                      \
3   {                                                       \
4     pos_shfl = __shfl_down_sync(mask, pos, shift);   \
5     val_shfl = __shfl_down_sync(mask, val, shift);   \
6                                                             \
7     if ( val < val_shfl ||                            \
8     (val == val_shfl && pos > pos_shfl))              \
9     {                                                 \
10      val = val_shfl;                                 \
11      pos = pos_shfl;                                 \
12    }                                                   \
13  } while (0)
```

Código 4.16: Operação pré-processada utilizada para realizar uma rodada de reduções síncronas para o método de histograma de colunas.

São reduzidos os arranjos de posições e valores de intensidade de cada *warp* com o intuito de obter a posição com o maior valor (à direita). As reduções são feitas de forma síncrona para cada *warp*, porém, utilizando uma redução para obter o valor e a posição dos picos. Após as reduções locais é obtido o valor e posição do pico global de cada um dos blocos. Sendo que, o primeiro bloco (`blockIdx.x=0`) contém a possível região vertical da faixa da esquerda e o segundo bloco, a posição da faixa da direita.

Neste momento, é feita a escolha do tipo do algoritmo de detecção de faixas que será adotado, pois apesar de serem utilizadas apenas as faixas centrais da via, esta operação permite a expansão do problema para a obtenção todos os candidatos a faixas, armazenando outros picos para detectar as faixas laterais.

Janelas Deslizantes

O método de janelas deslizantes tem se mostrado um bom detector de faixas de trânsito, especialmente em situações onde as faixas estão em perspectiva BEV. O método consiste em analisar pequenas amostras da imagem para identificar as posições das faixas, ou seja, ao invés de percorrer a imagem inteira, detectando os pontos de interesse, é utilizada uma heurística com base nas densidades de píxeis extraídas do histograma de colunas.

A base do método é a utilização de histogramas de colunas para a determinação das regiões de maior intensidade de píxeis. Portanto, assim como apresentado no *kernel* anterior, são utilizadas reduções para a obtenção dos somatórios de cada região. Estas operações foram implementadas na função pré-processada apresentada no Código 4.17 .

```
1#define SLIDING_WINDOWS_RED_X(mask, shift)                    \
2do                                                           \
3{                                                            \
4  sum_pos_shfl = __shfl_down_sync(mask, sum_pos, shift); \
5  nonzero_shfl = __shfl_down_sync(mask, nonzero, shift); \
6                                                            \
7  sum_pos += sum_pos_shfl;                                \
8  nonzero += nonzero_shfl;                                \
9} while (0)                                                  \
```

Código 4.17: Operação pré-processada utilizada para realizar uma rodada de reduções síncronas horizontal para o método de janelas deslizantes.

Inicialmente são carregados para a SM os píxeis que são analisados nas janelas, isto garante um acesso desalinhado com alta eficiência. O padrão utilizado na SM é exatamente o escolhido para as janelas, de forma que o padrão tenha a mesma geometria, tanto na SM, quanto no sensor virtual. São utilizados dois arranjos, um para a faixa da esquerda e um para a da direita, que são separados no lançamento do *kernel* por dois blocos distintos de *threads*, sendo `blockIdx.x=0` o bloco da esquerda e `blockIdx.x=1` o da direita.

Além da separação dos blocos, os arranjos têm o mesmo padrão das janelas utilizadas pelo método, elas possuem formato bidimensional de 32 × 30 píxeis. Desta forma, o *kernel* lançado possui um arranjo de 32 *threads* horizontais (`threadIdx.x`) e 30 na orientação vertical (`threadIdx.y`), o padrão bidimensional adotado para o *kernel* de janelas deslizantes, é apresentado na Figura 4.13.

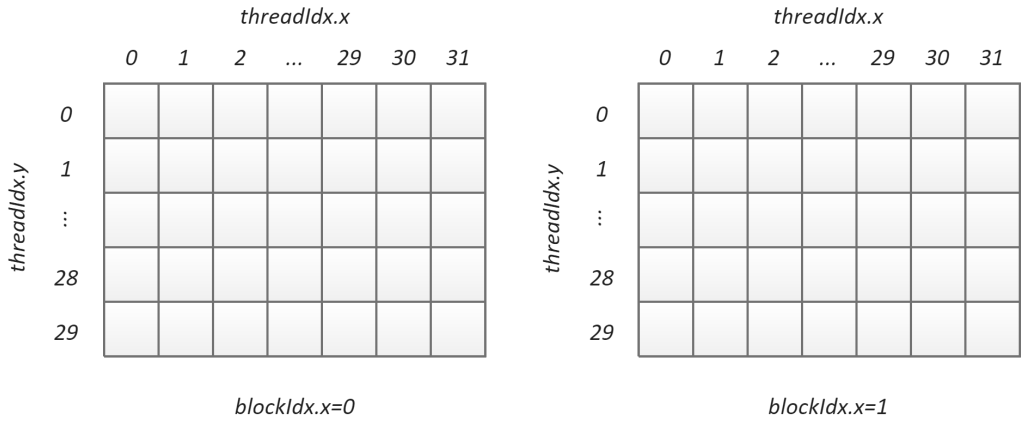


Figura 4.13: Diagrama simplificado do padrão bidimensional de lançamento do *kernel* de janelas deslizantes.

A posição da primeira janelas é determinada pelos valores obtidos no histograma de colunas do *kernel* anterior. Depois da determinação da região das janelas, uma redução vertical é implementada para acumular os valores de cada coluna, conforme apresentado no bloco da linha 34 do Código 4.18. Nesta operação, cada elemento horizontal de um bloco possui a somatória dos trinta itens da coluna. Apesar de ter um padrão de acesso desordenado, a operação é eficiente, graças ao uso da SM, que permite essa leitura sem alinhamento com baixa latência.

```
1 __global__ void img_sliding_windows_kernel( const unsigned char * const __restrict__ input,
      int *p, uint32_t cols, uint32_t rows)
2 {
3   int x, i, j;
4   int sum_pos, nonzero, sum_pos_shfl, nonzero_shfl;
5
6   /* Janelas na shared memory. */
7   __shared__ uint8_t hist[2][WINDOWS_THREADS_Y][WINDOWS_THREADS_X];
8   __shared__ int next_p[2];
9
10  /* Atualizacao do centro da janela (anterior). */
11  if (threadIdx.x == 0 && threadIdx.y == 0) next_p[blockIdx.x] = p[blockIdx.x];
12
13  __syncthreads();
14
15  /* Iteracao das janelas. */
16  for (int w = 1; w <= NUM_WINDOW; w++)
17  {
18    /* Ponto inicial da janela. */
19    x = next_p[blockIdx.x];
20
21    /* Elementos da janela dada a thread em execucao. */
22    i = x - WINDOWS_THREADS_X/2 + threadIdx.x;
23    j = (rows-HEIGHT_WINDOW(rows)*(w-1))-1 - threadIdx.y;
24
25    /* Carregamento da memoria global para shared memory. */
26    hist[blockIdx.x][threadIdx.y][threadIdx.x] = input[j * cols + i] +
27      input[(j - WINDOWS_THREADS_Y) * cols + i];
28
29    __syncthreads();
30
31    /* Reducao vertical dos elementos (threadIdx.y 0 a 30). */
32    if (threadIdx.y < 6)
33    {
34      hist[blockIdx.x][threadIdx.y][threadIdx.x] +=
35        hist[blockIdx.x][threadIdx.y +  6][threadIdx.x] +
36        hist[blockIdx.x][threadIdx.y + 12][threadIdx.x] +
37        hist[blockIdx.x][threadIdx.y + 18][threadIdx.x] +
38        hist[blockIdx.x][threadIdx.y + 24][threadIdx.x];
39    }
40
41    __syncthreads();
42
43    if (threadIdx.y == 0)
```

```
44      {
45        /* Reducao dos elementos verticais anteriores. */
46        hist[blockIdx.x][0][threadIdx.x] += hist[blockIdx.x][1][threadIdx.x] +
47          hist[blockIdx.x][2][threadIdx.x] +
48          hist[blockIdx.x][3][threadIdx.x] +
49          hist[blockIdx.x][4][threadIdx.x] +
50          hist[blockIdx.x][5][threadIdx.x];
51
52        /* Posicao zero guarda a soma dos elementos. */
53        sum_pos = (hist[blockIdx.x][0][threadIdx.x] > 0) ? i : 0;
54
55        /* Caso tenham elementos validos, incrementa o contador. */
56        nonzero = (hist[blockIdx.x][0][threadIdx.x] > 0) ? 1 : 0;
57
58        /* Reducao (horizontal) dos valores sum_pos e nonzero. */
59        SLIDING_WINDOWS_RED_X(0xFFFFFFFF, 16);
60        SLIDING_WINDOWS_RED_X(0x0000FFFF,  8);
61        SLIDING_WINDOWS_RED_X(0x000000FF,  4);
62        SLIDING_WINDOWS_RED_X(0x0000000F,  2);
63        SLIDING_WINDOWS_RED_X(0x00000003,  1);
64
65        if (threadIdx.x == 0)
66        {
67          /* Atualiza o centro da proxima janelas. */
68          if (nonzero > MIN_WHITE_PIXELS)
69          {
70            int new_x = (int)(sum_pos / nonzero);
71            if ((x * MAX_HORIZONTAL_TOLERANCE > new_x) && (x * MIN_HORIZONTAL_TOLERANCE <
                  new_x))
72              x = new_x;
73          }
74          next_p[blockIdx.x] = p[2*w+blockIdx.x] = x;
75        }
76      }
77      __syncthreads();
78    }
79  }
```

Código 4.18: Kernel que implementa a operação de janelas deslizantes

Para garantir a integridade dos dados, na primeira *thread* vertical, são acumulados os valores da última redução (linhas 46 à 50 do Código 4.18), então são atualizados os valores dos somatórios de cada uma das colunas na variável `sum_pos` e o contador de elementos não nulos `nonzero`.

Essas duas variáveis são então reduzidas de forma síncrona, através da operação pré-processada apresentada no Código 4.17. Desta forma, o resultado em cada uma dessas variáveis contém a redução do bloco completo. Por fim, como a redução armazena os resultados na primeira *thread* do bloco, ela é utilizada para validar a posição obtida pelo histograma da janela. Essa posição é obtida pela razão entre as coordenadas e a quantidade de zeros, conforme apresentado no bloco da linha 65 do Código 4.18. A saída deste método, são dois conjuntos de pontos que representam as faixas detectadas pelo algoritmo de janelas deslizantes armazenados no ponteiro `p`.

4.4.4 Conclusões sobre a Abordagem Paralela (GPU)

O presente capítulo expõe de forma minuciosa cada uma das principais otimizações propostas para a solução do problema de detecções de faixas centrais em um ambiente embar-

cado, por meio das implementações heterogêneas, foram abordas diversas práticas de otimização como, otimizações para ganho de banda de memória, transposições nos padrões de acesso aos dados e o uso efetivo das diferentes memórias apresentadas. São discutidas otimizações de instruções, por meio de primitivas SIMT e SIMD, bem como implementações que utilizam os arranjos dos dados para obter uma maior eficiência na execução das instruções. Além disso, abordou-se, do ponto de vista prático, diversas formas de otimizar e reduzir as latências do sistema, tanto em relação as transferências de dados, quanto o preenchimento deste tempo de trânsito com instruções, mascarando as latências, além de apresentar diversos padrões que aumentam a ocupância e eficiência da implementação.

Capítulo 5

Resultados Experimentais

O presente capítulo é dividido em duas principais partes, a primeira diz respeito aos resultados de uma implementação serial intermediária, que serviu de base para a elaboração da versão heterogênea final proposta no trabalho. Desta forma, a Seção 5.1 apresenta os resultados obtidos para a versão serial produzida durante a execução do projeto. Estes resultados foram compilados e publicados na forma de um artigo científico e é apresentado no Apêndice A. Em seguida, na Seção 5.2 são discutidos os resultados da implementação heterogênea, bem como as comparações qualitativas e quantitativas de cada um dos métodos implementados em diferentes cenários e métricas. Estes resultados são obtidos através das implementações otimizadas discutidas na Seção 4.4, para a elaboração dos experimentos da Seção 5.2 foram realizadas três implementações distintas, uma serial, uma híbrida e a versão paralela final e os testes foram realizados sobre a base de dados TuSimple.

5.1 Experimento Serial

O objetivo principal deste experimento é a busca por um método robusto para a detecção de faixas com custo computacional baixo, compatível com aplicações embarcadas. Diferentemente do software apresentado no Capítulo 4 este primeiro experimento não implementa um sistema completo, com todas as etapas descritas como aquisição de imagens, pós-processamento, registro e telemetria de dados. Além disto, como este foi um experimento intermediário do projeto, alguns dos algoritmos são substituídos na versão final, apresentada na Seção 5.2.

Assim, o método deste experimento é responsável apenas pela tarefa de detecção de faixas centrais, sendo composto por três etapas (similar ao apresentado na abordagem serial do Capítulo 4): segmentação de faixas através da combinação de filtros de limiar, geração de uma região de interesse em perspectiva aérea e detecção das faixas pelo método de janelas deslizantes. Para avaliação e validação do experimento proposto, o método foi implementado no mesmo sistema embarcado descrito na Seção 4.1 e imagens de um vídeo pré-capturado de uma base de dados foram utilizadas. Além disso, são analisadas métricas de acurácia e tempo de execução para diferentes cenários de vias, bem como para resoluções distintas de imagem.

A principal característica deste método é a etapa de segmentação de faixas, ela faz a combinação de quatro filtros de limiar, sendo dois baseados na magnitude e direção dos gradientes obtidos pelo operador de Sobel (TU et al., 2013; OZGUNALP et al., 2016; SONG et al., 2017) e outros dois relacionados a filtros de canais de cores (um do canal R do espaço RGB e outro do canal S do espaço HLS (YINGHUA HE; HONG WANG; BO ZHANG, 2004; TSUNG-YING SUN; SHANG-JENG TSAI; CHAN, 2006; MUTHALAGU; BOLIMERA; KALAICHELVI, 2020)).

Além disso, foram exploradas heurísticas para melhorar a detecção do método de janelas deslizantes, garantindo uma maior robustez quanto a variação horizontal das janelas em situações de faixas descontínuas. O Apêndice A apresenta os detalhes teóricos e metodológicos deste experimento na forma de um artigo científico.

Este experimento foi realizado com o objetivo de obter uma validação do método proposto e identificar seus principais problemas, podendo assim, realizar a correção e modificação destas etapas para a obtenção de um método mais robusto e eficiente. Desta forma, foram elaborados três diferentes cenários de resolução de imagem e de sinalizações de vias, afim de obter um visão ampla do método.

5.1.1 Métricas e Parâmetros de Desempenho

Os dados utilizados como entrada do método possuem uma resolução de 1280×720 píxeis e são adquiridos pelo sistema embarcado, simulando uma aquisição real. Além da resolução padrão da base de dados, foram geradas outras duas bases redimensionando os vídeos para as resoluções de 720×405 e 480×270 píxeis, buscando obter mais cenários de operação para avaliar o desempenho do sistema proposto.

Para estabelecer as análises de qualidade do método proposto são utilizadas duas métricas padrão, a taxa de falsos positivos (TFP) e o erro médio relativo (EMR) da detecção.

A TFP é a fração de quadros do total que não foram devidamente identificados. Estes são quadros onde as faixas não apresentam coerência entre si ou coerência com a realidade, ao passo que o EMR é uma figura de mérito do erro.

A Equação 5.1 define formalmente a métrica TFP como a razão entre a quantidade de falsas detecções (NF) e quantidade de quadros detectados (ND)

$$\mathrm{TFP} = \frac{\mathrm{NF}}{\mathrm{ND}}. \tag{5.1}$$

Os valores de ND são obtidos comparando, ainda em tempo de execução, as faixas detectadas com as faixas reais, em seguida, as falsas detecções são avaliadas de maneira manual para garantir a correta estimação deste parâmetro.

O erro médio relativo é definido pela Equação 5.3, que é a média dos erros quadráticos médios entre o conjunto de pontos detectados pelo método proposto (p_i) e os pontos do conjunto real ($\hat{p}_i$) em um determinado cenário. Além disso, a medida é normalizada pela dimensão horizontal (W) das imagens, garantindo uma medida adimensional para a comparação entre as diferentes resoluções

$$\mathrm{EQM} = \frac{1}{N}\sum_{i=1}^{N}(\hat{p}_i - p_i)^2, \tag{5.2}$$

$$\mathrm{EMR} = \frac{1}{M}\sum_{j=1}^{M}\frac{\mathrm{EQM}}{W}. \tag{5.3}$$

Os pontos reais das faixas foram extraídos manualmente a partir de quadros sorteados aleatoriamente da base de dados.

5.1.2 Análises e Discussões

O algoritmo proposto deve operar detectando as duas faixas principais da via, onde o veículo se encontra, sendo estas contínuas ou descontínuas. Para representar os resultados obtidos, as faixas são ilustradas em uma etapa de pós-processamento, utilizada apenas para gerar faixas na cor amarela, como forma de visualização do resultado obtido, conforme exemplos das Figuras 5.1(d), 5.1(e), 5.1(f), 5.1(j), 5.1(k) e 5.1(l).

Esta representação das faixas é obtida através do ajuste polinomial de segunda ordem dos pontos detectados pelo método. Este ajuste, busca obter as curvaturas da via, permitindo a

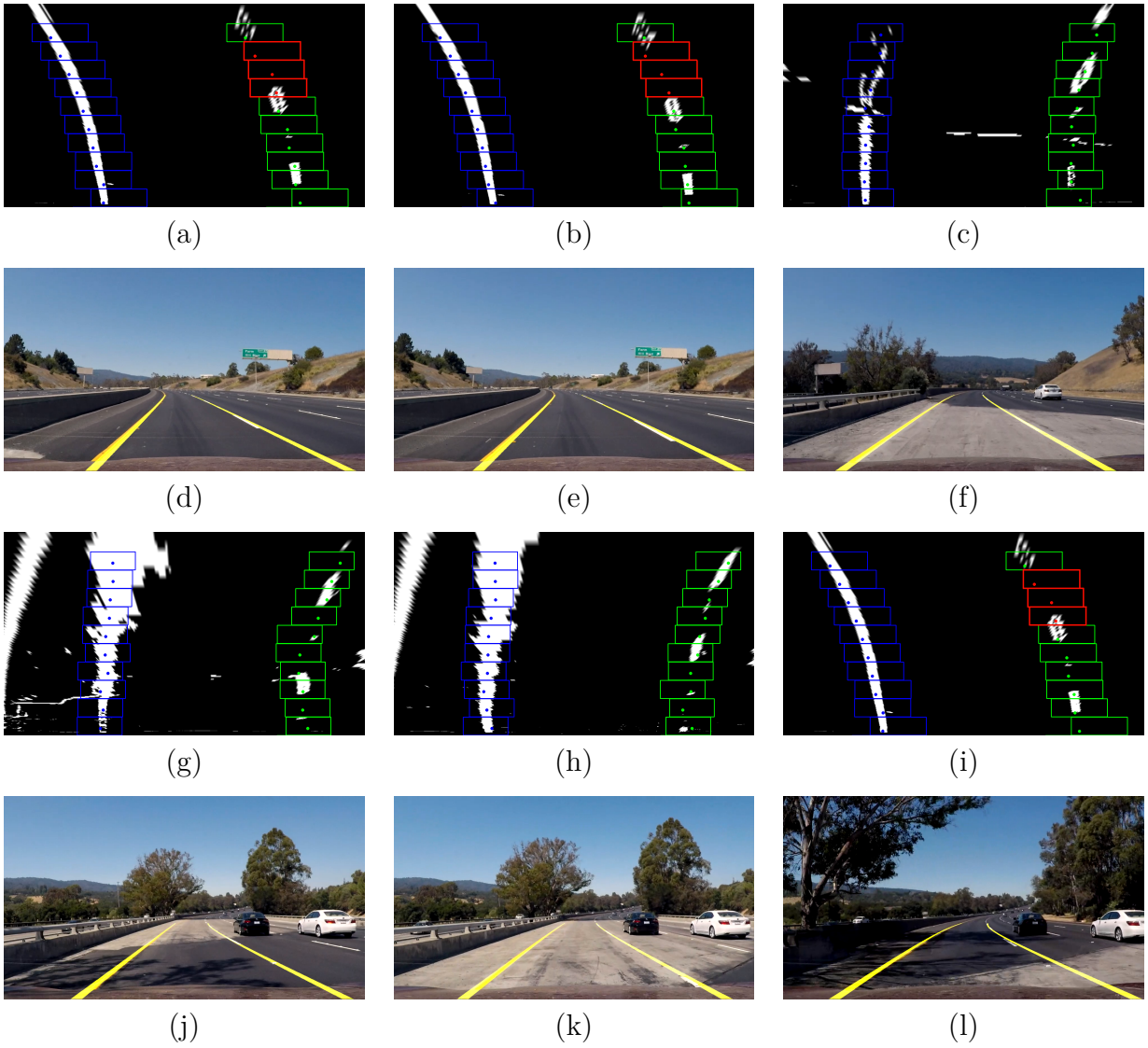


Figura 5.1: Resultados típicos obtidos através do algoritmo proposto. As imagens (a), (b), (c), (g), (h), e (i) apresentam a etapa de detecção das faixas, ilustrando os sensores virtuais através dos retângulos. O conjunto de pontos descreve a posição dos segmentos de faixa identificados nestes sensores. Em (d), (e), (f), (j), (k) e (l) são apresentadas as representações das faixas detectadas através do ajuste polinomial dos dois conjuntos de pontos obtidos na detecção das faixas da esquerda e da direita.

detecção de distâncias mais longas e a determinação dos contornos da estrada, diferenciando-se dos métodos que adotam o modelo de faixa linear.

As Figuras 5.1(a), 5.1(b) e 5.1(c) são exemplos de detecção de curvatura. Essas são representações do processo de detecção das faixas, sendo apresentadas na perspectiva aérea. Os retângulos em azul e verde representam os sensores virtuais de cada uma das faixas, ao passo que os pequenos círculos internos aos sensores formam o conjunto de pontos detectados na faixa.

Dois problemas típicos que afetam a detecção de faixas de trânsito são: a variação da luminosidade em trajetos percorridos pelo veículo, seja por sombras ou por reflexos da luz,

conforme exemplificado nas Figuras 5.1(j) e 5.1(l) e a variação da coloração do asfalto em regiões da via de trânsito, que podem ser vistas nas Figuras 5.1(f) e 5.1(k).

As Figuras 5.1(g) e 5.1(h) permitem observar o impacto no processo de detecção gerado pela presença de sombras e variações de luminosidade, enquanto nas Figuras 5.1(c) e 5.1(i) é observada a influência da mudança na coloração do asfalto.

A despeito das interferências visualizadas no processo de segmentação da imagem, a estimação das posições iniciais através do histograma de intensidade e a utilização de informações de amostras de quadros anteriores em conjunto com os sensores virtuais, garantem robustez ao método, alcançando os resultados satisfatórios como apresentados nas Figuras 5.1(j), 5.1(k) e 5.1(l).

Uma das principais fontes de imprecisão no sistema, são distorções geradas pela transformação inversa de perspectiva em pontos mais distantes nas faixas descontínuas. Contudo, este efeito é mitigado pela reutilização de informações de quadros anteriores e da relação de distância entre este ponto e seu correspondente na outra faixa. Isto, porque não há variações significativas na curvatura das faixas e na posição relativa do veículo entre dois quadros consecutivos. Este efeito, pode ser melhor visualizado em vermelho nos dois quadros consecutivos apresentados nas Figuras 5.1(a) e 5.1(b), onde mesmo na ausência de píxeis brancos é possível seguir a tendência da curvatura.

São utilizados três diferentes cenários de resolução de imagem para obter uma análise mais ampla sobre o método. O vídeo utilizado como entrada do sistema, possui um total de 1271 quadros, sendo todos processados pelo método. Tanto a quantidade de detecções realizadas (e consequentemente a TFP) quanto a EMR destas detecções são calculadas para cada um destes quadros e para cada uma das três resoluções distintas.

Os resultados quantitativos típicos coletados durante a execução da experiência estão disponíveis na Tabela 5.1, onde R_1, R_2 e R_3 representam respectivamente as resoluções 1280×720, 720×405 e 480×270 píxeis.

Tabela 5.1: Desempenho obtido sobre a base de dados para diferentes resoluções.

	Medidas		
Índice	R_1	R_2	R_3
Quadros Analisados	1271	1271	1271
Detecções	1234	1202	1129
TFP (%)	2,14	5,43	10,46
EMR (10^{-3})	8,05	11,36	12,90

Analisando a porcentagem de TFP para cada uma das resoluções é possível identificar seu impacto na quantidade de detecções. Enquanto na resolução R_1 o algoritmo apresenta uma alta taxa de detecção, falhando em apenas 2,14% dos quadros totais, na resolução mais baixa, essa perda foi cinco vezes maior.

Ao analisar a qualidade das detecções realizadas pelo método, é possível observar que não somente a quantidade de quadros detectados é maior em altas resoluções, como também os erros dessas detecções são menores.

Para avaliar a qualidade do método de detecção em diferentes cenários de operação, a determinação do EMR é separada em situações específicas: a detecção de faixas contínuas e descontínuas e vias com e sem curvaturas. Estas informações exprimem de forma mais clara e objetiva as métricas de qualidade de detecção do método proposto e estão sumarizadas na Tabela 5.2.

Na Tabela 5.2 é possível notar que não somente a resolução impacta na detecção, mas também o tipo de sinalização da faixa e sua curvatura. A melhor precisão na detecção ocorre em cenários onde as faixas são contínuas e com raios de curvatura longos ou retilíneas.

Tabela 5.2: EMR específico em diferentes situações para as diferentes resoluções ($\times 10^{-3}$).

	Contínua		**Descontínua**	
	Reta	**Curva**	**Reta**	**Curva**
R_1	3,09	11,89	4,80	12,43
R_2	3,65	13,46	9,28	19,09
R_3	5,33	13,86	12,8	19,61

Dado que este experimento propõe a implementação de um algoritmo de detecção de faixas, voltado para aplicações reais em uma plataforma embarcada, é necessário avaliar os tempos de execução do método proposto para realizar uma análise de viabilidade de operação.

Deste modo, são examinados os tempos de execução de cada uma das etapas do método proposto, no ambiente embarcado, com o intuito de levantar os custos destas operações e expor as etapas com maior impacto computacional. Estes dados estão dispostos na Tabela 5.3. Assim como na análise anterior, foram tomadas as medidas individualmente em cada cenário de resolução.

Analisando os tempos para cada operação do método, é evidente que o maior tempo de processamento é despendido nas tarefas de filtragem da etapa de segmentação de imagem, ou seja, as operações de visão computacional são as operações mais custosas no método proposto. Contudo, na experiência realizada, estas operações são desempenhadas por uma biblioteca

Tabela 5.3: Porcentagem do tempo de processamento para cada uma das operações em diferentes resoluções.

	Tempo (%)		
Operação	R_1	R_2	R_3
FL de Canais	17,06	20,63	35,60
FL de Gradiente	46,14	52,53	20,62
Região de Interesse	20,91	15,86	32,20
Histograma Inicial	7,09	4,64	4,80
Slidding Windows	8,60	5,84	5,36
Ajuste Polinomial	0,18	0,50	1,42
Total (ms)	72,913	26,498	11,555

de uso geral, não otimizada para aplicações de alto desempenho e aplicações embarcadas. Além disso, não são utilizados todos os recursos disponíveis na plataforma embarcada, como o processamento paralelo ou as operações inerentes ao processador gráfico.

Apesar de ser uma implementação intermediária do projeto proposto, o método descrito no Apêndice A apresenta bons índices de detecção e um tempo de execução coerente com a ideia do projeto, sendo executado em tempo real em algumas condições.

5.1.3 Conclusões do Experimento Serial

De forma geral, o experimento proposto evidencia os dois principais problemas do método. O primeiro diz respeito à variação na qualidade da detecção conforme o tipo de sinalização das faixas, isto é, apresentou uma qualidade mais elevada na detecção de faixas contínuas e retas. Além disso, as operações de segmentação de imagem (e geração dos mapas de características) se mostra extremamente custosa para o processador serial, sendo responsável por cerca de 50% do tempo de processamento do método.

Estes dois fatores, são avaliados e as modificações necessárias são implementadas em um segundo momento, no experimento paralelo descrito na Seção 5.2. São elaborados outros algoritmos para realizar os mapas de características, buscando melhorar a robustez da detecção à variação das sinalizações das faixas e também algoritmos que possuem uma complexidade computacional menor (Seção 2.2). Uma das alterações adicionadas é a integração do histórico de imagens através do desvanecimento temporal (Seção 2.1.3) e a geração de mapas de características pela combinação de métodos mais simples (Seção 2.2).

Além disso, a adoção do método de janelas deslizantes em conjunto com o ajuste polinomial de segunda ordem, apresentam resultados interessantes para aplicações embarcadas, aliando o compromisso de baixos erros relativos na detecção e o baixo tempo de execução,

mesmo na implementação serial. Desta forma, estes métodos são promissores e são também implementados em suas versões paralelas na versão final.

5.2 Experimento Paralelo

Na Seção 5.1 são abordados os resultados intermediários do projeto, tanto do ponto de vista do algoritmo, quanto do método de implementação. A presente seção expõe de forma direta as metodologias adotadas na avaliação do algoritmo proposto para a detecção de faixas centrais em um sistema heterogêneo. A implementação do algoritmo discutido nessa seção é apresentada na Seção 4.4.

O Apêndice B apresenta os detalhes teóricos e metodológicos deste experimento na forma de um artigo científico submetido a um periódico.

A primeira parte da seção apresenta de forma simplificada o ambiente de testes estabelecido para o experimento, bem como a base de dados utilizada. Em seguida, são realizadas diversas análises sobre os resultados coletadas e também as devidas comparações com outros métodos.

5.2.1 Ambiente de Teste e a Base de Dados TuSimple

O ambiente de teste proposto para a execução do experimento de implementação paralelo é, assim como no experimento apresentado na Seção 5.1, o hardware embarcado NVIDIA Jetson Nano. Não foram realizadas modificações no hardware entre as duas versões de implementações. A execução dos métodos de visão computacional e processamento de imagem são feitos inteiramente na GPU, com os métodos apresentados na Seção 4.4.

Como mencionado anteriormente, foi utilizada a linguagem de programação C++ na versão 7.5.0 em conjunto com a API CUDA 10.2, que forneceu acesso aos recursos da GPU e possibilitou o desenvolvimento e execução dos algoritmos propostos.

O experimento tem como entrada do sistema, um conjunto de imagens pré-capturadas, a base de dados em questão é a TuSimple (TUSIMPLE, 2018). Este conjunto de imagens possui um total de 72520 quadros capturados, os quadros foram extraídos de 3626 vídeo clipes, de um segundo cada, à uma taxa de 20 fps, ou seja, cada segundo de gravação possui exatos vinte quadros ($3626 \times 20 = 72520$). A Figura 5.2 apresenta uma coleção aleatória de quadros desta base de dados, representando os diferentes cenários.

Figura 5.2: Conjunto aleatório de imagens presentes na base de dados TuSimple utilizados na avaliação do método proposto.

Os clipes não são sequenciais, existe variação abrupta de cenário e de condições do ambiente, como mudança de luminosidade, tipo de faixa e posição do veículo na via. Esse tipo de mudança abrupta insere uma complexidade mais elevada na detecção, principalmente para o método proposto que possui, em uma das operações, uma análise temporal da via. Contudo, avaliar o método neste tipo de cenário, propicia uma validação sobre estresse das características do método.

As imagens são obtidas de uma câmera centralizada no painel frontal do veículo e possuem uma resolução de 1280×720 píxeis. Cada cenário possui diferentes condições climáticas e períodos do dia, bem como, diferentes tipos de vias, condições de asfalto e qualidade de marcações das faixas.

Cada uma das filmagens possui, em seu último quadro, uma imagem de anotação, isto significa que esta imagem funciona como um gabarito da detecção das faixas do quadro em específico. Desta forma, é possível realizar a detecção das faixas neste quadro e então comparar os resultados do método proposto e o valor confiável.

O Código 5.1 apresenta um exemplo do gabarito fornecido pela base de dados, explicitando como são as anotações dos últimos quadros de cada um dos clipes da base de dados.

```
{
"lanes": [
[-2, -2, -2, -2, 719, 734, 748, 762, 777, 791, 805, 820, 834, 848, 863, 877, 891, 906, 920,
      934, 949, 963, 978, 992, 1006, 1021, 1035, 1049, 1064, 1078, 1092, 1107, 1121, 1135, 1
     150, 1164, 1178, 1193, 1207, 1221, 1236, 1250, 1265, -2, -2, -2, -2, -2],
[-2, -2, -2, -2, -2, 532, 503, 474, 445, 416, 387, 358, 329, 300, 271, 241, 212, 183, 154,
     125, 96, 67, 38, 9, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2,
      -2, -2, -2, -2, -2, -2, -2],
],
"h_samples": [240, 250, 260, 270, 280, 290, 300, 310, 320, 330, 340, 350, 360, 370, 380, 39
     0, 400, 410, 420, 430, 440, 450, 460, 470, 480, 490, 500, 510, 520, 530, 540, 550, 560,
      570, 580, 590, 600, 610, 620, 630, 640, 650, 660, 670, 680, 690, 700, 710],
"raw_file": "nome_da_imagem"
}
```

Código 5.1: Exemplo da estrutura JSON de uma anotação de um quadro na base de dados TuSimple.

Cada um dos vigésimos quadros de cada clipe, possui uma anotação como a apresentada no Código 5.1. A anotação é um arranjo JSON que possui três campos: `lanes`, `h_samples` e `raw_file`. O primeiro é uma lista, onde cada item é uma lista das coordenadas horizontais das faixas, sendo que o valor `-2` representa que aquela posição vertical não possui faixa. A chave `h_samples` apresenta as posições verticais amostradas, e por fim, a chave `raw_file` apresenta no nome da imagem anotada da base de dados.

Descrição dos Testes

A análise do método proposto é baseada em dois principais pontos, o primeiro diz respeito a qualidade das detecções realizadas pelo método, ao passo que o segundo analisa o desempenho computacional das otimizações heterogêneas propostas.

Como mencionado anteriormente, a análise da qualidade das detecções é feita utilizando métricas baseadas nas anotações da base de dados TuSimple. Contudo, para analisar quais foram os impactos das otimizações propostas, são implementadas três versões do mesmo método, sendo:

- uma versão sequencial executada exclusivamente na CPU, sem uso de bibliotecas externas para processamento das imagens;
- uma versão intermediária, que faz uso de uma biblioteca de processamento de imagem (OpenCV) compilada para utilizar os recursos da GPU/CUDA;
- uma versão com as otimizações propostas, com execução heterogênea implementada em CUDA, conforme descrito ao longo da Seção 4.4.

Todos os métodos do algoritmo de detecção de faixas centrais propostos neste trabalho, são implementados nas três versões. Desta forma, é possível observar quais são os ganhos das otimizações apresentadas na Seção 4.4 em relação a versão serial do método, salientando os ganhos da computação heterogênea para a solução desta classe de problemas em sistemas embarcados. Ao passo que a comparação com a implementação híbrida, que também faz uso de computação heterogênea, salienta a necessidade da aplicação dos conceitos de otimização discutidos na Seção 4.4.

Para garantir que os métodos executem as mesmas operações, foi estabelecido um conjunto de testes unitários, utilizando a ferramenta GoogleTest (GOOGLE, 2021). Os testes em questão, têm como principal intuito, garantir que cada imagem processada pelas três versões dos métodos possuam as mesmas respostas. Neste sentido, são feitas análises píxel-a-píxel das imagens de saída, garantindo a integridade dos métodos. Como alguns métodos não possuem como saída uma imagem, o mesmo procedimento foi executado, afim de comparar outros tipos de dados de saída, como: vetores, inteiros, etc. Além disso, é possível também extrair os tempos de execução utilizando as primitivas disponíveis na linguagem C++. Estes testes foram vastamente utilizados durante a elaboração dos métodos e as análises, pois garantem a confiabilidade da execução das diferentes implementações.

5.2.2 Análise Qualitativa do Experimento

O objetivo da análise qualitativa do experimento é justamente entender como o método se comporta em diferentes cenários e salientar cada uma das condições e respostas observáveis. Para elaborar esta análise, é necessário extrair alguns quadros da execução do algoritmo na base de dados. Alguns dos quadros selecionados estão dispostos nos seis arranjos de imagens apresentados na Figura 5.3.

Cada arranjo possui quatro sub-figuras que representam diferentes etapas do método, sendo, a imagem de entrada (superior à esquerda), a execução do método de janelas deslizantes (superior à direita), o ajuste das faixas detectadas em perspectiva BEV (inferior à esquerda) e a saída do método representando a área da região detectada na perspectiva original (inferior à direita). Além disso, cada um dos conjuntos de imagem retratam um momento distinto da base de dados, variando o ambiente, a condição da via, a qualidade das marcações, etc.

As Figuras 5.3a e 5.3b apresentam a execução do método em diferentes colorações de asfalto, não apenas entre si, mas também asfaltos com uma variação interna de cor. Na primeira

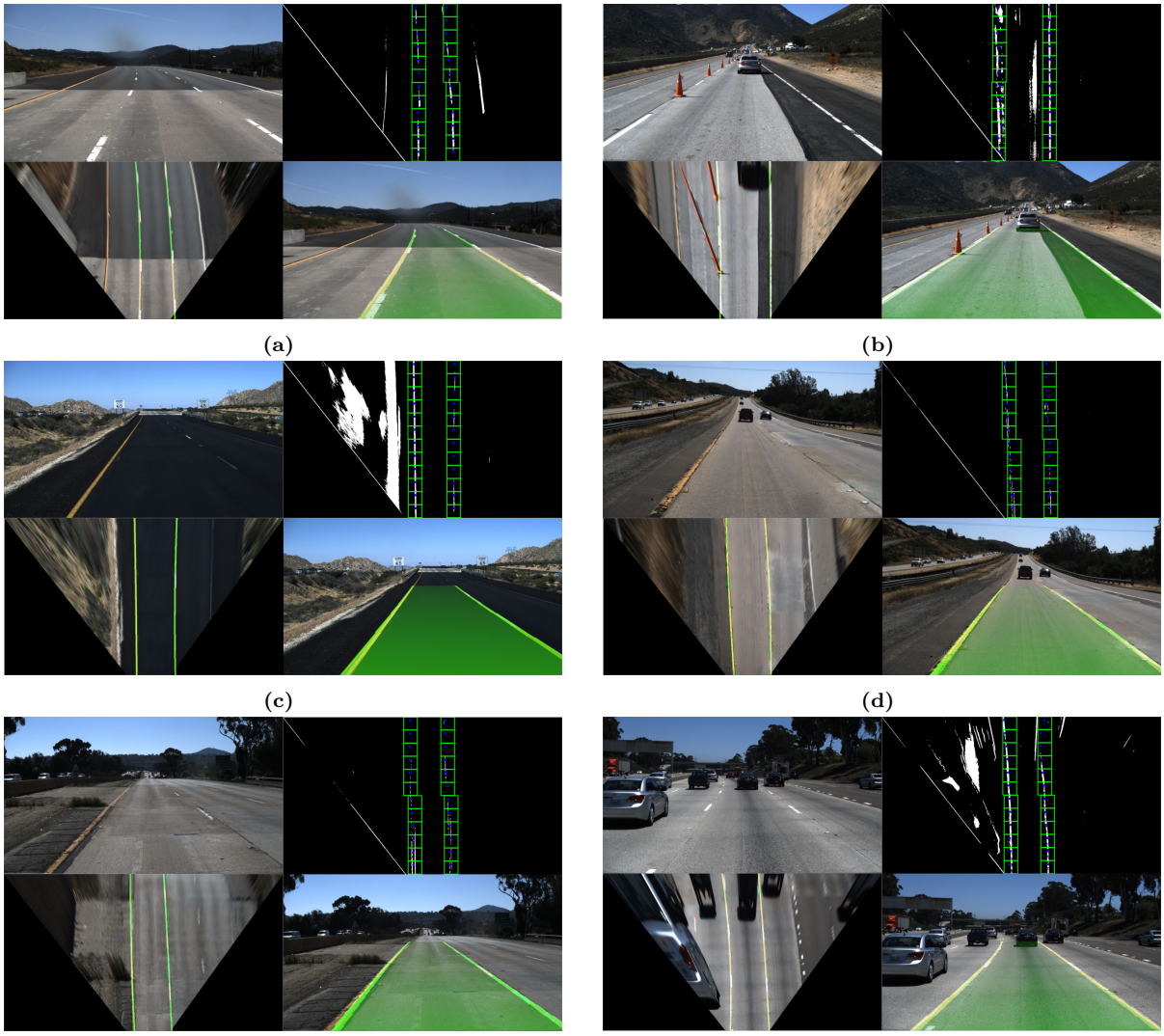


Figura 5.3: Exemplo de imagens em diferentes cenários para estágios distintos do método proposto.

imagem há um seção do asfalto claro e outra escura em uma variação vertical enquanto, na segunda há uma variação horizontal da coloração do asfalto, além de possuir cones sinalizadores em uma das faixas. É possível notar, que mesmo sobre estas variações internas, o método consegue realizar a detecção com clareza, sem a geração de pontos espúrios na detecção. O principal motivo para a qualidade da detecção neste tipo de cenário, são os filtros utilizados na etapa de extração de características, que se ajustam para diferentes colorações e também utilizando outro recurso que é a diferença de pigmentação entre o asfalto e as marcações. É visível como a pigmentação do asfalto mais escuro, reduziu o valor de luminância média da imagem, fazendo com que os limiares de filtragem, permitissem que mais pontos fossem detectados.

A Figura 5.3c apresenta um asfalto com pigmentação extremamente escura e faixas de trânsito com colorações e marcações distintas. Assim, como mencionado nos quadros anteriores, o algoritmo performa bem sobre este tipo de cenário, devido ao limiar adaptativo da geração dos mapas de características.

Exemplos de cenários com vias degradadas e com marcações desgastadas são apresentados nas Figuras 5.3d e 5.3e. Em ambas a qualidade do asfalto é baixa e as marcações são praticamente inapropriadas. Contudo, apesar do desgaste e da degradação, o método realiza a detecção com qualidade. Um ponto de interesse é a faixa direita da Figura 5.3d, onde a marcação é praticamente inexistente. Contudo, os filtros de extração de características conseguem obter pequenos fragmentos da posição das faixas, graças a integração temporal dos quadros, que aprimora a qualidade das marcações. Não obstante, a transformação de perspectiva, aliada ao método de janelas deslizantes, faz com que mesmo não obtendo uma faixa contínua na etapa de filtragem, seja possível determinar com qualidade a posição das faixas.

Por fim, a Figura 5.3f apresenta um cenário mais urbano, com a presença de veículos e também de curvatura da via, similar a curvatura da Figura 5.3a. A detecção neste tipo de cenário é mais delicada, os veículos tendem a gerar interferências na extração de características, em especial no detector de bordas. Além disso, as janelas deslizantes se deslocam horizontalmente com uma intensidade maior, devido as curvaturas. Porém, a imagem BEV reduz grande parte da complexidade da detecção deste tipo de cenário e o método performa com qualidade neste tipo de detecção.

De forma geral, é possível ver que o método proposto apresenta respostas precisas e satisfatórias para a detecção nos diversos ambiente apresentados, superando a qualidade das marcações, os tipos de faixas e a pigmentação dos asfaltos.

Para uma visualização mais real e com mais detalhes, é elaborado o vídeo Silva (2021) que ilustra a execução do algoritmo proposto. O vídeo apresenta a performance do método na base de dados TuSimple 0601, um subconjunto da base de dados. São utilizados 8200 quadros para a elaboração do vídeo, e como já mencionado, os clipes não são consecutivos o que insere uma complexidade maior para a detecção das faixas, pois o método faz uso de informações temporais para a detecção. O vídeo é separado em quatro quadrantes, o canto superior esquerdo apresenta o quadro original. A imagem do canto inferior esquerdo é o resultado obtido na etapa de extração de característica com o método de janelas deslizantes, representado pelos retângulos verdes e os pontos detectados pelos círculos azuis. O último quadrante apresenta

uma sobreposição da imagem original e os pontos estimados pelo método proposto. A Figura 5.4 apresenta um conjunto com alguns quadros extraídos do vídeo.

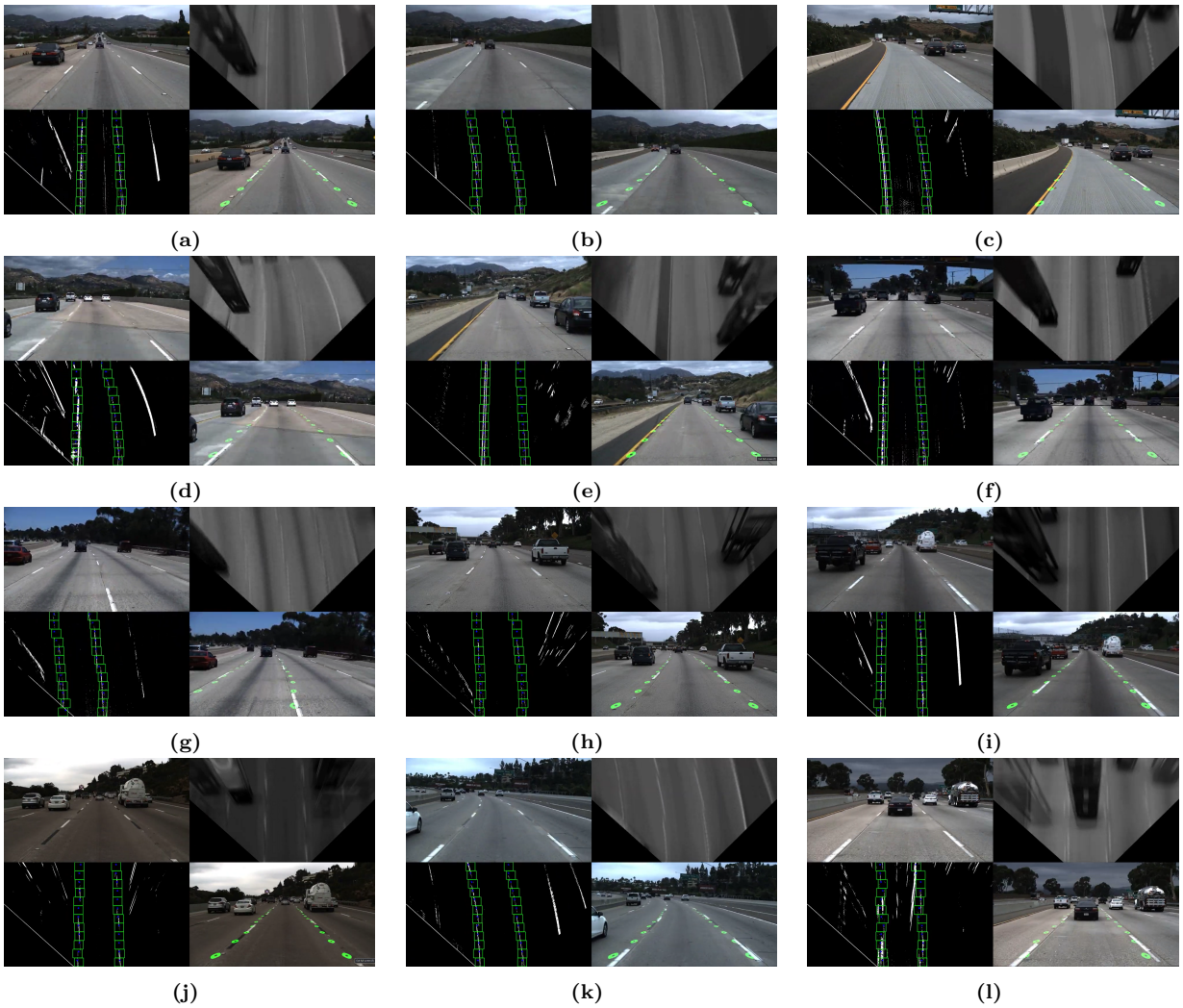


Figura 5.4: Exemplo de imagens em diferentes cenários para estágios distintos do método proposto.

5.2.3 Performance do Algoritmo

Para a avaliação do desempenho do método proposto e os ganhos em relação a implementação tradicional, são implementadas três versões distintas do mesmo algoritmo: *i)* uma versão inicial executada de forma serial no processador, que não utiliza bibliotecas de processamento de imagens; *ii)* uma implementação que faz uso de uma versão compilada da biblioteca OpenCV para GPU de forma a melhorar a performance e *iii)* uma implementação que possui execução somente no chip gráfico, utilizando todas as otimizações propostas para o método durante o Capítulo 4.

Com o objetivo de que todos os métodos executem a mesma operação, foram implementadas rotinas de testes que avaliam as imagens geradas por cada um dos métodos e faz uma verificação píxel-a-píxel para garantir que todas tenham o mesmo resultado.

A Tabela 5.4 apresenta as principais operações do algoritmo proposto e os tempos de execução dos mesmos no hardware NVIDIA Jetson Nano. A coluna CUDA apresenta os tempos de execução da implementação heterogênea otimizada, proposta para neste trabalho. A coluna OpenCV, expõe os tempos de execução da versão implementada com a biblioteca de processamento de imagem, que faz uso dos recursos do chip gráfico. Por fim, na coluna CPU é apresentada a versão padrão do método, sem otimizações e executado sem o auxílio do chip gráfico. Os valores apresentados nesta tabela, não levam em consideração os tempos de leitura e gravação das imagens, apresentando exclusivamente o tempo de execução dos métodos isoladamente, garantindo uma comparação mais justa.

Além disso, para uma melhor visualização dos ganhos de performance da implementação proposta em relação às outras, é elaborada a Tabela 5.5, que mostra a relação de otimização no tempo de execução entre a implementação paralela e serial (CUDA vs. CPU) e também em relação a implementação híbrida (CUDA vs. OpenCV).

Tabela 5.4: Tempos de execução das principais funções do método proposto em diferentes tipo de implementação (valores em milissegundos).

Operação	**CUDA**	**OpenCV**	**CPU**
Conversão para Cinza	0,269	1,320	48,514
IPM	0,305	20,913	97,181
Obscurecimento Temporal	0,763	2,834	33,346
Limiar Adaptativo	0,139	0,917	42,109
Filtro de Correlação	0,677	19,564	62,887
Histograma de Coluna	0,081	7,091	38,348
Janelas Deslizantes	0,106	8,603	9,098
Tempo de Execução Total	2,34	61,242	331,483

Os tempos de execução apresentados para o método heterogêneo, com as otimizações propostas, são significativamente melhores que os tempos das outras implementações. Em termos médios, as otimizações propostas para o método garantem um ganho de performance de 25 e 140 vezes em relação ao método intermediário e serial. Vale ressaltar que todos os métodos chegam ao mesmo resultado, todas saídas e imagens são comparadas entre si, garantindo que a execução seja a mesma.

Naturalmente os ganhos e os desempenhos em cada uma das etapas do algoritmo são distintos, em partes específicas do método é possível observar que a implementação proposta

Tabela 5.5: Ganhos de performance da implementação heterogênea (CUDA) em relação aos outros métodos propostos.

Operação	CUDA vs. CPU	CUDA vs. OpenCV
Conversão para Cinza	180,3	4,9
IPM	318,6	68,5
Obscurecimento Temporal	43,7	3,7
Limiar Adaptativo	302,9	6,6
Filtro de Correlação	92,9	28,9
Histograma de Coluna	473,4	87,5
Janelas Deslizantes	85,8	81,1
Ganhos Totais	141,6	26,1

é superior as outras, como no caso do método de histograma de colunas, onde a performance da execução otimizada é cerca de 473 vezes mais rápida que a versão serial e 87 vezes mais rápida que a versão que utiliza a biblioteca com recursos da GPU. Os ganhos são maiores nas operações que possuem alto grau de paralelização e que fazem uso de instruções SIMD e SIMT para a solução. Algumas operações mais comuns e de uso geral, como no caso da conversão de escala de cinza e a operação do filtro de correlação unilateral, possuem otimizações na biblioteca OpenCV, que utiliza os recursos da GPU para melhorar a eficiência, neste tipo de operação, os ganhos entre os dois métodos são menores.

A taxa de aquisição de uma câmera é tipicamente de 30 fps, isto significa que para atingir a execução em tempo real deste sensor, é preciso atender ao limite de execução entre um quadro e outro de entrada. Em outras palavras, atender os requisitos de tempo real neste tipo de sistema, significa processar um quadro antes que o próximo seja obtido pelo sensor. Caso haja um atraso no processamento, dois cenário são possíveis: os atrasos se somam causando um atraso na transmissão da informação, o que é catastrófico neste tipo de sistema, ou em um segundo cenário, os quadro são perdidos para evitar o atraso, o que significa perder informações das vias por períodos de tempo, o que pode inviabilizar ou mesmo inutilizar o sistema.

Analisando os tempos totais, é possível entender como é o desempenho de cada uma das implementações em cenário real na plataforma embarcada. A versão inicial na CPU utiliza pouco mais de 300 ms para processar um único quadro, isto significa uma taxa de três quadros por segundo (3 fps), inviabilizando a aplicação para a solução do problema de detecção de faixas centrais no hardware embarcado, pois não atinge os critérios de execução em tempo real. O principal motivo que inviabiliza a execução deste tipo de algoritmo é o tempo demandado nas operações de processamento de imagem, que são realizadas de forma sequencial, não aproveitando as características paralelizáveis deste tipo de algoritmo.

A execução da versão intermediária, utilizando a biblioteca compilada para GPU, foi comparativamente melhor que a implementação serial, executando 5,5 vezes mais rápido. Isto significa que cada quadro foi processado em pouco mais de 60 ms ou à uma taxa de 16 fps. Apesar de desempenhar melhor que o método serial, apresentando grandes melhorias no tempo de algumas operações de processamento de imagem, como conversão para escala de cinza, filtragens e integração temporal, as otimizações fornecidas pela biblioteca e o uso da GPU não são suficientes para atingir o critério para a execução em tempo real.

Finalmente, o método heterogêneo com as otimizações propostas pode processar um único quadro em menos de 3 ms. Isto é equivalente a processar mais de 300 fps, alcançando todos os requisitos para a execução em tempo real no sistema embarcado, sendo equivalente a dez vez a taxa requisitada.

É evidente que a implementação proposta, maximiza o uso dos recursos disponíveis no hardware embarcado e torna possível a aplicação deste tipo de algoritmo em um sistema real. Não obstante, as otimizações propostas não viabilizam somente a execução deste método em específico, mas torna possível a utilização de uma gama de outros algoritmos, para aprimorar a qualidade da detecção do método, ou seja, além de viabilizar a execução, permite que a mesma seja melhorada por outros métodos como, filtros estimadores, extratores de características, etc., sem perder o requisito de tempo real.

5.2.4 Análise Quantitativa do Experimento na Base de Dados TuSimple

Para avaliar o método do ponto de vista de qualidade de detecção das faixas, é feita a comparação dos resultados obtidos pelo método proposto com as anotações da base de dados TuSimple. A principal motivação para esta comparação, é obter e validar o algoritmo de detecção de faixas centrais proposto, como um método válido para desempenhar tal tarefa. Desta forma, validando se a implementação proposta atinge os requisitos de tempo real em um sistema embarcado e consegue realizar as detecções com uma precisão adequada para este tipo de aplicação.

Para conseguir mensurar a qualidade e a precisão das detecções é utilizado um conjunto de métricas e parâmetros com base nas anotações fornecidas pela base de dados. A principal métrica avaliada para a detecção de faixas é a acurácia (ACC), descrita pela Equação 5.4

$$\text{ACC} = \frac{\sum_{clipe} P_{válido}}{\sum_{clipe} P_{total}}, \tag{5.4}$$

onde $P_{válido}$ corresponde a quantidade de pontos estimados corretamente e P_{total} representa o total de pontos estimados. Um ponto estimado, é dito válido se a diferença entre ele e o valor real, fornecido pela anotação, for menor que um valor limiar $T_{píxel}$. Esta distância leva em consideração a angulação da faixa, tornando a comparação mais justa.

Em resumo, a métrica ACC mede a taxa de estimações válidas feitas pelo método proposto, comparando cada um dos pontos com o valor da anotação da base de dados, ou seja, para cada ponto estimado pelo método proposto é comparada a distância dele com um valor limite, determinando se o ponto é válido ou não.

Além de avaliar cada um dos pontos estimados é importante também analisar cada uma das faixas, ou seja, estender a análise ponto-a-ponto para uma faixa-a-faixa. Assim, uma faixa é dita válida, caso tenha pelo menos $T_{pontos}\%$ de seus pontos internos válidos. Esta métrica é melhor apresentada na Equação 5.5a. De forma similar, é medida a taxa de falso positivos na estimação da faixas de trânsito, como apresentado na Equação 5.5b, assim, as estimações que são obtidas, mas não pertencem a nenhuma faixa catalogada na anotação daquele quadro.

$$\text{D} = \frac{M_{pred}}{N_{pred}} \tag{5.5a}$$

$$\text{FP} = \frac{F_{pred}}{N_{pred}} \tag{5.5b}$$

O valor de M_{pred} representa o número de faixas estimadas que possuem uma correspondência com as faixas anotadas. Similarmente, F_{pred} representa a quantidade de faixas que não possuem correspondência nas faixas anotadas. Além disso, o total de faixas estimadas é representado por N_{pred}. Desta forma, é possível obter a taxa de faixas detectadas ($D\%$) e a taxa de falso positivos ($FP\%$).

Todas as imagens disponíveis na base de dados TuSimple foram executadas e os resultados foram coletados. Em seguida, cada um dos pontos e faixas são avaliados, segundo as métricas apresentadas. Os resultados de ACC, $D\%$ e $FP\%$ obtidos para a execução de cada um dos subconjuntos da base de dados TuSimple estão organizados e dispostos na Tabela 5.6. Os resultados são comparados utilizando diferentes valores de limiares, variando $T_{píxel}$ de 20 px, 35 px, e 50 px, enquanto o valor de T_{pontos} foi mantido em 80%.

Tabela 5.6: Métricas e parâmetros de performance para diferentes limiares em cada um dos subconjuntos da base da dados TuSimple para o método proposto.

Subconjunto	$T_{píxel}$ (px)	ACC (%)	$D\%$	$FP\%$
0313	20	81,3	74,0	26,0
	35	88,4	84,4	15,6
	50	92,2	87,7	12,3
0531	20	90,0	87,7	12,3
	35	95,2	94,8	5,2
	50	97,3	95,9	4,0
0601	20	92,3	91,0	9,0
	35	97,2	97,3	2,7
	50	99,0	97,9	2,1

As taxas de ACC para o método proposto variam de $81,3\%$ e $99,0\%$, no pior e melhor caso, respectivamente. O desempenho no subconjunto 0313 é degradado devido a grande parcela de imagens com as marcações do tipo Bott, usualmente prateados ou brancos. Este tipo de marcação não é usual, e consiste em pequenos pinos metálicos no concreto e em muitas imagens se mistura com o asfalto e sujeira. Os métodos de extração de características propostos, são otimizados para as marcações de faixas padrões, onde seguimentos de retas são pintando no asfalto. Por outro lado, o subconjunto 0601 possui longos períodos com faixas em boas condições, com sinalizações tradicionais, obtendo em resultados melhores.

O método proposto é executado e avaliado em diferentes condições, tanto em relação a qualidade do asfalto e o tipo de marcação, quanto a diferentes período do dia com diferente iluminações e condições. Os resultados obtidos são satisfatórios mesmo para a detecção das faixas não convencionais do subconjunto 0313.

Em uma análise geral, a acurácia para os valores de $T_{píxel}$ = 20 px, 35 px, e 50 px foram respectivamente $91,2\%$, $96,2\%$ e $98,1\%$. Comprovando que o método possui uma performance elevada em relação ao tempo de execução e a qualidade de detecção das faixas de trânsito.

5.2.5 Comparação com outros Métodos

Para completar a análise do experimento paralelo, é proposta uma avaliação qualitativa para entender onde o método proposto se situa perante o cenário de soluções de detecção de faixas. A Tabela 5.7 relaciona a implementação proposta com outros métodos no cenário de detecção de faixas de trânsito.

Os trabalhos analisados não correspondem exatamente as características do método proposto, existem diferentes tipos de implementações, variando entre métodos de detecção de faixas centrais, múltiplas faixas e aproximações lineares de faixas. A coluna Método apresenta a tarefa e a especificação do método analisado.

Além disso, a maioria dos projetos não é executado em um ambiente embarcado de tempo real, como discutido anteriormente, a revisão sobre este tema ainda é pouco discutida. Todos os valores apresentados na Tabela 5.7 são referentes as suas próprias publicações (coluna Algoritmo), portanto, são executados em hardwares diferentes, conforme disposto na coluna Hardware. Esses hardwares variam desde placas embarcadas de baixo custo, como a NVIDIA Jetson Nano, até computadores complexos para execuções de algoritmos de aprendizado de máquina.

Apesar das diferenças, a Tabela 5.7 possibilita uma comparação razoavelmente justa entre os métodos, mas se faz necessário considerar suas especificidades, tais como: os métodos são executados em sistemas diferentes, eles utilizam conjuntos de dados distintos e aplicam metodologias diferentes para estabelecer suas métricas. No entanto, a comparação mostra fortes evidências de que o sistema proposto tem um desempenho superior, considerando aplicações embarcadas de tempo real.

Os métodos de detecção de múltiplas faixas baseados em *deep learning* (PHILION, 2019; ZOU et al., 2020) são executados sobre a mesma base de dados TuSimple. Estas implementações apresentam alta taxa de detecção e conseguem performar em tempo real, mesmo executando uma tarefa mais complexa. Contudo, ambos os trabalhos executam seus métodos em computadores com placas gráficas dedicadas, possuindo capacidades computacionais muito superiores as encontradas em sistemas embarcados convencionais, o que extrapola a limitação dos recursos.

Os trabalhos Muthalagu, Bolimera e Kalaichelvi (2020), Wu, Wang e Wang (2019) e Kühnl, Kummert e Fritsch (2012) apresentam soluções para a detecção de faixas centrais, assim como o método proposto. Contudo, os algoritmos são executados em computadores convencionais, e não realizam a implementação embarcada de tempo real. A taxa de detecção destes métodos é inferior a apresentada pelos outros métodos, em especial os que utilizam as mesmas plataformas. Além disso, as implementações não conseguem atender os critérios de tempo real, nem mesmo na execução em computadores convencionais, sendo que o mais rápido dos métodos utiliza imagens de baixa resolução e um chip gráfico dedicado.

Tabela 5.7: Comparação na performance de diferentes algoritmo de detecção de faixas em diferentes hardwares.

Método	Algoritmo	Detecção Média (%)	Tempo de Processamento Médio (ms) / Quadro	Resolução de Entrada	Hardware
Múltiplas Faixas	(SON; LEE; KUM)	98,9	667,0	640x360	Intel Core i7-4th
Faixas Centrais	(WU; WANG; WANG)	96,33	261,1	1280x720	Intel Core i7-2th
Múltiplas Faixas	(ZHENG et al.)	95,7	65,4	768x432	Intel Core i7-6th
Múltiplas Faixas (deep leaning)	(PHILION)	97,25	65,3	1280x720	NVIDIA GeForce GTX 1080, GPU
Faixas Centrais	(MUTHALAGU; BOLIMERA; KALAICHELVI)	95,75	63,6	1280x720	Intel Core i5-6th
Faixas Centrais	(KÜHNL; KUMMERT; FRITSCH)	94,40	45,0	800x600	NVIDIA GeForce GTX 580, GPU
Múltiplas Faixas (deep learning)	(ZOU et al.)	97,25	42,0	1280x720	Intel Core Xeon E5-2th GeForce GTX TITAN-X, GPU
Múltiplas Faixas (retas)	(KIM; BEAK; PARK)	98,10	34,0	640x480	NVIDIA Jetson TK1, board
Múltiplas Faixas	(CAO et al.)	98,42	22,2	1280x720	Intel Core i5-6th
Faixas Centrais	Proposto	98,20	2,34	1280x720	NVIDIA Jetson Nano, board

O trabalho Jihun et al. (2017) apresenta uma solução ainda mais simples para detecção de múltiplas faixas, é feita uma aproximação que considera que as faixas de trânsito são linhas retas, não detectando as curvaturas. É apresentada uma alta taxa de detecção, utilizando imagens de baixa resolução, o que impacta diretamente no tempo de execução e na qualidade dessas detecções. Porém, ao contrário dos outros métodos propostos, consegue atingir requisitos próximos do tempo real em um sistema embarcado. Contudo, como a qualidade da imagem tende a ter uma forte dependência do tempo de execução das operações de processamento de imagem, pode ser que o método não seja adequado para a execução de imagens de alta resolução, como as da base de dados TuSimple, obtendo um resultado similar a implementação serial ou mesmo a implementação intermediária proposta anteriormente.

Por fim, a partir do tempo médio de processamento de cada quadro, apresentado na Tabela 5.7, apenas o método proposto pode atender aos critérios de tempo real estabelecidos neste trabalho. A otimização realizada pela implementação heterogênea usando CUDA leva a um tempo médio de processamento de 2,34 ms. Isso indica que o método proposto pode funcionar usando apenas parte do poder de processamento e recursos de uma GPU personalizada para sistemas embarcados. Além disso, sua taxa média de detecção, excluindo o subconjunto TuSimple 0313, foi de aproximadamente 98%. Esses resultados demonstram a viabilidade de implementação do método proposto em sistemas embarcados. Além disso, é possível adicionar melhorias ou novos recursos ao algoritmo, sem comprometer os critérios em tempo real.

5.2.6 Conclusões do Experimento Paralelo

A Seção 5.2 apresenta de forma direta todo o fluxo de desenvolvimento do experimento com implementação heterogênea proposta no trabalho. Inicialmente são apresentados os requisitos dos testes, explicitando quais são os cenários utilizados para a elaboração do projeto, suas imagens de entrada, seu ambiente de programação e as distintas implementações realizadas para avaliar, de forma precisa, cada uma das métricas relevantes para o trabalho.

Desta forma, são implementadas três versões do mesmo algoritmo, uma versão serial executada na CPU, uma híbrida baseada no uso de uma biblioteca de processamento de imagens compilada para GPU e a versão paralela otimizada, proposta neste trabalho. Essas implementações, possuem as mesmas entradas e saídas, avaliadas através de testes de comparação píxel-a-píxel, garantindo que todos os métodos tenham o mesmo resultado. Assim, é possível, realizar uma comparação direta de como a abordagem heterogênea, em especial as otimizações propostas, são necessárias e suficientes para garantir a execução do algoritmo de detecção de faixas, em tempo real, no sistema embarcado. O método foi executado em menos de 3 ms o que corresponde a uma taxa de quadros superior a 300 fps, sendo 140 vezes e 25 vezes mais eficiente que as implementações serial e híbrida, respectivamente. Além disso, essa abordagem ressalta o impacto deste tipo de arquitetura heterogênea para a solução desta gama de problemas, possibilitando inclusive o aprimoramento do algoritmo através de outros métodos, como filtros estimadores e refinamento de candidatos, de forma a melhorar o desempenho de detecção, sem perder os critérios de tempo real.

Por fim, são analisadas as métricas de acurácia do método na tarefa de detecção de faixas centrais. O algoritmo se mostrou eficiente para detectar as faixas de trânsito em diversos cenários, como estradas e rodovias, em diferentes condições, horários e trânsito. O desempenho do método variou de 92,3% a 99,0% de acurácia no melhor subconjunto da base de dados, detectando até 97,9% das faixas centrais disponíveis com 80% dos pontos detectados válidos. Do ponto de vista comparativo com outros trabalhos da área, o método se apresenta em uma faixa de acurácia similar, contudo realiza a tarefa em um sistema embarcado, executando dez vezes mais rápido que o critério de tempo real.

Capítulo 6

Conclusões

O objetivo deste capítulo é apresentar quais são as principais conclusões do trabalho, isto é, expor de forma direta quais etapas foram executadas, quais são seus desfechos e possíveis avanços. Desta forma, em um primeiro momento é apresentado um resumo geral de como foi feito o desenvolvimento dos métodos propostos em seus diferentes cenários, os resultados obtidos nesses testes e quais são suas implicações.

Em um segundo momento, são apresentadas as conclusões gerais sobre o hardware utilizado e como tem desempenhado com os algoritmos seriais e paralelos implementados. Além disso, são expostos os principais resultados obtidos da implementação completa do software proposto, apresentando conclusões sobre os métodos heterogêneos e os seus ganhos em relação aos convencionais.

Por fim, é discutida as implicações da abordagem otimizada proposta na solução do problema de detecção de faixas em sistemas embarcados de tempo real, salientando os ganhos da abordagem heterogênea e as otimizações em CUDA.

6.1 Métodos e Algoritmos

Nesta seção são apresentadas as principais conclusões e resultados obtidos em todas as implementações realizadas e discutidas no Capítulo 5. Como forma de cobrir todos os avanços, são apresentadas as conclusões dos algoritmos propostos para a solução do problema de detecção de faixas, bem como as implementações e otimizações que tornam esses algoritmos viáveis em um ambiente embarcado de tempo real.

O método proposto é constituído de três principais partes, a preparação da imagem de entrada, a extração das características das faixas de trânsito centrais e a estimação da posição das faixas. A primeira etapa, busca simplificar os dados de entrada reduzindo a quantidade de informação presente na imagem, em especial, as informações que possuem baixo valor para o método de detecção, como as cores. Além disso, para solucionar um problema encontrado na primeira implementação (SILVA et al., 2020) discutida na Seção 5.1, é proposto uma forma de aprimorar a qualidade das marcações das vias e inserir mais características na imagem, fazendo uso de uma integração temporal dos quadros recebidos. Esse mecanismo, em conjunto com a perspectiva BEV, garantem uma qualidade superior aos quadros padrões, facilitando a detecção e em muitos casos, tornando-a possível. A IPM utilizada na versão final é também aprimorada da versão inicial, garantindo um campo de visão maior e uma linearidade maior nas faixas, o que auxilia na detecção e estimação das posições. Em muitos casos, o fato das características lineares das faixas na imagem processada (garantido pela IPM) e o aprimoramento de informações do desvanecimento temporal, tornaq factível a detecção das faixas em quadros onde a informação das faixas é precário.

Como mencionado, as faixas após o pré-processamento possuem uma característica linear forte, sendo praticamente verticais na imagem, e tendem a ser marcações contínuas, mesmo em sinalizações tracejadas. Isso garante uma estabilidade suficiente para a utilização de detectores de bordas mais simples, como o apresentado na Seção 4.4.2, evitando os filtros mais complexos, utilizados na versão inicial (SILVA et al., 2020). Outro problema detectado na implementação inicial, foi a instabilidade na variação da iluminação das vias. Para contornar isto, foi proposto um método de limiar adaptativo, que se ajusta de acordo com a intensidade dos píxeis da imagem, garantindo uma extração estável, baseada na diferença da iluminação do asfalto e das faixas. Aliando esta filtragem anterior com a extração da variação horizontal da intensidade de luz na posição das faixas, é possível obter um extrator de características estável e robusto, que realiza a detecção das faixas em diferentes cenários, com variadas condições de vias, iluminação e sinalização.

Por fim, o método de estimação da posição das faixas utiliza o algoritmo de janelas deslizantes para detectar os pontos pertencentes as faixas nos mapas de características. Como avaliado, o algoritmo se mostrou suficientemente bom para realizar as detecções, mesmo em condições onde pouca informação era extraída, realizando a detecção de faixas contínuas, des-

contínuas em situação de curva ou em trechos lineares de via. O método se mostrou com um desempenho elevado para a execução em hardware limitados, mesmo na sua versão serial.

6.2 Implementações e Experimentos

Este trabalho apresenta uma solução viável para detecção de faixas centrais para sistemas embarcados de tempo real, por exemplo, NVIDIA Jetson Nano. O algoritmo proposto usa uma implementação heterogênea, permitindo a otimização de todas as partes do algoritmo de detecção, propiciando uma implementação que faz uso do máximo de recursos disponíveis no hardware, atingindo altas taxas de execução e eliminando os gargalos da aplicação.

O algoritmo pré-processa a imagem de entrada para obter uma perspectiva BEV e com as marcações das faixas aprimoradas por uma integração temporal. A partir desta imagem, dois mapas de características são extraídos e combinados para produzir uma imagem que contém apenas as marcações das faixas, que são então, detectadas.

O método funciona dentro dos limites de tempo real estabelecidos. O sistema embarcado é capaz de realizar a execução do método completo em aproximadamente 2,34 ms. Isto é equivalente a processar mais de 300 quadros por segundo, sendo cerca de dez vezes mais rápido que o critério de tempo real. Esta implementação, comparativamente, é 25 vezes mais rápida do que a versão que utiliza uma biblioteca de processamento de imagens compilada para utilizar os recursos da GPU e 140 vezes mais rápida, do que a versão sequencial executada pela CPU, obtendo os mesmos resultados. Além disso, considerando o tempo de execução do método, é possível constatar que além de possibilitar uma solução viável para a detecção de faixas em um sistema embarcado, o que não foi atingido nas outras implementações, possibilita também a oportunidade de aprimorar o método proposto com adições de novos algoritmos, como filtros estimadores e outras técnicas de detecção, ainda mantendo os critérios de tempo real.

Ademais, o método se mostrou eficiente para detectar as faixas de trânsito em diferentes cenários como estradas e rodovias, em diferentes condições. O experimento foi realizado através da base de dados TuSimple que fornece uma gama de cenários. O desempenho do método variou de 92,3% a 99,0% de acurácia no melhor subconjunto da base de dados, detectando até 97,9% das faixas centrais disponíveis com 80% dos pontos detectados válidos. Desta

forma, o trabalho está na mesma faixa de outros métodos semelhantes, com a diferença de ser executado em um sistema embarcado convencional de baixo custo em tempo real.

6.3 Trabalhos Futuros

Esta seção apresenta quais são as principais metas e objetivos sugeridos para os próximos trabalhos. De forma geral, existem três possíveis vertentes para novos avanços no projeto:

- ampliação das aplicações da implementação proposta, tanto em cenários online quanto offline;
- estudo de novas etapas e algoritmos para aprimoramento da acurácia do método;
- extensão da implementação proposta para atender a detecção de múltiplas faixas.

O primeiro ponto destacado é a ampliação dos cenários de aplicação do método, visando obter mais dados sobre a acurácia e performance real do algoritmo. Podendo ser executado em outras bases de dados, e especialmente, em veículos reais, obtendo análises mais profundas e necessárias de forma a estabelecer quais as modificações no método proposto devem ser exploradas.

O método desempenha a tarefa de detecção de faixas centrais de forma semelhante a outros métodos, do ponto de vista de acurácia, e com um desempenho superior ao critério de execução de tempo real. É possível, realizar o estudo e a análise de novos métodos para aprimorar a qualidade de detecção, ainda mantendo o critério de tempo real. Por exemplo, a implementação de extratores que tornem os mapas de características mais robustos a alguns casos de borda, como mudança abrupta de iluminação, ultrapassagens, etc. Outro ponto de melhoria, é a aplicação de filtros estimadores, como o filtro de Kalman, aplicado ao modelo das faixas, para realizar a estimação dos parâmetros com maior precisão, facilitando também, a etapa de detecção. Por fim, é possível realizar a implementação de uma etapa mais sofisticada para o refinamento de candidatos, utilizando algoritmos como o RANSAC, para selecionar as faixas de forma mais robusta, sendo possível ainda, alterar o método de janelas deslizantes.

Por fim, como proposto anteriormente, é possível realizar a implementação do método visando a detecção de múltiplas faixas, utilizando o algoritmo de janelas deslizantes, porém, estimando mais candidatos. Contudo, é possível realizar a análise de mais algoritmos

de detecção e até mesmo a combinação deles para a obtenção de resultados consistentes na detecção de múltiplas faixas.

Referências

AFIF, Mouna; SAID, Yahia; ATRI, Mohamed. Computer vision algorithms acceleration using graphic processors NVIDIA CUDA. *Cluster Computing*, v. 23, n. 4, p. 3335–3347, 2020. ISSN 1573-7543.

ALMAGAMBETOV, Akhan; VELIPASALAR, Senem; CASARES, Mauricio. Robust and Computationally Lightweight Autonomous Tracking of Vehicle Taillights and Signal Detection by Embedded Smart Cameras. *IEEE Transactions on Industrial Electronics*, v. 62, n. 6, p. 3732–3741, 2015.

ALY, Mohamed. Real time Detection of Lane Markers in Urban Streets. In: p. 7–12. ISBN 978-1-4244-2568-6.

ASUS. *Tiker Board S Product Details*. [S.l.: s.n.], 2019. Disponível em: <`https://www.asus.com/Single-Board-Computer/Tinker-Board-S/`>.

AYDIN, Semra; SAMET, Refik; BAY, Omer. Real-time parallel image processing applications on multicore CPUs with OpenMP and GPGPU with CUDA. *The Journal of Supercomputing*, 2017.

BAR HILLEL, Aharon et al. Recent progress in road and lane detection: A survey. *Machine Vision and Applications*, v. 25, 2014.

BAR HILLEL, Aharon et al. Recent progress in road and lane detection: a survey. *Machine Vision and Applications*, v. 25, n. 3, p. 727–745, 2014.

BENGLER, Klaus et al. Three Decades of Driver Assistance Systems: Review and Future Perspectives. *Intelligent Transportation Systems Magazine, IEEE*, v. 6, p. 6–22, 2014.

BERRIEL, Rodrigo et al. A Particle Filter-Based Lane Marker Tracking Approach Using a Cubic Spline Model. In.

BODIS-SZOMORU, Andras; DABOCZI, Tamas; FAZEKAS, Zoltan. A Far-Range Off-line Camera Calibration Method for Stereo Lane Detection Systems. In: p. 1–6.

BORKAR, A.; HAYES, M.; SMITH, M. T. A Novel Lane Detection System With Efficient Ground Truth Generation. *IEEE Transactions on Intelligent Transportation Systems*, v. 13, n. 1, p. 365–374, 2012.

BORKAR, A.; HAYES, M.; SMITH, M. T. A Novel Lane Detection System With Efficient Ground Truth Generation. *IEEE Transactions on Intelligent Transportation Systems*, v. 13, n. 1, p. 365–374, 2012.

BORKAR, Amol; HAYES, Monson; SMITH, Mark; PANKANTI, S. A layered approach to robust lane detection at night. In: p. 51–57.

CAO, Jianjun et al. Lane Detection Algorithm for Intelligent Vehicles in Complex Road Conditions and Dynamic Environments. *Sensors*, v. 19, p. 3166, 2019.

CAO, Jianjun et al. Lane Detection Algorithm for Intelligent Vehicles in Complex Road Conditions and Dynamic Environments. *Sensors*, v. 19, p. 3166, 2019.

CHAN, Y.; LIN, Y.; CHEN, P. Lane Mark and Drivable Area Detection Using a Novel Instance Segmentation Scheme. In: 2019 IEEE/SICE International Symposium on System Integration (SII). [S.l.: s.n.], 2019. p. 502–506.

CHEN, C. et al. DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. In: 2015 IEEE International Conference on Computer Vision (ICCV). [S.l.: s.n.], 2015. p. 2722–2730.

CHENG, H. et al. Lane Detection With Moving Vehicles in the Traffic Scenes. *IEEE Transactions on Intelligent Transportation Systems*, v. 7, n. 4, p. 571–582, 2006.

CHUANXIANG, Li. Multi-lane detection based on multiple vanishing point detection in structured environments. In.

DE PAULA, M. B.; JUNG, C. R. Automatic Detection and Classification of Road Lane Markings Using Onboard Vehicular Cameras. *IEEE Transactions on Intelligent Transportation Systems*, v. 16, n. 6, p. 3160–3169, 2015.

DENG, Jiayong; HAN, Youngjoon. A real-time system of lane detection and tracking based on optimized RANSAC B-spline fitting. In: p. 157–164.

DÍAZ-PERNIL, Daniel et al. Segmenting images with gradient-based edge detection using Membrane Computing. *Pattern Recognition Letters*, v. 34, n. 8, p. 846–855, 2013. Computer Analysis of Images and Patterns.

FELISA, Mirko; ZANI, Paolo. Robust monocular lane detection in urban environments. In: p. 591–596.

FOUNDATION, Raspberry Pi. *Raspberry Pi 4 Model B Product Details*. [S.l.: s.n.], 2019. Disponível em: <`https://www.raspberrypi.org/products/raspberry-pi-4-model-b/`>.

GAO, Qingji; QIJUN, Luo; MOLI, Sun. Rough Set based Unstructured Road Detection through Feature Learning. In: p. 101–106.

GOOGLE. *GoogleTest Users's Guide*. [S.l.: s.n.], 2021. Disponível em: <`http://google.github.io/googletest/`>.

GUO, Chunzhao; MITA, Seiichi. Drivable road region detection based on homography estimation with road appearance and driving state models. In: p. 204–209.

HARDKERNEL. *ODROID N2+ Product Details*. [S.l.: s.n.], 2020. Disponível em: <`https://www.hardkernel.com/shop/odroid-n2-with-4gbyte-ram-2/`>.

HARDKERNEL. *ODROID XU4Q Product Details*. [S.l.: s.n.], 2020. Disponível em: <`https://www.hardkernel.com/shop/odroid-xu4q-special-price/`>.

HARTLEY, Richard; ZISSERMAN, Andrew. *Multiple View Geometry in Computer Vision*. Segunda Edição: [s.n.], 2004.

HUANG, Yingping et al. Lane Detection Based on Inverse Perspective Transformation and Kalman Filter. *KSII Transactions on Internet and Information Systems*, v. 12, p. 643–661, 2018.

HUQQANI, Altaf Ahmad et al. Multicore and GPU Parallelization of Neural Networks for Face Recognition. *Procedia Computer Science*, v. 18, p. 349–358, 2013. 2013 International Conference on Computational Science.

HUVAL, Brody et al. An Empirical Evaluation of Deep Learning on Highway Driving, 2015.

JIHUN, Kim et al. Fast learning method for convolutional neural networks using extreme learning machine and its application to lane detection. *Neural Networks*, v. 87, p. 109–121, 2017. ISSN 0893-6080.

KHADAS. *Vim3 Pro Product Details*. [S.l.: s.n.], 2020. Disponível em: <https://www.khadas.com/product-page/vim3>.

KIM, Hee-Soo; BEAK, Seung-Hae; PARK, Soon-Yong. Parallel Hough Space Image Generation Method for Real-Time Lane Detection. In: ADVANCED Concepts for Intelligent Vision Systems. [S.l.: s.n.], 2016.

KIRK, David B.; HWU, Wen-Mei W. *Programming Massively Parallel Processors: A Hands-on Approach: Third Edition*. [S.l.: s.n.], 2016.

KREUCHER, Chris; LAKSHMANAN, Sridhar; KLUGE, Karl. A Driver Warning System Based on the LOIS Lane Detection Algorithm. *Proceeding of IEEE International Conference on Intelligent Vehicles*, 1998.

KÜHNL, Tobias; KUMMERT, Franz; FRITSCH, Jannik. Spatial ray features for real-time ego-lane extraction. In: 2012 15th International IEEE Conference on Intelligent Transportation Systems. [S.l.: s.n.], 2012. p. 288–293.

KUK, Junggap et al. Fast lane detection and tracking based on Hough transform with reduced memory requirement. In: p. 1344–1349.

KUM, Chang-Hoon et al. Lane detection system with around view monitoring for intelligent vehicle. *Proc. International SoC Design Conference*, p. 215–218, 2013.

LEE, C.; MOON, J. Robust Lane Detection and Tracking for Real-Time Applications. *IEEE Transactions on Intelligent Transportation Systems*, v. 19, n. 12, p. 4043–4048, 2018.

LI, Jincheng et al. Realization of CUDA-based real-time multi-camera visual SLAM in embedded systems. *Journal of Real-Time Image Processing*, 2019.

LI, Q. et al. A Sensor-Fusion Drivable-Region and Lane-Detection System for Autonomous Vehicle Navigation in Challenging Road Scenarios. *IEEE Transactions on Vehicular Technology*, v. 63, n. 2, p. 540–555, 2014.

LI, Wenhui et al. A robust lane detection method based on hyperbolic model. *Soft Computing*, v. 23, 2018.

LI, Xiangyang et al. Lane Detection and Tracking Using a Parallel-snake Approach. *Journal of Intelligent and Robotic Systems*, v. 77, 2014.

LIPSKI, Christian et al. A Fast and Robust Approach to Lane Marking Detection and Lane Tracking. In: v. 0, p. 57–60. ISBN 978-1-4244-2296-8.

MA, Bing; LAKSHMANAN, Sridhar; HERO, Alfred. Pavement Boundary Detection Via Circular Shape Models, 2000.

MARR, D.; HILDRETH, Ellen. Theory of Edge Detection. *Proceedings of the Royal Society of London. Series B, Containing papers of a Biological character. Royal Society (Great Britain)*, v. 207, p. 187–217, 1980.

MERTES, Jacqueline G.; MARRANGHELLO, Norian; PEREIRA, Aledir S. Real-time Module for Digital Image Processing Developed on a FPGA. *IFAC Proceedings Volumes*, v. 46, n. 28, p. 405–410, 2013. 12th IFAC Conference on Programmable Devices and Embedded Systems.

MONDAL, Pulak; BISWAL, Pradyut Kumar; BANERJEE, Swapna. FPGA based accelerated 3D affine transform for real-time image processing applications. *Computers and Electrical Engineering*, v. 49, p. 69–83, 2016.

MUTHALAGU, Raja; BOLIMERA, Anudeepsekhar; KALAICHELVI, V. Lane detection technique based on perspective transformation and histogram analysis for self-driving cars. *Computers and Electrical Engineering*, v. 85, p. 106653, 2020. ISSN 0045-7906.

NAROTE, Sandipann P. et al. A review of recent advances in lane detection and departure warning system. *Pattern Recognition*, v. 73, p. 216–234, 2018. ISSN 0031-3203.

NEVEN, D. et al. Towards End-to-End Lane Detection: an Instance Segmentation Approach, p. 286–291, 2018.

NHTSA. *Critical reasons for crashes investigated in the national motorvehicle crash causation survey*. [S.l.], 2015.

NIETO, Marcos; ARROSPIDE, Jon; SALGADO, Luis. Road environment modeling using robust perspective analysis and recursive Bayesian segmentation. *Mach. Vis. Appl.*, v. 22, p. 927–945, 2011.

NIETO, Marcos; CORTES, Andoni et al. Real-time lane tracking using Rao-Blackwellized particle filter. *Journal of Real-Time Image Processing*, v. 11, 2012.

NVIDIA. *CUDA C++ Programming Guide, The programming guide to the CUDA model and interface*. [S.l.: s.n.], 2021. Disponível em: <`https://docs.nvidia.com/cuda/cuda-c-programming-guide`>.

NVIDIA. *Developer Zone NVidia - CUDA Toolkit Documentation*. [S.l.: s.n.], 2020. Disponível em: <`https://docs.nvidia.com/cuda/`>.

NVIDIA. *GeForce GTX 980 - Featuring Maxwell, The Most Advanced GPU Ever Made.* [S.l.], 2014.

NVIDIA. *Jetson Nano Devolper Kit - User Guide.* [S.l.: s.n.], 2019. Disponível em: <https://developer.download.nvidia.com/embedded/L4T/r32-3-1_Release_v1.0/Jetson_Nano_Developer_Kit_User_Guide.pdf>.

NVIDIA. *Jetson Nano Product Details.* [S.l.: s.n.], 2019. Disponível em: <https://www.nvidia.com/pt-br/autonomous-machines/embedded-systems/jetson-nano/>.

NVIDIA. *Profiler User's Guide, The user manual for NVIDIA profiling tools for optimizing performance of CUDA applications.* [S.l.: s.n.], 2021. Disponível em: <https://docs.nvidia.com/cuda/profiler-users-guide>.

OPENCV. *Open Source Computer Vision - OpenCV Modules.* [S.l.: s.n.], 2020. Disponível em: <https://docs.opencv.org/>.

OZGUNALP, Umar et al. Multiple Lane Detection Algorithm Based on Novel Dense Vanishing Point Estimation. *IEEE Transactions on Intelligent Transportation Systems*, v. 18, p. 1–12, 2016.

PHILION, Jonah. FastDraw: Addressing the Long Tail of Lane Detection by Adapting a Sequential Prediction Network. In: 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). [S.l.: s.n.], 2019. p. 11574–11583.

PROCHAZKA, Zdenek. Road Tracking Method Suitable for Both Unstructured and Structured Roads. *International Journal of Advanced Robotic Systems*, v. 10, p. 1, 2013.

RASMUSSEN, C. Grouping dominant orientations for ill-structured road following. In: v. 1, p. i–470. ISBN 0-7695-2158-4.

REICHENBACH, M. et al. Comparison of Lane Detection Algorithms for ADAS Using Embedded Hardware Architectures, p. 48–53, 2018.

SATZODA, R. K.; TRIVEDI, M. M. On Enhancing Lane Estimation Using Contextual Cues. *IEEE Transactions on Circuits and Systems for Video Technology*, v. 25, n. 11, p. 1870–1881, 2015.

SATZODA, R. K.; TRIVEDI, M. M. On Performance Evaluation Metrics for Lane Estimation. In: 2014 22nd International Conference on Pattern Recognition. [S.l.: s.n.], 2014. p. 2625–2630.

SATZODA, R. K.; TRIVEDI, M. M. Vision-Based Lane Analysis: Exploration of Issues and Approaches for Embedded Realization. In: 2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops. [S.l.: s.n.], 2013. p. 604–609.

SATZODA, Ravi; SATHYANARAYANA, Suchitra; SRIKANTHAN, Thambipillai. Gradient angle histograms for efficient linear Hough transform. In: p. 3273–3276.

SCARAMAL, João Marcos. *An Embedded Real-Time Homography Estimation Using SURF Descriptors*. 2017. Tese (Doutorado) – Instituto Tecnológico de Aeronáutica.

SILVA, Guilherme et al. Estratégia de detecção de faixas de trânsito baseada em câmera monocular para sistemas embarcados. *Anais do XXIII Congresso Brasileiro de Automática*, Volume 2 No 1, 2020. ISSN 2525-8311.

SILVA, Guilherme B. *CUDA-based Real-Time Ego-Lane Detection in Embedded System-TuSimple#0601*. 15 out. 2021. Disponível em: <`https://youtu.be/1q_Jy8MJjFw`>.

SIVARAMAN, S.; TRIVEDI, M. M. Integrated Lane and Vehicle Detection, Localization, and Tracking: A Synergistic Approach. *IEEE Transactions on Intelligent Transportation Systems*, v. 14, n. 2, p. 906–917, 2013.

SON, Jongin; YOO, Hunjae et al. Real-time illumination invariant lane detection for lane departure warning system. *Expert Systems with Applications*, v. 42, 2014.

SON, Yeongho; LEE, Elijah; KUM, Dongsuk. Robust multi-lane detection and tracking using adaptive threshold and lane classification. *Machine Vision and Applications*, v. 30, 2018.

SON, Yeongho; LEE, Elijah; KUM, Dongsuk. Robust multi-lane detection and tracking using adaptive threshold and lane classification. *Machine Vision and Applications*, v. 30, 2018.

SONG, Wenjie et al. Real-time lane detection and forward collision warning system based on stereo vision. In: p. 493–498.

SOQUET, Nicolas; AUBERT, Didier; HAUTIÈRE, Nicolas. Road Segmentation Supervised by an Extended V-Disparity Algorithm for Autonomous Navigation. In: p. 160–165.

TRANSP., Stationery Office Dept. *Reported road casualties Great Britain 2009*. [S.l.], 2010.

TSUNG-YING SUN; SHANG-JENG TSAI; CHAN, V. HSI color model based lane-marking detection. In: 2006 IEEE Intelligent Transportation Systems Conference. [S.l.: s.n.], 2006. p. 1168–1172.

TU, Chunling et al. Vehicle Position Monitoring Using Hough Transform. *IERI Procedia*, v. 4, p. 316–322, 2013.

TUSIMPLE. *TuSimple Dataset Benchmark*. [S.l.: s.n.], 2018. Disponível em: <`https://github.com/TuSimple/tusimple-benchmark`>.

VAJAK, Denis et al. Recent advances in vision-based lane detection solutions for automotive applications. *Proceedings Elma - International Symposium Electronics in Marine*, p. 45–50, 2019.

WANG, Yifei; DAHNOUN, Naim; ACHIM, Alin. A novel system for robust lane detection and tracking. *Signal Processing*, v. 92, p. 319–334, 2012.

WANG, Yue; TEOH, Eam; SHEN, Dinggang. Lane detection and tracking using B-Snake. *Image and Vision Computing*, v. 22, p. 269–280, 2004.

WU, Bing-Fei; LIN, Chuan-Tsai; CHEN, Yen-Lin. Dynamic Calibration and Occlusion Handling Algorithms for Lane Tracking. *Industrial Electronics, IEEE Transactions on*, v. 56, p. 1757–1773, 2009.

WU, C.; WANG, L.; WANG, K. Ultra-Low Complexity Block-Based Lane Detection and Departure Warning System. *IEEE Transactions on Circuits and Systems for Video Technology*, v. 29, n. 2, p. 582–593, 2019.

YAN, X.; LI, Y. A method of lane edge detection based on Canny algorithm. *2017 Chinese Automation Congress (CAC)*, p. 2120–2124, 2017.

YENIKAYA, Sibel; YENIKAYA, Gökhan; DÜVEN, Ekrem. Keeping the Vehicle on the Road: A Survey on on-Road Lane Detection Systems. New York, NY, USA, v. 46, n. 1, 2013. ISSN 0360-0300.

YINGHUA HE; HONG WANG; BO ZHANG. Color-based road detection in urban traffic scenes. *IEEE Transactions on Intelligent Transportation Systems*, v. 5, n. 4, p. 309–318, 2004.

YOO, Hunjae; YANG, Ukil; SOHN, Kwanghoon. Gradient-Enhancing Conversion for Illumination-Robust Lane Detection. *Intelligent Transportation Systems, IEEE Transactions on*, v. 14, p. 1083–1094, 2013.

YU, Jaehyoung; HAN, Youngjoon; HAHN, Hernsoo. An Efficient Extraction of On-Road Object and Lane Information Using Representation Method. In: p. 327–332.

YU, Yang; JO, Kang-Hyun. Lane detection based on color probability model and fuzzy clustering. In: YU, Hui; DONG, Junyu (Ed.). *Ninth International Conference on Graphic and Image Processing (ICGIP 2017)*. [S.l.: s.n.], 2018. v. 10615. International Society for Optics e Photonics, p. 44–49.

YUE, Wang; DINGGANG, Shen; EAM KHWANG, Teoh. Lane detection using spline model. *Pattern Recognition Letters*, v. 21, n. 8, p. 677–689, 2000.

ZENG, Haihua et al. Visual tracking using multi-channel correlation filters. In: 2015 IEEE International Conference on Digital Signal Processing (DSP). [S.l.: s.n.], 2015. p. 211–214.

ZHANG, Hao; HOU, Dibo; ZHOU, Zekui. A Novel Lane Detection Algorithm Based on Support Vector Machine. *Piers Online*, v. 1, p. 390–394, 2005.

ZHENG, Fang et al. Improved Lane Line Detection Algorithm Based on Hough Transform. *Pattern Recognition and Image Analysis*, v. 28, n. 2, p. 254–260, 2018. ISSN 1555-6212.

ZHI, Xiyang et al. Realization of CUDA-based real-time registration and target localization for high-resolution video images. *Journal of Real-Time Image Processing*, v. 16, n. 4, p. 1025–1036, 2019.

ZHICHENG ZHANG, Xin Ma. Lane Recognition Algorithm Using the Hough Transform Based on Complicated Conditions. *Journal of Computer and Communications*, v. 7, p. 65–75, 2019.

ZHOU, S. et al. A novel lane detection based on geometrical model and Gabor filter, p. 59–64, 2010.

ZOU, Qin et al. Robust Lane Detection From Continuous Driving Scenes Using Deep Neural Networks. *IEEE Transactions on Vehicular Technology*, v. 69, n. 1, p. 41–54, 2020.

ZUO, Wen-hui; YAO, Tuo-zhong. Road model prediction based unstructured road detection. *Journal of Zhejiang University SCIENCE C*, v. 14, p. 822–834, 2013.

Apêndice A

Experimento Serial (CBA20) - Artigo publicado no XXIII Congresso Brasileiro de Automática

Título: Estratégia de Detecção de Faixas de Trânsito Baseada em Câmera Monocular para Sistemas Embarcados

Autores: Guilherme Brandão da Silva, Daniel Strufaldi Batista, Marcelo Carvalho Tosin, e Leonimer Flávio de Melo.

Evento: XXIII Congresso Brasileiro de Automática (CBA 2020)

Data: 23 Novembro de 2020

Estratégia de Detecção de Faixas de Trânsito Baseada em Câmera Monocular para Sistemas Embarcados

Guilherme B. da Silva * **Daniel S. Batista** * **Marcelo C. Tosin** * **Leonimer F. de Melo** *

* *Departamento de Engenharia Elétrica, Universidade Estadual de Londrina, PR,*
e-mails: brandaogbs@gmail.com; daniel.s.batista@ieee.org; mctosin@uel.br; leonimer@uel.br

Abstract:
Road lane detection is comming a fundamental technology in autonomous smart car developments. Main objective of this work is to find a robust lane detection metodology maintaing a low computational cost, compatible with real embedded applications. The method proposed has three major steps: lane segmentation through combined threshold filters, generation of a region of interest from an aerial perspective, and lane marks detection using the Slidding Windows method. To evaluate it, the algorithm was implemented in a conventional embedded system based on an ARM A57 processor, processing a previously recorded footage from a monocular camera installed in a vehicle. Results include the analysis of accuracy metrics and execution times considering different road scenarios as well as different image resolutions. The tests achieved good lane marks detection rate, between 89% and 97%, according to the image resolution, and showed that its computational cost allows the execution in a typical embedded system.

Resumo:
A detecção de faixas sobre as vias de trânsito vem se destacando como uma das tecnologias fundamentais na implementação de veículos autônomos inteligentes. O objetivo principal deste trabalho é a busca por um método robusto para a detecção de faixas com custo computacional baixo, compatível com aplicações embarcadas. O método proposto é composto por três etapas: segmentação de faixas através da combinação de filtros de limiar, geração de uma região de interesse em perspectiva aérea, e detecção das faixas pelo método *Slidding Windows*. Para avaliação e validação do sistema proposto, o método foi implementado em um sistema embarcado convencional com processador ARM A57 e imagens reais de um veículo foram utilizadas. Foram analisadas métricas de acurácia e tempo de execução para diferentes cenários de vias, bem como para diferentes resoluções de imagem. Os testes realizados apresentaram bons índices de detecção das faixas de trânsito, entre aproximadamente 89 % e 97 %, conforme a resolução da imagem, além de mostrar que o algoritmo possui custo computacional compatível para ser executado por uma plataforma embarcada tradicional.

Keywords: Lane detection; embedded system; image segmentation; aerial perspective; slidding windows.

Palavras-chaves: Detecção de faixas; sistema embarcado; segmentação de imagem; perspectiva aérea; slidding windows.

1. INTRODUÇÃO

O aumento do fluxo de veículos nas últimas décadas foi acompanhado da alta no número de acidentes reportados e da consequente alta no número de vítimas. Segundo Transp. (2010), as falhas relacionadas ao tempo de resposta dos motoristas é responsável por 70% dos acidentes na Grã-Bretanha. De acordo com o estudo da NHTSA (2015), aproximadamente 94% das falhas críticas que precedem a cadeia de eventos de um acidente de trânsito podem ser atribuídas aos motoristas.

Buscando a redução dos acidentes, a indústria automobilística realiza constantes investimentos em sistemas de auxílio aos motoristas durante a condução, e até mesmo em sistemas que intervém de forma autônoma em condições identificadas como de grande risco ou críticas (Bengler et al., 2014).

Neste sentido, diversos sistemas automotivos foram desenvolvidos nas últimas décadas. Atualmente, o desenvolvimento destas tecnologias aponta para a criação de sistemas que futuramente evoluam para a direção autônoma e coo-

⋆ O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

perativa. Tais sistemas têm por fonte de informação uma rede integrada de sensores, tais como uma rede multiplataforma com câmeras, radares, sistemas de visão noturna, inerciais, entre outros.

Ainda segundo Bengler et al. (2014), para garantir que as tecnologias de direção autônoma e cooperativa funcionem adequadamente, diversos subsistemas devem atuar em conjunto. Um destes subsistemas essenciais é a detecção e identificação de faixas em vias e rodovias, que é tópico de estudo importante no campo dos sistemas de carros inteligentes.

Através da detecção das faixas, é possível fornecer a posição do veículo em relação às faixas, possibilitando a determinação de uma direção eficaz para o carro. Esse recurso pode aprimorar significativamente a eficiência e a segurança na direção (Kum et al., 2013).

A informação visual gerada por câmeras é um dos meios mais utilizados para a detecção de faixas, e consequentemente para a direção inteligente. A principal vantagem desta é a quantidade de informação que pode ser obtida em função do custo, tanto operacional quanto monetário. Adicionalmente, outros sensores, tais como ultrassônicos, radares e lasers, podem ser utilizados em conjunto com as câmeras (Bar Hillel et al., 2014).

É essencial que os algoritmos para a detecção de faixas tenham a menor complexidade computacional possível, uma vez que estes devem atender a requisitos de tempo real para sua operação em um sistema embarcado no veículo. Além disso, sistemas embarcados possuem recursos limitados e reduzidos, tais como a capacidade de processamento e a quantidade de memória, frente a um sistema computacional completo de uso pessoal, por exemplo.

Grande parte dos trabalhos sobre detecção de faixa encontrados na literatura desconsideram a complexidade computacional das soluções propostas, pois não têm como objetivo a implementação destes em sistemas embarcados. Este fato limita ou impossibilita a reprodução destes resultados em aplicações reais embarcadas.

O presente trabalho se propõe a implementar um método de detecção de faixas baseado em visão computacional para sistemas embarcados convencionais, operando com imagens de uma câmera monocular para realizar a detecção de vias com e sem curvatura e limitado à operação diurna. O método proposto adota uma estratégia iterativa de detecção de faixas contínuas e descontínuas, através de uma cadeia de processamento que tem como base a segmentação das sinalizações das vias e detecção das faixas.

Assim, a principal contribuição deste trabalho é a proposta de um método de detecção de faixas baseado em visão monocular capaz de operar em um sistema embarcado convencional. Isto é, um método que realize a detecção de faixas em tempo real e mantenha o compromisso de baixa complexidade computacional e alta taxa de detecção.

2. TRABALHOS RELACIONADOS

A detecção de faixas baseadas em sensores óticos utilizam imagens bidimensionais capturadas por uma câmera. Geralmente esta é montada no para-brisas do veículo e suas imagens são processadas por métodos de visão computacional, cujas informações são repassadas para o controle do mesmo. Tal processamento pode ser categorizado em duas principais classes: os baseados em recursos e os baseados em modelos.

A abordagem por modelos propõem um modelo matemático parametrizado que descreve a estrutura das faixas (Zhou et al., 2010). Esta técnica apresenta alta robustez a ruídos. Contudo sua implementação possui limitações, tais como o elevado custo computacional e o conhecimento prévio sobre os parâmetros geométricos da faixas.

Na metodologia baseada em recursos (Lee and Moon, 2018; Wang et al., 2004; Borkar et al., 2012; Li et al., 2014; Son et al., 2014), utilizada como base no presente trabalho, a análise da imagem busca detectar os gradientes, os padrões de cores, e outras informações presentes nos pixeis da imagem, a fim de reconhecer as sinalizações das faixas de trânsito.

Em Lee and Moon (2018) foi proposto um algoritmo de detecção baseado no uso de duas regiões de interesse, uma retangular e uma em formato de Λ, contornando os efeitos de ruídos externos e diminuindo o tempo de execução do algoritmo. Este sistema utiliza um filtro de Kalman e uma aproximação de movimento linear para rastrear as faixas, ao mesmo tempo que faz as detecções.

Em Wang et al. (2004) a região de interesse é dividida em seções finitas tornando mais simples o rastreamento de faixas de trânsito. Segmentos de linha extraídos pela transformada de Hough interpolam cada seção através do método *B-Snake*. O método é aprimorado em Li et al. (2014) que faz uso de um filtro de Kalman para estimar e refinar o rastreamento das faixas de transito.

Borkar et al. (2012) introduzem um método de detecção baseado na operação IPM (*Inverse Perspective Mapping*). Uma técnica de limiar adaptativo é utilizada para produzir imagens binárias. Além disso, são utilizados formatos pré determinados para selecionar as possíveis faixas. Para eliminar as detecções destoantes é utilizado o algoritmo de RANSAC (*Random Sample Consensus*).

Para garantir uma maior robustez com as varições de luminosidade, Son et al. (2014) apresentam um método de detecção de faixas que busca manter a iluminação das faixas constante através de filtros em diferentes espaços de cores.

3. ESTRATÉGIAS PARA DETECÇÃO DE FAIXAS

O método proposto para a detecção de faixas é sumarizado no diagrama da Figura 1. Na sequência, são apresentadas as estratégias e algoritmos propostos para a solução do problema em um sistema embarcado.

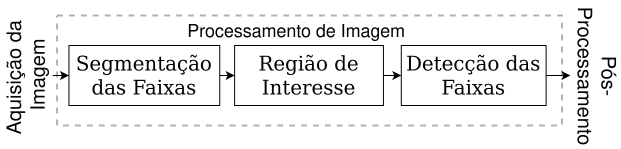


Figura 1. Diagrama simplificado da etapa de processamento da imagem.

Inicialmente, é obtida uma imagem binária em uma perspectiva aérea, através da segmentação dos pixeis com maior probabilidade de pertencimento às faixas de transito e da operação IPM.

Na sequência é aplicado um algoritmo que busca classificar as regiões segmentadas da imagem, identificando os pontos ao longo das faixas, tornando a detecção mais robusta e permitindo a parametrização das faixas de transito.

3.1 Segmentação das Faixas

Um dos métodos mais utilizados para a extração das características das faixas é a utilização de detectores de bordas. Abordagens que utilizam algoritmos convencionais de detecção de contornos apresentam diversos problemas para a aplicação de detecção de faixas, incluindo baixa robustez ao ruído e complexidade computacional elevada, e consequentemente um alto tempo de processamento. (Yousaf et al., 2018). Para contornar estes problemas, foi adotado um algoritmo que combina filtros de limiar de gradiente e filtros de limiar de canais.

Filtros de limiar (FL) são comumente utilizados na segmentação de imagens, separando objetos ou regiões da imagem original para uma imagem binária em preto e branco. A abordagem de combinação filtros de limiar proposta está ilustrada no diagrama da Figura 2.

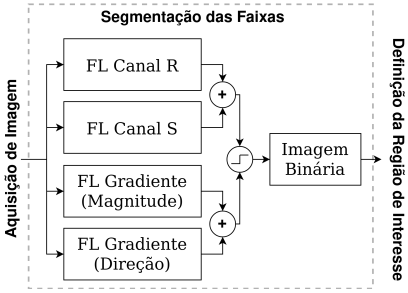


Figura 2. Diagrama do processo de filtragem e binarização da imagem referente a etapa de segmentação das faixas de transito.

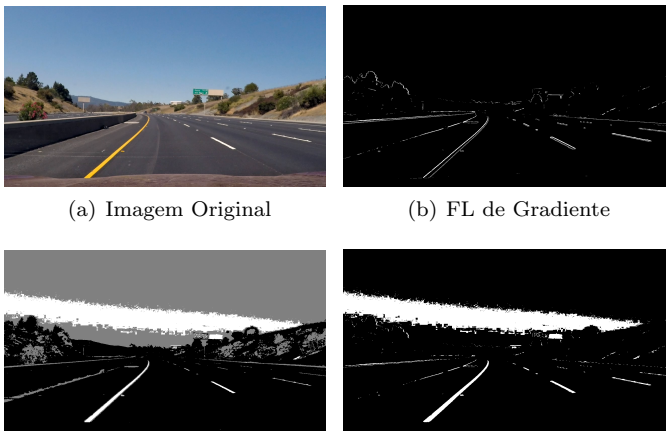
Utilizando o filtro Sobel são calculados os gradientes de intensidade da imagem em cada ponto. Obtendo uma noção de como varia a luminosidade nos pixeis, sendo de forma mais suave ou abrupta (Cao et al., 2019).

São determinadas a magnitude e direção dos gradientes do operador de Sobel, resultando em duas filtragens distintas. Ambas contribuições são combinadas em uma imagem binária exemplificada na Figura 3(b).

Como as faixas de transito tendem a ser majoritariamente verticais, as contribuições referentes a este eixo recebem um maior peso melhorando a detecção das faixas. Para garantir uma maior robustez ao método o resultado desta etapa é combinado com dois filtros de limiar de canais.

Estes são um filtro de liminar no canal R do espaço de cor RGB (*Red-Green-Blue*) e um filtro no canal S no espaço de cores HLS (*Hue-Saturation-Lightness*). O primeiro busca filtrar as informações presentes no espectro do vermelho em diversas intensidade de luz. O segundo impacta diretamente a detecção das faixas brancas e amarelas tornando a detecção robusta à variações de luminosidade, como em regiões de sombra ou alta intensidade de luz. Os resultados de ambos os filtros de limiares são sobrepostos em uma única imagem, exemplificada na Figura 3(c).

Conforme ilustrado no diagrama da Figura 2, são combinados os resultados obtidos em ambos os processos de filtragem, limiar de gradiente e limiar de canais, em uma única imagem binária. Esta imagem resultante é exemplificada na Figura 3(d).



(a) Imagem Original (b) FL de Gradiente

(c) FL de Canal (d) Imagem Binária

Figura 3. Exemplo do procedimento de filtragem e binarização de uma imagem.

3.2 Extração da Região de Interesse

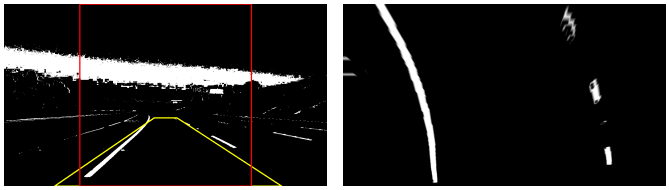
A curvatura e outros parâmetros geométricos das faixas são de grande importância para o sistema de controle do veículo inteligente. Assim, é proposta uma transformação de perspectiva que favoreça a determinação desses parâmetros como forma de melhorar as características de detecção do método. A transformação busca uma perspectiva aérea da via, onde as faixas sejam praticamente paralelas e não se aproximem em direção ao ponto de fuga, como na imagem original.

Esta transformação apresenta dois principais benefícios. O primeiro diz respeito à obtenção de uma região de interesse focada apenas na área das faixas, o que minimiza os ruídos externos e o tempo de processamento. Ao passo que o segundo, evita que a imagem seja afetada pela perspectiva, onde as faixas convergem para o horizonte em um ponto focal, favorecendo sua detecção.

Assim, uma perspectiva superior onde as faixas são aproximadamente paralelas em relação a câmera, aprimora a visualização da distância e da curvatura da via, tornando a detecção das faixas mais robusta e possibilitando que o método explore outras características das faixas, como sua distância horizontal e atributos geométricos.

Para realizar a transformação de perspectiva são necessários oito pontos na imagem, quatro para cada plano de transformação (Cao et al., 2019). Estes pontos são as arestas dos quadriláteros apresentados na Figura 4(a), em vermelho o plano desejado e em amarelo o plano inicial.

A Figura 4(b) mostra o resultado da etapa de determinação da região de interesse apresentando uma perspectiva aérea da região das faixas de transito.



(a) Delimitação das regiões de interesse para a transformação de perspectiva.

(b) Imagem com a perspectiva aérea da via.

Figura 4. Procedimento de transformação da perspectiva original para a perspectiva aérea.

3.3 Detecção das Faixas através de Slidding Windows e Histogramas de Intensidade

Conforme analisado em Reichenbach et al. (2018), a abordagem de *Slidding Windows* apresentou um baixo custo computacional em relação a outros métodos para detecção de faixas, como transformada de Hough, *Random Lines* e *Active Line Modeling*.

Além da baixa complexidade computacional, a abordagem apresenta índices satisfatórios de detecção e precisão quando comparada aos outros métodos. Algumas variações deste método, tais como *Fixed Windows* e análise da forma de onda baseada em histogramas, também foram avaliadas. Todavia, os resultados utilizando o método de *Slidding Windows* foram superiores. Portanto, os resultados e discussões apresentados no restante do trabalho baseiam-se somente neste método.

Este procedimento de detecção faz uso de regiões retangulares, denominados sensores virtuais, para realizar uma análise de densidade de pixel via histograma, detectando os pontos de maior probabilidade de conter informações das faixas. Estes sensores virtuais são deslocados através da imagem, da parte inferior até a parte superior, encontrando assim os pontos ao longo das faixas.

O método em questão pode ser descrito pelas seguintes etapas e está ilustrado na Figura 5:

(1) A primeira etapa consiste em encontrar os pontos iniciais das faixas na parte inferior da imagem. Estes pontos, são utilizados para inicializar o algoritmo de *Slidding Windows*. É obtida a intensidade de pixeis na metade inferior da imagem através dos cálculos dos valores acumulados de cada coluna da região. Isto resulta em um histograma de intensidade, onde os picos indicam a posição de maior probabilidade das faixas de transito. Dessa forma, o pico da parte esquerda da imagem representa o início da faixa da esquerda. O mesmo ocorre para a faixa da direita na seção direita da imagem.

(2) São definidos dois sensores virtuais e suas dimensões são previamente definidas segundo as características das faixas analisadas. Cada sensor é centralizado horizontalmente em um dos picos obtidos na etapa anterior e verticalmente na parte inferior da imagem. Então, são calculados novos histogramas de intensidade para cada um dos sensores, obtendo assim uma forma de onda similar a etapa anterior, porém restrita apenas a diminuta área observada pelo sensor virtual.

(3) Com base no histograma de intensidade obtido por cada sensor virtual, é calculada uma nova posição horizontal que centralize o sensor sobre a faixa de transito, esta posição é obtida através do histograma de intensidade da área do sensor virtual. Então, este sensor é deslocado verticalmente para cima e posicionado de forma centralizada na faixa, seguindo assim a sinalização das faixas de transito ao longo da imagem.

(4) A etapa do item 3 é repetida até que os sensores virtuais encontrem o topo da imagem, determinando assim o fim do processo de detecção. O número de iterações, N, é definido pela razão da alturas da imagem pela altura dos sensores virtuais. Experimentalmente foram utilizadas onze seções ($N = 11$), onde a visualização deste processo é apresentada nas Figuras 6(a), 6(c) e 6(e). Ao fim do processo, são obtidos dois conjuntos de pontos na imagem. Estes indicam os pontos de maior probabilidade de conter as faixas de transito.

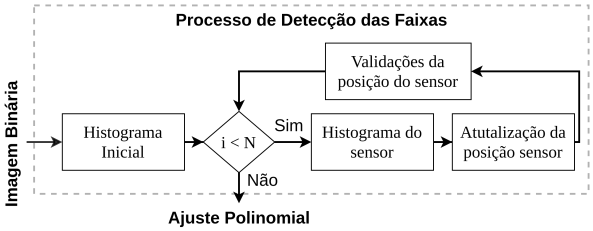


Figura 5. Diagrama do processo iterativo de detecção com *Slidding Windows* e histograma de intensidade.

Para garantir uma maior qualidade nas detecções, foram estabelecidos três critérios que auxiliam tanto na detecção quanto no rastreamento das faixas, uma vez que existem faixas descontínuas e interferências de segmentação nas imagens analisadas. Estes critérios são aplicados em todas as atualizações de posição dos sensores virtuais. O primeiro critério diz respeito à tolerância de deslocamento horizontal dos sensores. O segundo, à quantidade mínima de pixeis válidos em cada sensor e o último tem relação com a distância entre as faixas.

No primeiro, é estabelecido um limiar máximo para a movimentação dos sensores virtuais, evitando que possíveis interferenciais ou falsas detecções prejudiquem o posicionamento dos sensores. Este parâmetro é baseado no comportamento estrutural das faixas, uma vez que não há mudanças abruptas da sinalização nas vias.

Já o segundo estabelece um limiar mínimo de pixeis que devem ser detectados para que um sensor faça uma medida válida. Caso este liminar não seja alcançado, pode significar que o sensor esta mal posicionado ou que a faixa é descontínua. Portanto, para contornar estes casos, as coordenadas das detecções dos últimos quadros são armazenadas, permitindo a reutilização destas informações e possibilitando uma estimação melhor para a posição dos sensores. Isto é possível caso uma sequência de quadros das faixas de transito não apresente alterações súbitas, e, mesmo que haja, há o resguardo da tolerância horizontal do primeiro critério.

O último critério adotado considera a estrutura geométrica das faixas de transito. Na imagem em perspectiva aérea, é esperado que as faixas obtidas estejam aproximadamente

paralelas. Desta forma, é possível utilizar esta informação em conjunto com os critérios anteriores para melhorar a estimação da posição da faixa quando na ausência da detecção e também como validação da qualidade desta.

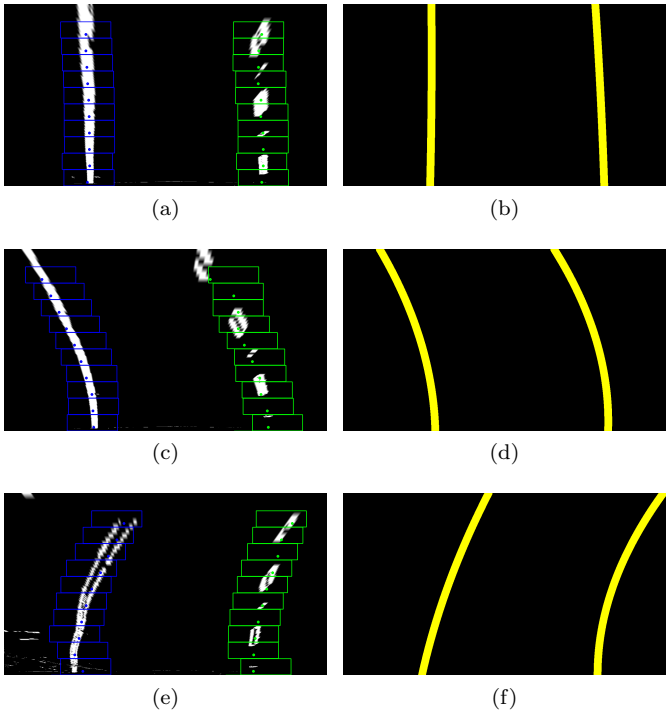
3.4 Ajuste Polinomial e Modelo de Faixa

O resultado do método de *Slidding Windows* são dois conjuntos de pontos que contém maior probabilidade de conter as faixas, conforme apresentado nas Figuras 6(a), 6(c) e 6(e). Estes são representados pelos pontos azuis e verdes internos aos retângulos nas imagens. Com base nestes pontos é possível realizar um ajuste polinomial que adéque as curvaturas das faixas de trânsito.

Para encontrar as faixas são utilizadas as coordenadas, horizontais e verticais, de cada uma das posições de maior probabilidade fazendo uma regressão para um polinômio de segunda ordem, uma vez que as curvaturas das vias são quase constantes e o polinômio quadrático é suficientemente preciso para o ajuste (Son et al., 2018).

As curvas em amarelo nas Figuras 6(b), 6(d) e 6(f) exemplificam os polinômios calculados a partir do conjunto de dados obtidos pelo método de detecção.

Após o cálculo do polinômio que ajusta a curvatura da faixa, é possível estimar diversos parâmetros importantes para o sistema de controle do veículo, tais como: a posição do veículo em relação ao centro da pista, a angulação das curvas e as manobras que devem ser realizadas no percurso.



(a) (b) (c) (d) (e) (f)

Figura 6. Diferentes informações sobre o processo de detecção. Em branco as informações segmentadas que representam as faixas. Os retângulos são os sensores virtuais do método *Slidding Windows* e em amarelo os polinômios obtidos através do ajuste do conjunto de dados detectados.

4. PROCEDIMENTO EXPERIMENTAL

4.1 Montagem Experimental

Com o intuito de obter um sistema prático condizente com as limitações de aplicações reais, as experiências foram realizadas na plataforma embarcada comercial Jetson Nano, que possui um processador ARM A57 de 1,43 GHz. O software que desempenha as tarefas de detecção de faixas foi implementado utilizando a linguagem de programação C++ versão 14. Para algumas operações foram utilizadas funções fornecidas pela biblioteca visão computacional de uso geral OpenCV na versão 4.1.1 sem otimizações de processamento paralelo e sem utilização de funções específicas de processador gráfico. O software proposto foi compilado utilizando a versão 9.3.1 do compilador GCC.

Para se obter resultados comparáveis e manter o ambiente controlado, foram utilizados quadros extraídos de vídeos pré-gravados. Estes foram obtidos de uma base de dados aberta disponível em Udacity (2017) que contempla cenas diurnas.

Os dados em questão possuem uma resolução de 1280×720 @ 30 fps (*frames per second*) e são adquiridos pelo sistema embarcado, simulando uma aquisição real. Além da resolução padrão da base de dados, foram geradas outras duas bases redimensionando os vídeos para as resoluções de 720×405 e 480×270, buscando obter mais cenários de operação para avaliar o desempenho do sistema proposto.

4.2 Métricas e Parâmetros de Desempenho

Para estabelecer as análises de qualidade do método proposto são utilizadas duas métricas padrão, a taxa de falsos positivos (TFP) e o erro médio relativo (EMR) da detecção. A TFP é a fração de quadros do total que não foram devidamente identificados. Estes são quadros onde as faixas não apresentam coerência entre si ou coerência com a realidade. O EMR é uma figura de mérito do erro.

A Equação 1 define formalmente a métrica TFP como a razão entre a quantidade de falsas detecções (NF) e quantidade de quadros detectados (ND).

$$\mathrm{TFP} = \frac{\mathrm{NF}}{\mathrm{ND}}. \tag{1}$$

Os valores de ND são obtidos comparando, ainda em tempo de execução, as faixas detectadas com as faixas reais. Em seguida, as falsas detecções são avaliadas de maneira manual para garantir a correta estimação deste parâmetro.

O erro médio relativo é definido pela Equação 3, que é a média dos erros quadráticos médios entre o conjunto de pontos detectados pelo método proposto (p_i) e os pontos do conjunto real ($\hat{p}_i$) em um determinado cenário. Além disso, a medida é normalizada pela dimensão horizontal (W) das imagens, garantindo uma medida adimensional para a comparação entre as diferentes resoluções.

$$\mathrm{EQM} = \frac{1}{N}\sum_{i=1}^{N}(\hat{p}_i - p_i)^2, \tag{2}$$

$$\text{EMR} = \frac{1}{M}\sum_{j=1}^{M}\frac{\text{EQM}}{W}. \quad (3)$$

Os pontos reais das faixas foram extraídos manualmente a partir de quadros sorteados aleatoriamente da base de dados.

5. RESULTADOS PRÁTICOS

Os resultados obtidos nesta experiência são analisados em duas principais perspectivas. Primeiramente sob o ponto de vista qualitativo, como forma de compreender a operação e as características do sistema proposto. Posteriormente, são apresentadas e discutidas as métricas dos tempos de execução e qualidade de detecção em diferentes cenários, como forma de avaliar o método proposto.

5.1 Análise e Discussão Qualitativa

O algoritmo proposto deve operar detectando as duas faixas principais da via onde o veículo se encontra, sendo estas contínuas ou descontínuas. Para representar os resultados obtidos, as faixas serão ilustradas em uma etapa de pós-processamento, utilizada apenas para gerar faixas na cor amarela como forma de visualização do resultado obtido, conforme exemplos das Figuras 7(d), 7(e), 7(f), 7(j), 7(k) e 7(l)

Esta representação das faixas é obtida através do ajuste polinomial de segunda ordem dos pontos detectados pelo método. Este ajuste busca obter as curvaturas da via, permitindo a detecção de distâncias mais longas e a determinação dos contornos da estrada, diferenciando-se dos métodos que adotam o modelo de faixa linear.

As Figuras 7(a), 7(b) e 7(c) são exemplos de detecção de curvatura. Essas são representações do processo de detecção das faixas, sendo apresentadas na perspectiva aérea. Os retângulos em azul e verde representam os sensores virtuais de cada uma das faixas, ao passo que os pequenos círculos internos aos sensores formam o conjunto de pontos detectados na faixa.

Dois problemas típicos que afetam a detecção de faixas de transito são: a variação da luminosidade em trajetos percorridos pelo veículo, seja por sombras ou por reflexos da luz, conforme exemplificado nas Figuras 7(j) e 7(l) e a variação da coloração do asfalto em regiões da via de trânsito, que podem ser vistas nas Figuras 7(f) e 7(k).

Desta forma, o sistema deve ser robusto o suficiente para garantir a detecção das faixas em tais condições. A combinação das técnicas de segmentação e *Slidding Windows* em conjunto com os as heurísticas apresentadas na Seção 3.3 aprimoram a detecção mesmo na presença destes fenômenos, garantindo um alto índice de detecção e acurácia.

As Figuras 7(g) e 7(h) permitem observar o impacto no processo de detecção gerado pela presença de sombras e variações de luminosidade. Enquanto nas Figuras 7(c) e 7(i) é observada a influência da mudança na coloração do asfalto.

A despeito das interferências visualizados no processo de segmentação da imagem, a estimação das posições iniciais através do histograma de intensidade e a utilização de informações de amostras de quadros anteriores em conjunto com os sensores virtuais garantem robustez ao método, alcançando os resultados satisfatórios como apresentados nas Figuras 7(j), 7(k) e 7(l)

Uma das principais fontes de imprecisão no sistema são distorções geradas pela transformação inversa de perspectiva em pontos mais distantes nas faixas descontínuas. Contudo, este efeito é mitigado pela reutilização de informações de quadros anteriores e da relação de distância entre este ponto e seu correspondente na outra faixa. Isto, pois não há variações significativas na curvatura das faixas e na posição relativa do veículo entre dois quadros consecutivos. Este efeito pode ser melhor visualizado em vermelho nos dois quadros consecutivos apresentados nas Figuras 7(a) e 7(b), onde mesmo na ausência de pixeis brancos foi possível seguir a tendência da curvatura.

5.2 Análise e Discussão Quantitativa

Conforme apresentado na Seção 4, a experiência foi realizada em um sistema embarcado convencional. Foram utilizados três diferentes cenários de resolução de imagem para obter uma análise mais ampla sobre o método.

O vídeo utilizado como entrada do sistema possui um total de 1271 quadros, sendo todos processados pelo método. Tanto a quantidade de detecções realizadas (e consequentemente a TFP) quanto a EMR destas detecções foram calculadas para cada um destes quadros e para cada uma das três resoluções distintas.

Estes resultados quantitativos típicos coletados durante a execução da experiência estão disponíveis na Tabela 1, onde R_1, R_2 e R_3 representam respectivamente as resoluções 1280×720, 720×405 e 480×270.

Tabela 1. Desempenho obtido sobre a base de dados para diferentes resoluções

	Medidas		
Índice	R_1	R_2	R_3
Quadros Analisados	1271	1271	1271
Detecções	1234	1202	1129
TFP (%)	2,14	5,43	10,46
EMR (10^{-3})	8,05	11,36	12,90

Analisando a porcentagem de TFP para cada uma das resoluções é possível identificar seu impacto na quantidade de detecções. Enquanto na resolução R_1 o algoritmo apresentou uma alta taxa de detecção, falhando em apenas 2,14% dos quadros totais, na resolução mais baixa essa perda foi cinco vezes maior.

Ao analisar a qualidade das detecções realizadas pelo método, foi possível observar que não somente a quantidade de quadros detectados é maior em altas resoluções, como também os erros dessas detecções são menores.

Para avaliar a qualidade do método de detecção em diferentes cenários de operação, a determinação do EMR foi separada em situações específicas: a detecção de faixas contínuas e descontínuas, e vias com e sem curvaturas. Estas informações exprimem de forma mais clara e objetiva as métricas de qualidade de detecção do método proposto e estão sumarizadas na Tabela 2.

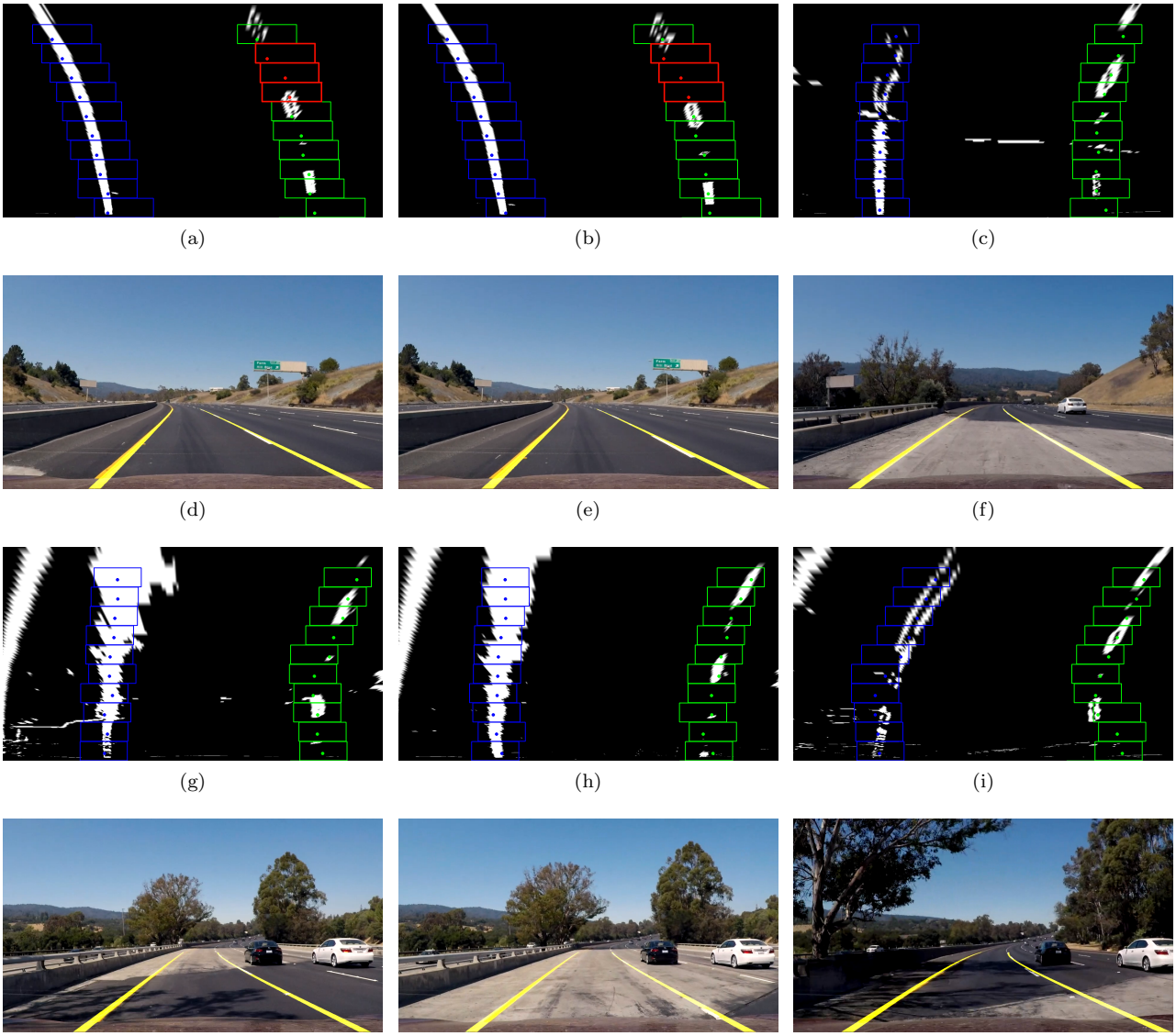


Figura 7. Resultados típicos obtidos através do algoritmo proposto. As imagens (a), (b), (c), (g), (h), e (i) apresentam a etapa de detecção das faixas, ilustrando os sensores virtuais através dos retângulos. O conjunto de pontos descreve a posição dos segmentos de faixa identificados nestes sensores. Em (d), (e), (f), (j), (k) e (l) são apresentadas as representações das faixas detectadas através do ajuste polinomial dos dois conjuntos de pontos obtidos na detecção das faixas da esquerda e da direita.

Tabela 2. EMR específico em diferentes situações para as diferentes resoluções ($\times 10^{-3}$).

	Contínua		**Descontínua**	
	Reta	**Curva**	**Reta**	**Curva**
R_1	3,09	11,89	4,80	12,43
R_2	3,65	13,46	9,28	19,09
R_3	5,33	13,86	12,8	19,61

Na Tabela 2 é possível notar que não somente a resolução impacta na detecção, mas também o tipo de sinalização da faixa e sua curvatura. A melhor precisão na detecção ocorre em cenários onde as faixas são contínuas e com raios de curvatura longos ou retilíneas.

Dado que este trabalho propõe a implementação de um algoritmo de detecção de faixas voltado para aplicações reais em uma plataforma embarcada, é necessário avaliar os tempos de execução do método proposto para realizar uma análise de viabilidade de operação.

Deste modo, foram examinados os tempos de execução de cada uma das etapas do método proposto no ambiente embarcado, com o intuito de levantar os custos destas operações e expor as etapas com maior impacto computacional, estes dados estão dispostos na Tabela 3. Assim como na análise anterior, foram tomadas as medidas individualmente em cada cenário de resolução.

Analisando os tempos para cada operação do método é evidente que o maior tempo de processamento é despendido nas tarefas de filtragem da etapa de segmentação de imagem. Ou seja, as operações de visão computacional são as operações mais custosas no método proposto. Contudo, na experiência realizada estas operações são desempenhadas

Tabela 3. Porcentagem do tempo de processamento para cada uma das operações em diferentes resoluções.

	Tempo (%)		
Operação	R_1	R_2	R_3
FL de Canais	17,06	20,63	35,60
FL de Gradiente	46,14	52,53	20,62
Região de Interesse	20,91	15,86	32,20
Histograma Inicial	7,09	4,64	4,80
Slidding Windows	8,60	5,84	5,36
Ajuste Polinomial	0,18	0,50	1,42
Total (ms)	72,913	26,498	11,555

por uma biblioteca de uso geral, não otimizada para aplicações de alto desempenho e aplicações embarcadas. Além disso, não foram utilizados todos os recursos disponíveis na plataforma embarcada, como o processamento paralelo ou as operações inerentes ao processador gráfico.

6. CONCLUSÃO

Em determinadas condições, o algoritmo proposto neste trabalho é passível de execução em tempo real e sem otimizações em uma típica unidade de processamento embarcada atual. Os resultados indicam que o algoritmo possui robustez para a detecção de faixas na presença de curvas e em ambientes com variação de luminosidade.

A combinação de filtros de limiar para a obtenção de uma imagem binária da via foi fundamental para a alta taxa de detecção observada. Esta técnica garantiu robustez à variação de luminosidade e coloração do asfalto. Entretanto, seu processamento demanda uma carga computacional elevada, fato que pode ser contornado através do uso de bibliotecas de visão computacional otimizadas e do uso de processamento gráfico dedicado.

A adoção do método de *Slidding Windows* em conjunto com o ajuste polinomial apresentou resultados interessantes para aplicações embarcadas, aliando baixos erros relativos na detecção do conjunto de pontos da faixa e baixos tempos de execução.

As análises neste trabalho indicam a factibilidade do método proposto. Este apresentou alta taxa de detecção de quadros e baixos erros, mesmo operando em diferentes situações, tais como: vias com e sem curvatura, presença de sombras, variação luminosidade e mudança de coloração do asfalto. Além disso, o método é executado em tempo real em um sistema embarcado convencional, mesmo sem otimizações, e operando abaixo dos 30 ms para as resoluções menores.

Vale observar que apesar dos resultados promissores este trabalho limitou-se a um banco de imagens com cenários diurnos e suas variações. Desta forma, ambientes noturnos, ou com baixa luminosidade, devem ser incluídos em futuras análises. Adicionalmente, o método apresentado deve ser avaliado quando executado de forma paralela, maximizando a execução em sistemas com múltiplos núcleos. Por fim, os resultados também podem ser aprimorados para operação em um chip gráfico e com a otimização das funções de visão computacional, o que seria mais eficiente em aplicações embarcadas.

REFERÊNCIAS

Bar Hillel, A., Lerner, R., Levi, D., and Raz, G. (2014). Recent progress in road and lane detection: A survey. *Machine Vision and Applications*, 25. doi:10.1007/s00138-011-0404-2.

Bengler, K., Dietmayer, K., Farber, B., Maurer, M., Stiller, C., and Winner, H. (2014). Three decades of driver assistance systems: Review and future perspectives. *Intelligent Transportation Systems Magazine, IEEE*, 6, 6–22. doi:10.1109/MITS.2014.2336271.

Borkar, A., Hayes, M., and Smith, M.T. (2012). A novel lane detection system with efficient ground truth generation. *IEEE Transactions on Intelligent Transportation Systems*, 13(1), 365–374.

Cao, J., Song, S., Xiao, W., and Peng, Z. (2019). Lane detection algorithm for intelligent vehicles in complex road conditions and dynamic environments. *Sensors*, 19, 3166. doi:10.3390/s19143166.

Kum, C.H., Cho, D.C., Ra, M.S., and Kim, W.Y. (2013). Lane detection system with around view monitoring for intelligent vehicle. *Proc. International SoC Design Conference*, 215–218. doi:10.1109/ISOCC.2013.6864011.

Lee, C. and Moon, J. (2018). Robust lane detection and tracking for real-time applications. *IEEE Transactions on Intelligent Transportation Systems*, 19(12), 4043–4048.

Li, X., Fang, X., ci, W., and Zhang, W. (2014). Lane detection and tracking using a parallel-snake approach. *Journal of Intelligent and Robotic Systems*, 77. doi:10.1007/s10846-014-0075-0.

NHTSA (2015). Critical reasons for crashes investigated in the national motorvehicle crash causation survey. Technical report, Washigton, DC, USA.

Reichenbach, M., Liebischer, L., Vaas, S., and Fey, D. (2018). Comparison of lane detection algorithms for adas using embedded hardware architectures. In *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 48–53.

Son, J., Yoo, H., Kim, S., and Sohn, K. (2014). Real-time illumination invariant lane detection for lane departure warning system. *Expert Systems with Applications*, 42. doi:10.1016/j.eswa.2014.10.024.

Son, Y., Lee, E., and Kum, D. (2018). Robust multi-lane detection and tracking using adaptive threshold and lane classification. *Machine Vision and Applications*, 30. doi:10.1007/s00138-018-0977-0.

Transp., S.O.D. (2010). Reported road casualties great britain 2009. Technical report, London, U.K.

Udacity (2017). Carnd repository - vehicle detection. `https://github.com/udacity/CarND-Vehicle-Detection/blob/master/project_video.mp4`. Acessado: 16.06.2020.

Wang, Y., Teoh, E., and Shen, D. (2004). Lane detection and tracking using b-snake. *Image and Vision Computing*, 22, 269–280. doi:10.1016/j.imavis.2003.10.003.

Yousaf, R.M., Habib, H.A., Dawood, H., and Shafiq, S. (2018). A comparative study of various edge detection methods. In *2018 14th International Conference on Computational Intelligence and Security (CIS)*, 96–99.

Zhou, S., Jiang, Y., Xi, J., Gong, J., Xiong, G., and Chen, H. (2010). A novel lane detection based on geometrical model and gabor filter. In *2010 IEEE Intelligent Vehicles Symposium*, 59–64.

Apêndice B

Experimento Paralelo (ITS) - Artigo Submetido para IEEE Transaction on Intelligent Transportation Systems

Título: CUDA-based Real-Time Ego-Lane Detection in Embedded Systems

Autores: Guilherme Brandão da Silva, Daniel Strufaldi Batista, Marcelo Carvalho Tosin, Décio Luiz Gazzoni Filho, e Leonimer Flávio de Melo.

Periódico: IEEE Transaction on Intelligent Transportation Systems

Data: 15 Outubro de 2021

CUDA-based Real-Time Ego-Lane Detection in Embedded Systems

Guilherme Brandão da Silva, Daniel Strufaldi Batista, *Member, IEEE,* Décio Luiz Gazzoni Filho, Marcelo Carvalho Tosin and Leonimer Flávio de Melo

***Abstract*—Ego-lane detection is one of the main tasks for the development of intelligent vehicles. The proposed method uses a heterogeneous approach based on the extraction of features from an aerial perspective image. It aligns high detection rate and real-time requirements in embedded systems, combining methods to enhance markings and remove noise. The implemented algorithm fulfills the requirements for the accurate detection of ego-lanes. Practical experiments using TuSimple's image datasets were conducted in NVIDIA's Jetson Nano embedded computer. The method detected up to 97.9% of the ego-lanes with an accuracy of 99.0% in the best-evaluated scenario. The system performed detections at a rate greater than 300 fps.**

***Index Terms*—Ego-lane detection, real-time embedded system, CUDA, optimization, heterogeneous computing.**

I. Introduction

MOST traffic accidents can be attributed to drivers or to a person's response time [1], [2]. Seeking a reduction in traffic fatalities and injuries caused by accidents, considerable investments in research have been made regarding driving assistance systems (ADAS). In general, these systems have a network of integrated sensors as their source of information, most commonly cameras, due to the amount of information provided at a relatively low-cost [3], [4]. Such assistance and safety devices have many subsystems acting together to prevent as many accidents as possible.

Lane identification and detection is a mandatory subsystem for intelligent vehicles technology. Among other systems, it aids the driver in some dangerous circumstances, known in cars as the Lane Departure Warning System (LDWS), a feature that boldly increases the driver safety [5].

An obstacle to lane detection in practical implementations is that most of the literature around the topic disregards the computational complexity of its algorithms, as they do not aim to execute their methods in embedded systems. Therefore, it limits or renders it impossible to reproduce such methods in real-time applications. This paper focuses directly on this aspect, proposing a lane detection method compatible with modern embedded system processing capability.

The proposed method was initially approached on [6]. In that work, the implemented ego-lane detection was a non-parallel algorithm executed in an embedded system.

Now, we propose an enhanced solution using heterogeneous computing and CUDA (Compute Unified Device Architecture) to achieve a real-time execution in a typical embedded system. Our method is optimized to take advantage of a highly parallel solution running on a GPU. Furthermore, it has some improvements compared to the originally proposed.

The proposed method has three steps: image preparation, feature extraction, and lane estimation. The first reduces the image to a monochromatic color space scale and applies an inverse perspective mapping. This removes camera position distortions and obtains a suitable region of interest, where the lanes tend to be parallel and with constant width. Next, an adaptive filter is applied to suppress unnecessary image information and highlight the traffic lane markings. Finally, a sliding window method obtains the positions of the lanes iteratively.

Benchmarks were performed using a third-party database to assess the accuracy, the false positive, and the matched lane rate of the proposed method. The TuSimple database provides several road scenes and their ground truths. In addition, the runtime of the improved implementation proposed is compared with a non-parallel implementation and also with an implementation using a common GPU compiled library [7]. Finally, the performance of selected works is summarized and compared to our method.

The main contribution of this work is to propose an algorithm for lane detection suitable for use with inexpensive sensors, such as a monocular camera, which is capable of running efficiently in real-time in a conventional embedded system with a high detection rate by using heterogeneous computing based on CUDA.

A. Related Work

The images captured by cameras are usually in color. To simplify the detection problem, some lane detection methods convert the images to gray scale [8]–[10].

In addition, predefined regions of interest are used to remove external noise and reduce the amount of information processed. In [11] a detection algorithm was proposed based on the use of two regions of interest, a rectangular and a Λ shaped one. [12] divides the region of interest into finite sections to make the traffic lanes tracking simpler, while [13] makes use of estimators to get straight segments from these regions.

It is also possible to determine a region of interest by using transforms to remove perspective distortions. Some works use the inverse perspective mapping (IPM) to obtain an aerial panoramic image of the road (Bird's Eye View, BEV) seeking to obtain an image where the traffic lanes tend to be parallel and of constant width [10], [14], [15].

All the authors are with the department of Electrical Engineering of the State University of Londrina, Paraná, Brazil.
This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil (CAPES) - Finance Code 001.

A temporal blurring technique was proposed by [16], [17], performing temporal integration of the previous frames to improve quality of the degraded and worn traffic stripes and increase detection confidence.

Most of the work done on traffic lane detection are based on feature extraction [11]–[14], [18]. In this type of algorithm, the image analysis seeks to detect the gradients, color patterns, and other information present in the pixels of the image, in order to recognize the traffic stripes. To ensure greater robustness to variations of luminosity, [19] proposes an adaptive luminance threshold filter to highlight the lanes and filter noise.

In [20], several methods for lane detection in embedded systems are evaluated. [15], [20], [21] highlight the sliding window method as an efficient method for detecting traffic lanes, achieving good accuracy and reduced computational complexity. In addition, it becomes possible to obtain the curvatures of the lanes.

Other works such as [22]–[24] highlight the gains made by parallelizing computer vision operations and the use of hybrid computing to solve image processing problems in real-time applications utilizing CUDA.

Recently, deep-learning, convolutional neural networks and pixel-level classification networks have been used for lane detection [25]–[28]. According to [21], results show that deep-learning-based methods increase detection rates and recognition accuracy, but have limitations for real-time implementation. Specific computational capabilities are required for real-time implementation of these techniques, which is rarely found in embedded systems.

II. Proposed Embedded Lane Detection Technique

Figure 1 summarizes the proposed method showing its three main steps: (1) image preparation, (2) feature extraction, and (3) lane estimation. It's important to emphasize that the algorithms within each step are highly parallelizable and thus suitable for CUDA implementation, achieving higher energy efficiency and lower runtimes.

The first step is responsible for treating the image to obtain a better representation of the road. Initially, the image is reduced to a monochromatic color space scale, resulting in a single-channel image where each pixel is represented by 8 bits. Then, an IPM obtains a region of interest that mitigates noise and distortions, providing a BEV image [10]. Thus, we obtain in this step a gray-scale image in an aerial perspective where the traffic lanes tend to be parallel and of constant width. A temporal integration technique is then used on the frames, increasing the quality of degraded lanes and further removing more unwanted information [17]. This last technique relies on the fact that lanes do not vary considerably in subsequent frames.

In the second step, we use two methods to highlight lane markings on the road and to reduce noise and unwanted information. Given the color variability of lanes, the algorithm uses a threshold adaptive filter based on the image luminance [19]. It uses the brightness difference between the lane marks and the road to remove any unnecessary information. Additionally, it uses a simple edge detector to obtain the vertical markings of the lanes [8]. Then, both feature maps are combined to obtain a version of the image that contains only the stripes' markings.

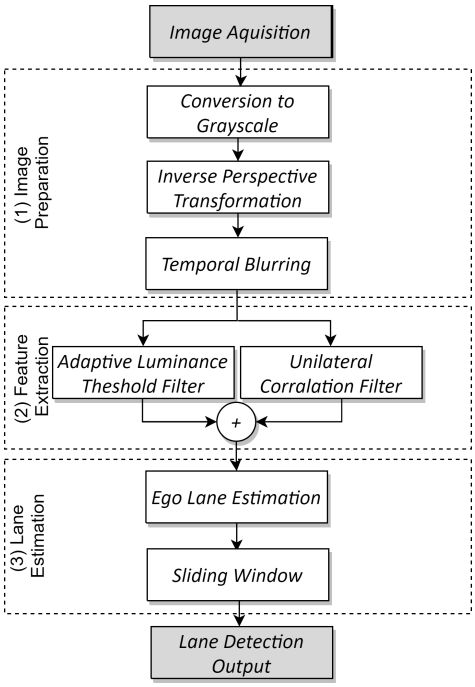


Fig. 1. Simplified diagram of the proposed ego-lane detection method.

The last step will estimate the lane position by identifying its guiding straps, and it is divided into two parts as shown in Figure 1. Initially, a rectangular region at the lower portion of the image is set based on a pixel intensity histogram, so the region has the highest probability to include the lane's traffic stripes. From this point, the system slides the window upward and laterally repositions the window, so as to maximize the probability of finding the stripes [21]. In this process, the window runs vertically over the image, estimating the position of the lane markings. The sliding window method obtains two sets of markings representing the lane in the BEV image.

III. Acceleration of algorithms using CUDA

A. Architecture of a CUDA-based GPU

Unlike a traditional Central Process Unit (CPU), a Graphics Processing Unit (GPU) has an architecture that can effectively accelerate image processing and computer vision algorithms [24]. CUDA (Compute Unified Device Architecture) is a programming interface developed by NVIDIA to facilitate software development on GPU, and which has found wide use in the acceleration of image processing algorithms.

A modern GPU has Stream Multiprocessors (SM), which are independent units of decoding, instruction fetch, and execution [29], [30]. SM are composed of dependent units, called Streaming Processors (SP), which are the basic units for thread execution on GPUs. These units are commonly called CUDA cores on NVIDIA chips.

The Maxwell architecture [31] features an enhanced SM. It is partitioned into four distinct processing blocks of 32

CUDA cores each (128 CUDA cores per SM). Each block has its features for instruction scheduling and buffering. This configuration provides a warp size alignment, making it easier to use and improving efficiency.

Shared memory allows threads from the same block to act cooperatively, facilitating the reuse of resources and mitigating transfers between slower off-chip memories [29], [32]. Heterogeneous programming using CUDA is split between host and device code. Each thread that runs on the device has a unique identifier, called `threadIdx`. A set of thread blocks form a thread grid. A kernel can run simultaneously in all threads of a grid. Each thread has its local private memory, while each block has a shared memory. All threads within that block can access the shared memory. Also, threads in a block have the same lifecycle as the block [32].

An NVIDIA Jetson Nano board is used to evaluate all the algorithms and to validate the method using a GPU architecture. The Jetson Nano is a commercial single board embedded computer composed by a quad-core ARM Cortex A57 CPU and an NVIDIA 128 core (1 SM) Maxwell architecture GPU. To run the method in real-time in such an embedded system, the algorithm must perform using the maximum GPU resources as possible. Thus, several CUDA optimizations were made to improve efficiency.

The remainder of this section will discuss the usual optimizations applied to kernels (functions that are executed n times in parallel by n CUDA threads) that are part of the proposed method. The goal is to identify and mitigate the implementation bottlenecks. The following sections present detailed and specific CUDA optimizations. They classify into three types of implementation bottlenecks: 1) memory bandwidth, 2) instruction latency, and 3) instruction throughput.

The NVIDIA Visual Profiler, an NVIDIA profiling tool used to optimize CUDA applications, is used to help analyze kernel execution metrics, providing important feedback for optimization.

B. Memory optimization

In general, these optimizations seek to increase the memory throughput. They aim to minimize global memory accesses while employing the most suitable access pattern to realize memory coalescing, thus minimizing memory transactions.

One of the techniques is the use of shared memory for redundant and non-coalescing accesses. Shared memory is usually faster than global memory, i.e. has lower latency and higher bandwidth. The use of shared memory is especially profitable for unaligned data accesses.

It is also important to use the best access pattern according to each situation, ensuring coalesced access to global memory. Also, some optimizations such as data transposition from array of structures (AoS) to structures of arrays (SoA), the use of padding, and changes to the parallelization strategy were required.

C. Latency optimization

Latency optimization aims to guarantee that there are as many threads as possible in execution to mask the latency effect. The block sizes must always be a multiple of the warp size to ensure that an entire warp performs the same operation, achieving higher occupancy and latency hiding. Grid size selection is crucial and was done empirically by comparing the performance of different sizes using the profiler.

Additionally, grid-strided loops are used for latency hiding. It provides an efficient solution to avoid monolithic kernels and ensure a higher occupancy.

D. Instruction optimization

Some of the kernels are compute-intensive, which can cause a bottleneck in instruction execution. Therefore, it is necessary to organize and optimize their operations to avoid that. The main objective of this type of optimization is to reduce the number of total instructions required to carry out an operation. One of the techniques adopted is the use of operations with high throughput, such as SIMD (Single Instruction, Multiple Data) and SIMT (Single Instruction, Multiple Threads) warp-level primitives. Those are instructions that perform operations with larger data blocks. Several operations were vectorized in a grid-strided loop using the data arrangement to ensure the highest execution efficiency. Also, we seek to avoid branch divergence in warps and reducing bank conflicts.

IV. Image Preparation and Inverse Perspective Mapping

There are an unlimited amount of scenarios for driving a car. Naturally, there is plenty of content irrelevant to the problem in the captured image. Therefore, it is a common strategy to pre-process the input image followed by the feature extraction technique [33]–[35]. This processing step has been applied to images to reduce noise and external distortions, providing a better region of interest and reducing the amount of processed data. Doing so also increases the reliability of the input data [35]. This section details the pre-processing employed in our technique, which includes the image preparation, the IPM transformation, and the temporal blurring.

A. Image Preparation

Transforming the three-color channel RGB image to grayscale reduces its size to a third of the original and minimizes the brightness interference over the three-channel image. Furthermore, it simplifies the problem and algorithms when seeking embedded processing [9], [10], [22], [35]. Equation (1) shows the transformation required to convert from RGB to grayscale.

$$I(x, y) = R \times 0.299 + G \times 0.587 + B \times 0.114 \qquad (1)$$

Processing the gray-scale conversion equation has a higher degree of parallelization. Given that all operators are simple, the throughput is limited only by the memory bandwidth. Patterns of 16 bytes are read in each loop and loaded to the shared memory to ensure better efficiency when accessing the main memory, where the access is strided without performance loss. These patterns are converted from an AoS to SoA,

such as seen in section III-B. Furthermore, all operations are vectorized, increasing the number of pixels processed in each iteration. The conversion to gray in 16 pixels is computed through structures that allow isolated access to this information. This procedure will be used frequently in the other kernels of the project.

B. Inverse Perspective Mapping

In the lane detection problem, the camera typically faces forward and causes a perspective distortion. Such distortion shows the lane markings tending to a vanishing point, disobeying a few assumptions. The main problem is that the traffic lane stripes lose the characteristic of being parallel to each other so that the two markings that define a lane became close as they are away from the camera. The lane curvature estimation becomes more complex and less predictable [36].

These assumptions are inherent characteristics of transit lanes, and computer vision algorithms can use these parameters to fine-tune their operation. To re-establish the characteristics of the road several works [10], [13], [15], [37] perform a change of perspective to obtain an aerial image (BEV) through the inverse perspective mapping (IPM). Figure 2 shows a diagram of the perspective change in this processing.

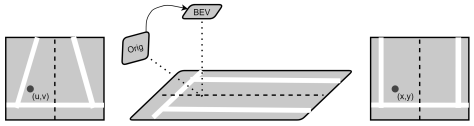


Fig. 2. Simplified diagram of the perspective change effect from the original front view to BEV.

This operation is a projective transformation with eight degrees of freedom that can be constrained by several elements present in the image. The eight degrees of freedom are obtained by selecting 8 points on the image, which allows solving the homography directly. Homography is the transformation relation between two planes formed by the corresponding 8 points, with 4 points in each plane [38]. An example of these points is shown in Figure 3a. This homography translates into a third-order matrix, responsible for transforming a point in the original image into a point in the BEV perspective image, as shown in Figure 3b. There are two methods to obtain this matrix: *1)* dynamically during execution [39] and, *2)* using a pre-computed static manner [9], [15], [40]. The dynamic way has greater computational complexity and depends on a specific trigger sequence. The static method is more susceptible to variations but has a minimal computational cost, thus making it suitable for real-time embedded processing.

Using the homography matrix H it is possible to obtain the pixels of the transformed image $I(x, y)$ as a function of the input image's pixel $O(x, y)$ through (2) [9], [15].

$$I(x,y) = O\left(\frac{H_{11}x + H_{12}y + H_{13}}{H_{31}x + H_{32}y + H_{33}}, \frac{H_{21}x + H_{22}y + H_{23}}{H_{31}x + H_{32}y + H_{33}}\right) \tag{2}$$

Equation (2) can produce float points coordinates that do not belong to the image. The workaround is to calculate the original image pixels (I) to each of the transformed ones (O). This approach provides an output image without interference, and it avoids the calculation of about 45% of the total pixels because of the transformation characteristics, e.g., the two black triangles in Figure 3b. The operations were vectorized to ensure a higher throughput and thus improve performance. Besides, some terms of the equation are constant for a couple of pixels, which allows to keep those calculations in the registers and reuse them in other ones.

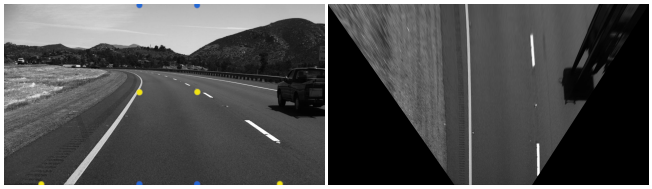


Fig. 3. A typical example of perspective change. (a) Marking the eight points used in the original image. (b) BEV image resulting from IPM.

C. Temporal Blurring

The temporal blurring improves the lane markings by making them more robust and obscuring moving objects, reducing possible sources of external noises. One of the expected effects is that lanes with discontinuous straps or worn ones will have a more uniform and continuous appearance [16], providing better detection quality [6].

The technique exploits the assumption that lanes do not change abruptly in a short period (in this case, in subsequent frames), especially when compared to other road objects (vehicles, pedestrians, etc.). Thus, an amount of N previous frames are temporally integrated to increase the confidence of the markings. By combining these images, the marks are overlapped, increasing the amount of information in the lanes. An example of the temporal blurring effect is shown in Figure 4.

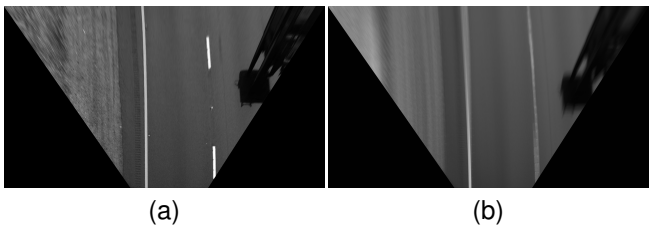


Fig. 4. An example of the temporal blurring effect. (a) An input BEV image of the method. (b) Result of the temporal blurring effect for N=20.

The temporal frame integration is performed by means of (3) [17].

$$I_{avg} = \sum_{i=0}^{N} \frac{I(n-i)}{N}, \tag{3}$$

where I_{avg} is the resulting image with N previous frames combined.

V. Feature Extraction

The feature extraction procedure seeks to obtain an image with highlighted markings features and mitigating further useless information for the detection method. The method assembles feature maps that filter different characteristics of the marking. The maps are then combined to obtain a better result.

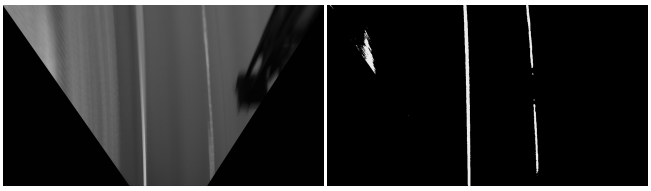
A. Adaptive threshold feature map

The lane markings tend to be the brightest elements in the image or, at least, brighter than the lanes themselves. Therefore, the first feature map highlights the markings based on the difference between pixel values and the average luminance. Fixed thresholds filters are inadequate due to ambient variation, such as asphalt and marking colors. The solution is to use an adaptive threshold filter based on the average luminance of the region of interest [17], [19].

The adaptive threshold calculates an average of the intensity (L_{avg}) of each pixel to the given region of interest. Based on that value, it estimates a superior (T_U) and inferior (T_L) threshold, as shown in (4). Each pixel of the image is then binarized using (5). Figure 5b presents an example of this feature map, where white traces represent the possible traffic lane markings.

$$T_L,\ T_U = \begin{cases} 60,\ 220 & \text{if } 0 \leq L_{avg} \leq 25 \\ 115,\ 234 & \text{if } 25 < L_{avg} \leq 40 \\ 125,\ 240 & \text{if } 40 < L_{avg} \leq 70 \\ 135,\ 250 & \text{if } 70 < L_{avg} \leq 100 \\ 145,\ 255 & \text{otherwise} \end{cases} \tag{4}$$

$$I(x,y) = \begin{cases} 1 & \text{if } T_L \leq I(x,y) \leq T_U \\ 0 & \text{otherwise} \end{cases} \tag{5}$$



(a) (b)

Fig. 5. An example of the adaptive threshold feature map. (a) Typical pre-processed image. (b) Feature map obtained by the adaptive threshold method.

To implement this procedure there are two kernels. The first calculates the average pixel intensity of the image, while the second performs the binarization using (5) and the threshold values.

Considering the image already allocated within the GPU memory, the kernel uses a grid-stride loop to maximize the reading bandwidth. Furthermore, batches of 128 bits are read and then written at the shared memory. The calculation of the sum of the pixels is vectorized to compute 16 pixels simultaneously.

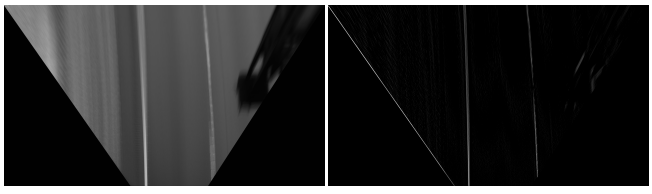
The grid-stride loop is done by summing all input bytes of the data. After the operation, the threads from this block are reduced to a single value. This is done using the SIMT operator (`__shfl_down_sync`) available at the CUDA (synchronous reduction to each GPUs warp). At last, we make the atomic sum of the accumulated values at each warp, obtaining the total sum of the pixels.

The second kernel is responsible for making the image binarization using T_L and T_U values. It has a low computational cost as previous kernels that are bandwidth bound. To increase the throughput it uses a set of SIMD instructions. Those are logical operators like setting if greater than (`__vsetgtu4`), setting if lower than (`__vsetltu4`), and unsigned minimum (`__vminu4`). The operators run in groups of 4 bytes each per iteration of the grid-stride loop.

B. Unilateral correlation feature map

Conventional edge detectors are not suitable for real-time embedded system implementation due to their computational cost. This work uses a unilateral correlation filter [8], [41] as solution. This type of filter has lower computational complexity and it is based on the convolution of the image with a separable convolution kernel.

The implemented filter is a third-order matrix that extracts a single type of edge in the image. As lane markings are almost vertical in the BEV image, the filter is designed to extract vertical edges, detecting horizontal variation of pixels. Figure 6 presents an example of the feature map obtained by processing the unilateral correlation filter.



(a) (b)

Fig. 6. Example of unilateral correlation filter operation. (a) Typical pre-processed image. (b) Feature map obtained by the unilateral correlation filter method.

The filter also has the property of being separable. Therefore, it is possible to replace the 3×3 convolution of the filter by two convolutions using the vectors $[1, 2, 1]$ and $[-1, 0, 1]$. This property decreases the computational cost of the operation from $O(m \times n)$ to $O(m + n)$.

The kernel implemented for this part uses several optimizations aforementioned, such as the grid-stride loop, shared memory allocation, and operation vectorization. However, its implementation uses the tilling pattern, which are two-dimensional structures of threads [42], exploiting the intrinsic characteristics of the operation. This kernel uses sub-matrices to carry large amounts of information from the main to the shared memory, allowing strided access for calculation. Furthermore, due to the filter's features, some of the values can be pre-computed and reused in other operations, reducing the number of operations performed in each loop.

VI. Lane Detection

The lane detection step has two stages. First, it detects regions most likely to contain the markings, and then, it estimates the points belonging to each of the markings. The first uses a column histogram algorithm to obtain the regions with the highest pixel density. The second uses these regions to initiate an iterative sliding window method that detects the pixels belonging to the lane markings.

A. Column Histogram

The column histogram operation determines the regions with the highest probability of containing the traffic lanes based on the intensity of the pixels in each column [20], [21]. This is achieved by accumulating the value of each column of the feature map (Figure 7a). The peaks obtained in this histogram are considered as candidates, such as the example of Figure 7b.

Fig. 7. Example of column histogram operation. (a) Typical input feature map. (b) Column histogram obtained for the Figure 7a.

The kernel responsible for the column histogram operation adopts a two-dimensional block to launch the threads and uses some of the strategies presented before. The shared memory stores the input data and allocates the positions and values used across the process. Operations were parallelized to improve the throughput and warp-level primitives are employed. Furthermore, the operation is symmetric and so it is possible to launch two similar blocks. Each one is responsible for processing one side of the image, making use of the problem's geometry to ensure greater efficiency. Another important aspect is that the kernel obtains only the initial position of the markings. Therefore, only the inferior half of the image is analyzed.

The warp-level primitive `__shfl_down_sync` is used twice to obtain the maximum value in each block. The first time it seeks the peaks of each warp. Then we use it again to get the global peaks. The first block obtains the peak referring to the left side of the image (left lane marking), while the second block obtains the one on the right.

B. Sliding Windows

To detect the lane stripes' positions, we group the regions with a higher density of pixels on the feature maps. The typical approach to obtain these positions is to process the entire image storing them. The issue with embedded execution is that such an approach is inefficient, as only a minor portion of the pixels contain lane markings information. This search can be optimized using the sliding windows method [15], [20], [21], which evaluates only the regions of the image that are more likely to contain the lane marks. Based on the points estimated by the column histogram, this method places windows at those positions and then runs vertically over that region. Each window computes the peak of that small region. At the end of the process, each peak describes a point of the lane.

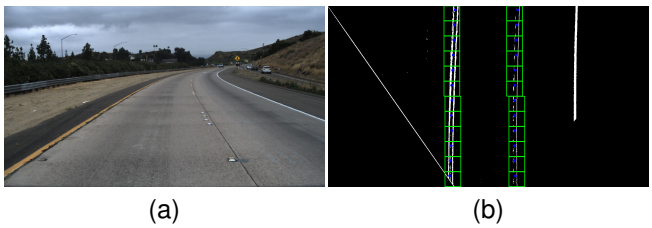


Fig. 8. Example of sliding windows operation. (a) Original image of the road. (b) Result of the sliding windows operation. In green the windows and in blue the points obtained in the process.

This kernel has some differences from previous ones. Instead of performing over the entire image, the operation is performed only on two regions using the geometry of the lanes. To improve the performance we launch a tri-dimension grid of threads exploring the inherent parallelism of the operation. The x and y-axis are the sizes of the windows, and the z-axis is the other windows on that same lane.

The global memory data is loaded into the shared memory to ensure strided access. A column histogram computes the peak for each window using warp-level primitives, similarly to the previous kernel. To improve this operation the size of the windows should fit an integer number of warps.

VII. Simulation Test Experiment

Results are divided into qualitative and quantitative analysis, as well as the execution time performance. The first subsection briefly discusses the setup and the dataset used during the tests, followed by the results and, at last, by a discussion and comparison with other works.

A. Test Environment and the Tusimple dataset

The test environment uses the NVIDIA Jetson Nano board and its GPU, mentioned in section III-A. For reference, the algorithms were developed in C++ language version 7.5.0 and the heterogeneous implementations were programmed using the CUDA 10.2 API.

The experiment uses images available at the TuSimple dataset. It has three subsets totaling 3626 video clips (72520 frames) with roads composed of annotated image frames of US highways. Each image has a 1280x720 resolution captured by a camera centered in the middle of the vehicle's windshield. The dataset consists of small scenes made of a varying number of 1-second clips. Each of these clips contains 20 frames each, and the last one having an annotation of the polyline types for lane markings. Each scenario may have different weather conditions and at various times of the day. Furthermore, they are taken on different types of roads and under arbitrary traffic conditions.

B. Qualitative analysis of ego-lane detection

We use different scenes from the dataset to analyze the algorithm's parts response comparing it for different implementations. Figures 9a, 9b, 9c, 9d, 9e, and 9f shows some examples. Each one has four subfigures that display the original figure (top left), the sliding window output (top right), the adjustment of the detected ranges in the BEV perspective (bottom left), and the final output (bottom right). Those specific scenes present to the reader the algorithm's response under different cases and types of roads.

Figures 9a and 9b show results from roads with distinct asphalt colors and with internal coloring variation. Figure 9c is an example of a very dark asphalt with worn lane markings. Even in this scenario, the feature extraction performance is satisfactory, and the algorithm performance is maintained. Other cases of degraded asphalts and worn markings are shown in Figures 9d and 9e. In such cases with precarious markings, the image preparation and the BEV perspective are fundamental for improving the detection.

The perspective transformation also reduces the complexity of lane detection with curvatures. Figures 9a and 9f are examples of that. Those results show the algorithm has satisfactory ego-lane detection even under the least favorable circumstances, including worn asphalts and markings, curvatures, and color variations.

The video in [43] illustrates the operation of our algorithm. The video shows the images available on the Tusimple dataset 0601. That dataset is composed of a sequence of 8200 frames assembled as a motion picture of 1-second clips. We split the image into quadrants: The upper left corner shows the original video. The upper right shows the preprocessed video by the method. The lower-left shows the resulting feature map with the windows (formed by the green rectangles) and the detected dots in blue. The last shows the original video with its frames overlapped by dots marking the estimated road stripes.

C. Algorithm performance

Table I shows the proposed method's operations execution times for three distinct implementations. The first column presents the execution times for the optimized heterogeneous implementation using CUDA. The second brings the times for an intermediate implementation, which uses the graphic chip through the OpenCV library and CUDA. The last shows the execution times for the embedded system's CPU implementation. The values obtained in each function do not consider the loading and storing times of the image. Table II summarizes the speed-ups of the proposed method in comparison with the other two.

The results show that the runtime of the heterogeneous implementation is significantly better than the others. The optimized algorithm is 25 and 140 times faster than the intermediate and the CPU methods, respectively. Naturally, the execution time gain varies depending on the algorithm's part. At specific parts, it is possible to observe that the proposed implementation achieves much higher performance, even compared to the version running on the GPU compiled in OpenCV.

TABLE I
ALGORITHM OPERATIONS' RUNTIMES CONSIDERING DIFFERENT TYPES OF IMPLEMENTATION (VALUES IN MILLISECONDS).

Operation	CUDA	OpenCV	CPU
Conv. grayscale	0.269	1.320	48.514
IPM	0.305	20.913	97.181
Temporal blurring	0.763	2.834	33.346
Adapt. threshold	0.139	0.917	42.109
Uni. correlation	0.677	19.564	62.887
Column Histogram	0.081	7.091	38.348
Sliding Windows	0.106	8.603	9.098
Total runtime	2.34	61.242	331.483

TABLE II
SPEED-UPS COMPARISON BETWEEN THE PROPOSED HETEROGENEUS (CUDA) AND OTHER IMPLEMENTED METHODS.

Operation	CUDA vs. CPU	CUDA vs. OpenCV
Conv. Grayscale	180.3	4.9
IPM	318.6	68.5
Temporal blurring	43.7	3.7
Adapt. threshold	302.9	6.6
Uni. correlation	92.9	28.9
Column Histogram	473.4	87.5
Sliding Windows	85.8	81.1
Total speed-up	141.6	26.1

In the CPU-based approach, processing a single frame took longer than 300 milliseconds, equivalent to approximately three frames per second (fps). The runtimes obtained demonstrate the inability to meet the criteria for a real-time application. The main reason for this long time is the image processing operations, which occur in a serialized fashion.

A typical camera (or video) has 30 fps, and the time between frames is a reasonable value to assume as the minimum for a real-time application.

The OpenCV approach had a speed-up of approximately 5.5 times compared to the CPU one, processing a single frame every 60 milliseconds or 16 fps. Much of the improvement is due to the speed-up of image processing operations, such as conversion to grayscale, adaptive threshold, and temporal integration. Despite a significant improvement, the implementation still does not meet the necessary criteria for real-time execution.

Finally, the proposed heterogeneous method can process a frame in less than three milliseconds. That is equivalent to processing over 300 fps and complying with a real-time requirement. It is evident that to maximize the performance on the GPU, the parallelization of the methods is necessary. It is worth noting that since the algorithm runs at least five times faster than the minimum required for a real-time application, it would still be possible to implement other features in the lane detection method. Additional implementations in the algorithm could further improve the detection quality without preventing the real-time requirement.

D. Quantitative analysis of ego-lane detection on Tusimple dataset

The benchmark compares the ego-lanes obtained by the algorithm with the ground truth. Thus, we demonstrate that the method using heterogeneous computing is capable of detecting

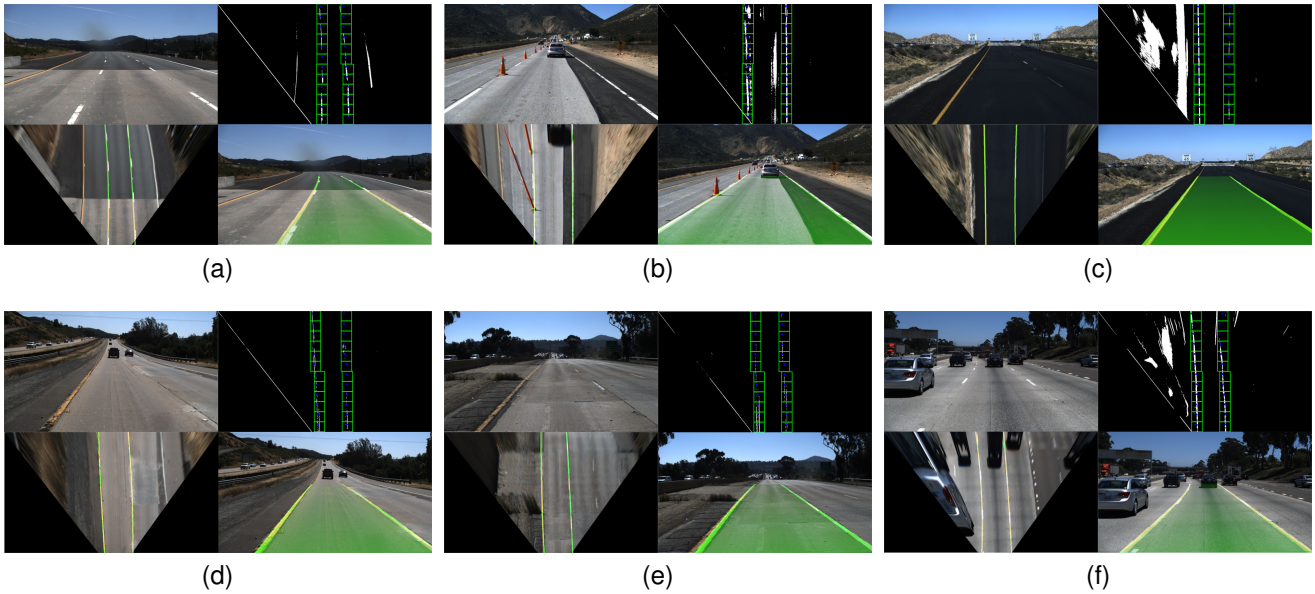


Fig. 9. Example of images in different scenarios for different stages of the proposed method.

the lane markings accordingly and is suitable for embedded applications.

It is possible to evaluate the quality of the lane markings estimation using the ground truth available in each clip of the dataset comparing with the results obtained by the proposed algorithm. The principal quality metric evaluated is the accuracy (ACC), described in (6). It measures the rate of valid estimates made by the method compared to the ground truth.

$$\text{ACC} = \frac{\sum_{clip} P_{valid}}{\sum_{clip} P_{total}}, \tag{6}$$

where P_{valid} corresponds to the number of correct points and P_{total} to the number of total points. An estimated point is valid if the difference between the ground truth point and the estimated one is smaller than a threshold value (T_{pixel}).

In addition to validating each point in the lane, it is necessary to validate the lane itself. A detected lane is said to be valid if it has at least $T_{point}\%$ of valid points, as shown in (7a). Similarly, (7b) measures the rate of markings estimated by the method that does not match the ground truth ones.

$$\text{Matched} = \frac{M_{pred}}{N_{pred}} \tag{7a}$$

$$\text{FP} = \frac{F_{pred}}{N_{pred}}, \tag{7b}$$

where M_{pred} is the number of matched predicted lanes, N_{pred} is the number of all predicted lanes, F_{pred} is the number of wrong predicted lanes. Matched represents the lane rate and FP the false positive rate.

The results of ACC, Matched and FP obtained for the TuSimple dataset are arranged in Table III. The data presents values for a T_{pixel} of 20 px, 35 px, and 50 px, and $T_{points} = 80\%$.

TABLE III
PERFORMANCE METRICS FOR DIFFERENT THRESHOLDS IN EACH OF THE THREE SUBSETS OF THE TUSIMPLE DATASET.

Dataset	T_{pixel} (px)	ACC (%)	$Matched$ (%)	FP(%)
0313	20	81.3	74.0	26.0
	35	88.4	84.4	15.6
	50	92.2	87.7	12.3
0531	20	90.0	87.7	12.3
	35	95.2	94.8	5.2
	50	97.3	95.9	4.0
0601	20	92.3	91.0	9.0
	35	97.2	97.3	2.7
	50	99.0	97.9	2.1

The ACC rate of the proposed method ranged from 81.3% to 99.0%, in the worst and best performance, respectively. The dataset 0313 detection rate is degraded because most images are taken from roads with non-reflective raised pavement markers called Botts' dots in place of painted lane marks. These Botts' dots are usually yellow or white. White-colored dots on concrete pavements will turn a pavement-like color due to dirt. The feature extraction algorithm is optimized for traditional traffic stripes and not such markings. On the other hand, dataset 0601 has long stretches of lanes with signaling in good condition, ensuring better results.

The proposed method performed well under the different conditions present in the datasets. The results were satisfactory even for the unconventional lanes present in dataset 0313. The overall average accuracy values for the different $T_{pixel} = 20$ px, 35 px, and 50 px are respectively 91.2%, 96.2%, and 98.1% when disregarding such markings. Comparison with other works and algorithms, both in terms of accuracy and computational complexity, is presented next.

TABLE IV
PERFORMANCE COMPARISON BETWEEN SELECTED LANE DETECTION ALGORITHMS.

Method	Algorithm	Average Detection (%)	Average Processing Time (ms) / Frame	Input Resolution	System Environment
Multi-lane	[17]	98.9	667.0	640x360	Intel Core i7-4th
Ego-lane	[19]	96.33	261.1	1280x720	Intel Core i7-2th
Multi-lane	[44]	95.7	65.4	768x432	Intel Core i7-6th
Multi-lane (deep leaning)	[45]	97.25	65.3	1280x720	NVIDIA GeForce GTX 1080, GPU
Ego-lane	[15]	95.75	63.6	1280x720	Intel Core i5-6th
Ego-lane	[46]	94.40	45.0	800x600	NVIDIA GeForce GTX 580, GPU
Multi-lane (deep learning)	[47]	97.25	42.0	1280x720	Intel Core Xeon E5-2th GeForce GTX TITAN-X, GPU
Multi-lane (straight)	[48]	98.10	34.0	640x480	NVIDIA Jetson TK1, board
Multi-lane	[21]	98.42	22.2	1280x720	Intel Core i5-6th
Ego-lane	Ours	98.2	2.34	1280x720	NVIDIA Jetson Nano, board

E. Comparison with other methods and discussions

Table IV places the proposed method in the lane detection scenario. It concisely presents the performance of other works, including similar methods for ego-lane detection and multi-lane detection. All values are from their respective publications. Thus, we can make a reasonably fair comparison between the methods. But we have to consider that they run on different systems, where some use different datasets and apply different methodologies to establish their metrics. However, the comparison shows strong evidence that the proposed system has a superior performance considering embedded real-time applications.

The deep learning methods ([45], [47]) perform multi-lane detection using the TuSimple dataset. They show high detection rates and approach real-time execution, even performing a more complex task than ego-lane detection. However, both woks report running their methods on desktops with dedicated graphics processing systems. Therefore, they have far superior processing capabilities than a GPU integrated into an embedded system, which would extrapolate the limited resources of the latter.

The works [15], [19] and [46] present solutions for ego-lane detection using conventional desktop computers. Their detection rates are slightly lower than the other methods that also use desktops. Among them, the fastest one utilizes lower resolution images and a dedicated GPU card.

The work in [48] presents an even simpler solution for multi-lane detection, which approximates the lanes to straight lines, not detecting the curvatures. It features a high detection rate but uses a low-resolution image, which directly impacts the runtime and quality of these detections. However, unlike the other proposed methods, it manages to achieve requirements close to real-time in an embedded system.

From the average processing time per frame in Table IV, only the proposed method can satisfy the real-time criteria established in this work. The optimization performed by the heterogeneous implementation using CUDA leads to an average processing time of 2.34 ms. That indicates the proposed method could perform using only part of the processing power and capabilities of a tailored for embedded systems GPU. Additionally, its average detection rate, excluding the TuSimple subset 0313, was approximately 98%. These results demonstrate the feasibility of implementing our method in embedded systems. Furthermore, adding improvements or new features to the algorithm is possible without compromising the real-time criteria.

VIII. CONCLUSIONS

This work presents a viable solution for ego-lane detection using embedded systems, e.g., NVIDIA Jetson Nano. Our proposal uses a heterogeneous implementation, allowing the optimization of all parts of the detection algorithm.

The algorithm preprocesses the input image to obtain a BEV perspective with the enhanced road markings. From this image, two feature maps are extracted and combined to produce an image that contains only the lane markings, which are then detected.

The method performs within the established real-time limits. The embedded system executed it in just 2.34 ms, which is 25 faster than the OpenCV GPU compiled version and 140 times faster than the CPU-executed sequential version. Additionally, considering the method's run-time, future implementations can improve even further and enhance the detection quality still copping with the real-time criteria.

The method proved efficient for detecting ego-lanes on roads and highways running TuSimple dataset images with different scenarios. The method's performance ranged from 92.3% to 99.0% accuracy in the best subset, detecting up to 97.9% of the ego-lanes available with 80% of the valid points. In this way, the work is in the same range as other similar methods with the difference of being executed in an inexpensive graphics-processing-tailored real-time embedded system.

REFERENCES

[1] S. O. D. Transp., "Reported road casualties great britain 2009," London, U.K., Tech. Rep., 2010.

[2] NHTSA, "Critical reasons for crashes investigated in the national motorvehicle crash causation survey," Washigton, DC, USA, Tech. Rep., 02 2015.

[3] K. Bengler, K. Dietmayer, B. Farber, M. Maurer, C. Stiller, and H. Winner, "Three decades of driver assistance systems: Review and future perspectives," *Intelligent Transportation Systems Magazine, IEEE*, vol. 6, pp. 6–22, 12 2014.

[4] A. Bar Hillel, R. Lerner, D. Levi, and G. Raz, "Recent progress in road and lane detection: A survey," *Machine Vision and Applications*, vol. 25, 04 2014.

[5] C.-H. Kum, D.-C. Cho, M.-S. Ra, and W.-Y. Kim, "Lane detection system with around view monitoring for intelligent vehicle," *Proc. International SoC Design Conference*, pp. 215–218, 11 2013.

[6] G. Silva, D. Batista, M. Tosin, and L. Melo, "Estratégia de detecção de faixas de trânsito baseada em câmera monocular para sistemas embarcados," *Anais do XXIII Congresso Brasileiro de Automática*, vol. Volume 2 No 1, 12 2020.

[7] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[8] Z. Zhang and X. Ma, "Lane Recognition Algorithm Using the Hough Transform Based on Complicated Conditions," *Journal of Computer and Communications*, vol. 07, no. 11, pp. 65–75, 2019.

[9] Y. Huang, Y. Li, X. Hu, and W. Ci, "Lane detection based on inverse perspective transformation and Kalman filter," *KSII Transactions on Internet and Information Systems*, vol. 12, no. 2, pp. 643–661, 2018.

[10] W. Li, F. Qu, Y. Wang, L. Wang, and Y. Chen, "A robust lane detection method based on hyperbolic model," *Soft Computing*, vol. 23, no. 19, pp. 9161–9174, Oct 2019.

[11] C. Lee and J. Moon, "Robust lane detection and tracking for real-time applications," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 12, pp. 4043–4048, 2018.

[12] Y. Wang, E. Teoh, and D. Shen, "Lane detection and tracking using b-snake," *Image and Vision Computing*, vol. 22, pp. 269–280, 04 2004.

[13] X. Li, X. Fang, W. ci, and W. Zhang, "Lane detection and tracking using a parallel-snake approach," *Journal of Intelligent and Robotic Systems*, vol. 77, 03 2014.

[14] A. Borkar, M. Hayes, and M. T. Smith, "A novel lane detection system with efficient ground truth generation," *IEEE Transactions on Intelligent Transportation Systems*, vol. 13, no. 1, pp. 365–374, 2012.

[15] R. Muthalagu, A. Bolimera, and K. Venkatesan, "Lane detection technique based on perspective transformation and histogram analysis for self-driving cars," *Computers & Electrical Engineering*, vol. 85, p. 106653, 07 2020.

[16] A. Borkar, M. Hayes, and M. T. Smith, "Robust lane detection and tracking with ransac and kalman filter," in *2009 16th IEEE International Conference on Image Processing (ICIP)*, 2009, pp. 3261–3264.

[17] Y. Son, E. Lee, and D. Kum, "Robust multi-lane detection and tracking using adaptive threshold and lane classification," *Machine Vision and Applications*, vol. 30, 10 2019.

[18] J. Son, H. Yoo, S. Kim, and K. Sohn, "Real-time illumination invariant lane detection for lane departure warning system," *Expert Systems with Applications*, vol. 42, 10 2014.

[19] C.-B. Wu, L.-H. Wang, and K.-C. Wang, "Ultra-low complexity block-based lane detection and departure warning system," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 29, no. 2, pp. 582–593, 2019.

[20] M. Reichenbach, L. Liebischer, S. Vaas, and D. Fey, "Comparison of lane detection algorithms for adas using embedded hardware architectures," in *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2018, pp. 48–53.

[21] J. Cao, S. Song, W. Xiao, and Z. Peng, "Lane detection algorithm for intelligent vehicles in complex road conditions and dynamic environments," *Sensors*, vol. 19, p. 3166, 07 2019.

[22] X. Zhi, J. Yan, Y. Hang, and S. Wang, "Realization of cuda-based real-time registration and target localization for high-resolution video images," *Journal of Real-Time Image Processing*, vol. 16, no. 4, pp. 1025–1036, Aug 2019.

[23] M. Afif, Y. Said, and M. Atri, "Computer vision algorithms acceleration using graphic processors nvidia cuda," *Cluster Computing*, vol. 23, no. 4, pp. 3335–3347, Dec 2020.

[24] J. Li, G. Deng, W. Zhang, C. Zhang, F. Wang, and Y. Liu, "Realization of CUDA-based real-time multi-camera visual SLAM in embedded systems," *Journal of Real-Time Image Processing*, vol. 17, no. 3, pp. 713–727, 2020.

[25] B. He, R. Ai, Y. Yan, and X. Lang, "Accurate and robust lane detection based on dual-view convolutional neutral network," in *2016 IEEE Intelligent Vehicles Symposium (IV)*, 2016, pp. 1041–1046.

[26] D. Cácere Hernández, A. Filonenko, A. Shahbaz, and K.-H. Jo, "Lane marking detection using image features and line fitting model," in *2017 10th International Conference on Human System Interactions (HSI)*, 2017, pp. 234–238.

[27] J. Kim, J. Kim, G.-J. Jang, and M. Lee, "Fast learning method for convolutional neural networks using extreme learning machine and its application to lane detection," *Neural Networks*, vol. 87, pp. 109–121, 2017.

[28] W. Van Gansbeke, B. De Brabandere, D. Neven, M. Proesmans, and L. Van Gool, "End-to-end lane detection through differentiable least-squares fitting," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, 2019, pp. 905–913.

[29] Z. Yonglong, M. Kuizhi, J. Xiang, and D. Peixiang, "Parallelization and optimization of sift on gpu using cuda," in *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, 2013, pp. 1351–1358.

[30] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[31] N. Corporation, "Geforce gtx 980 - Featuring Maxwell, The Most Advanced GPU Ever Made," NVIDIA, Tech. Rep., 2014.

[32] NVIDIA Corporation, *NVIDIA CUDA Programming Guide*. NVIDIA Corporation, 2011.

[33] S. Yenikaya, G. Yenikaya, and E. Düven, "Keeping the vehicle on the road: A survey on on-road lane detection systems," *ACM Comput. Surv.*, vol. 46, no. 1, Jul. 2013.

[34] A. Bar Hillel, R. Lerner, D. Levi, and G. Raz, "Recent progress in road and lane detection: A survey," *Machine Vision and Applications*, vol. 25, 04 2014.

[35] S. P. Narote, P. N. Bhujbal, A. S. Narote, and D. M. Dhane, "A review of recent advances in lane detection and departure warning system," *Pattern Recognition*, vol. 73, pp. 216–234, 2018.

[36] Y. Yu and K.-H. Jo, "Lane detection based on color probability model and fuzzy clustering," in *Ninth International Conference on Graphic and Image Processing (ICGIP 2017)*, ser. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, H. Yu and J. Dong, Eds., vol. 10615, Apr. 2018, p. 1061508.

[37] S. Sivaraman and M. M. Trivedi, "Looking at vehicles on the road: A survey of vision-based vehicle detection, tracking, and behavior analysis," *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 4, pp. 1773–1795, 2013.

[38] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. New York, NY, USA: Cambridge University Press, 2003.

[39] M. B. de Paula and C. R. Jung, "Automatic detection and classification of road lane markings using onboard vehicular cameras," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 6, pp. 3160–3169, 2015.

[40] A. Bodis-Szomoru, T. Daboczi, and Z. Fazekas, "A far-range off-line camera calibration method for stereo lane detection systems," 2007.

[41] H. Zeng, N. Peng, Z. Yu, Z. Gu, H. Liu, and K. Zhang, "Visual tracking using multi-channel correlation filters," in *2015 IEEE International Conference on Digital Signal Processing (DSP)*, 2015, pp. 211–214.

[42] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach: Third Edition*. Elsevier Inc., Dec. 2016.

[43] G. B. Silva. Cuda-based real-time ego-lane detection in embedded system-tusimple#0601. [Online]. Available: https://youtu.be/1q_Jy8MJjFw

[44] F. Zheng, S. Luo, K. Song, C.-W. Yan, and M.-C. Wang, "Improved lane line detection algorithm based on hough transform," *Pattern Recognition and Image Analysis*, vol. 28, no. 2, pp. 254–260, Apr 2018. [Online]. Available: https://doi.org/10.1134/S1054661818020049

[45] J. Philion, "Fastdraw: Addressing the long tail of lane detection by adapting a sequential prediction network," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 11 574–11 583.

[46] T. Kühnl, F. Kummert, and J. Fritsch, "Spatial ray features for real-time ego-lane extraction," in *2012 15th International IEEE Conference on Intelligent Transportation Systems*, 2012, pp. 288–293.

[47] Q. Zou, H. Jiang, Q. Dai, Y. Yue, L. Chen, and Q. Wang, "Robust lane detection from continuous driving scenes using deep neural networks," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 1, pp. 41–54, 2020.

[48] H.-S. Kim, S.-H. Beak, and S.-Y. Park, "Parallel hough space image generation method for real-time lane detection," in *Advanced Concepts for Intelligent Vision Systems*, J. Blanc-Talon, C. Distante, W. Philips,

D. Popescu, and P. Scheunders, Eds. Cham: Springer International Publishing, 2016, pp. 81–91.

IX. Biography Section

Guilherme Brandão da Silva hold's a bachelor's degree in Electrical Engineer in with the State University of Londrina (2019) and is currently pursuing his master's degree with the same institution. He has been a researcher with the Inertial Measurements and Aerospace Instrumentation Laboratory since 2017. His areas of interest are: real-time embedded systems, computer vision, and intelligent vehicles.

Daniel Strufaldi Batista hold's a bachelor's (2014) and Master's degree (2016), and is currently pursuing his Ph.D. degree, all in electrical engineering with the State University of Londrina, Londrina, Brazil. He has been a researcher and collaborator with the Inertial Measurements and Aerospace Instrumentation Laboratory with the State University of Londrina since 2012. His areas of interest are: electronics systems, embedded systems, instrumentation, sensor calibration and attitude determination systems.

Marcelo Carvalho Tosin holds a bachelor's degree in physics from the University of São Paulo (1994), and the Master's and Ph.D. degrees in electrical engineering from the University of Campinas, Campinas, Brazil, in 1998 and 2001, respectively. Since 2000 he is the Head with the Inertial Measurements and Aerospace Instrumentation Laboratory and a Professor, at the State University of Londrina. His areas of interest are embedded systems, inertial sensors, sensor calibration and attitude determination.

Décio Luiz Gazzoni Filho received the bachelor's degree in electrical engineering from the State University of Londrina (2005), Brazil, and the Master's degree in electrical engineering from the University of São Paulo (2008), Brazil. Since 2020 he is a Ph.D. candidate at the Institute of Computing of the University of Campinas. Since 2012, he has been a researcher with the Inertial Measurements and Aerospace Instrumentation Laboratory and a Professor, at State University of Londrina, Brazil. His areas of interest are: high-performance implementation of cryptographic algorithms, electronics systems, real-time embedded systems.

Leonimer Flávio de Melo received the bachelor's degree and the master's degree both in electrical engineering in 1985 and 2002, respectively, and the Ph.D. degree in mechanical engineering, in 2007, all from the University of Campinas, Campinas Brazil. He is currently a Professor at the Ph.D. program with the State University of Londrina, Londrina, Brazil, and also a Collaborator with the Inertial Measurements and Aerospace Instrumentation Laboratory with the same institution.