CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS TIMÓTEO

João Pedro Medeiros Brier

COMPARAÇÃO DE DESEMPENHO ENTRE CPU E GPU NA DETECÇÃO E RASTREAMENTO DE OBJETOS

Timóteo

2021

João Pedro Medeiros Brier

COMPARAÇÃO DE DESEMPENHO ENTRE CPU E GPU NA DETECÇÃO E RASTREAMENTO DE OBJETOS

Monografia apresentada à Coordenação de Engenharia de Computação do Campus Timóteo do Centro Federal de Educação Tecnológica de Minas Gerais para obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Odilon Corrêa da Silva

Timóteo

2021

João Pedro Medeiros Brier

COMPARAÇÃO DE DESEMPENHO ENTRE CPU E GPU NA DETECÇÃO E RASTREAMENTO DE OBJETOS

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais, campus Timóteo, como requisito parcial para obtenção do título de Engenheiro de Computação.

Trabalho aprovado. Timóteo, 20 de abril de 2021:

_____

Prof. Me. Odilon Corrêa da Silva
Orientador

_____

Prof. Dr. Lucas Pantuza Amorim
Professor Convidado

_____

Prof. Dr. Elder de Oliveira Rodrigues
Professor Convidado

Timóteo
2021

Emitido em 20/04/2021

CÓPIA DE FOLHA DE ASSINATURAS Nº 2/2021 - DCCTM (11.63.05)

(Nº do Protocolo: NÃO PROTOCOLADO)

(Assinado digitalmente em 04/06/2021 07:59)
ELDER DE OLIVEIRA RODRIGUES
PROFESSOR ENS BASICO TECN TECNOLOGICO
DCCTM (11.63.05)
Matrícula: 1694225

(Assinado digitalmente em 03/06/2021 21:05)
LUCAS PANTUZA AMORIM
PROFESSOR ENS BASICO TECN TECNOLOGICO
DCCTM (11.63.05)
Matrícula: 2897411

(Assinado digitalmente em 03/06/2021 15:18)
ODILON CORREA DA SILVA
PROFESSOR ENS BASICO TECN TECNOLOGICO
DCCTM (11.63.05)
Matrícula: 2794495

(Assinado digitalmente em 03/06/2021 19:30)
João Pedro Medeiros Brier
DISCENTE
Matrícula: 201617060127

Para verificar a autenticidade deste documento entre em https://sig.cefetmg.br/documentos/ informando seu número: **2**, ano: **2021**, tipo: **CÓPIA DE FOLHA DE ASSINATURAS**, data de emissão: **03/06/2021** e o código de verificação: **1f9d4f1e57**

Dedico a minha família, que sempre esteve presente por mim quando precisei.

Agradecimentos

Agradeço ao meu orientador Odilon Corrêa da Silva por aceitar conduzir o meu trabalho de pesquisa.

A todos os meus professores do curso de Engenharia da Computação do CEFET/MG do Campus Timóteo pela excelência da qualidade técnica de cada um.

Aos meus pais e familiares que sempre estiveram ao meu lado me apoiando ao longo de toda a minha trajetória.

Também agradeço a todos os meus amigos do curso de graduação que compartilharam dos inúmeros desafios que enfrentamos, sempre com o espírito colaborativo.

“Os grandes navegadores
devem sua reputação aos temporais e tempestades.”
Epicuro

Resumo

Apesar de sua grande aplicabilidade e usabilidade, utilizar um computador para reconhecer objetos em imagens digitais de forma eficiente continua sendo um desafio da computação. Este trabalho propõe a implementação e comparação em CPU e GPU de um método de reconhecimento e rastreamento de objetos em imagens ou sequências de imagens, capturadas de um vídeo, utilizando a métrica de tempo de processamento. Para tanto, foram aplicadas técnicas de processamento digital de imagens e o algoritmo YOLO. Para validar o funcionamento do método proposto, desenvolveu-se um algoritmo em Python utilizando recursos da biblioteca OpenCV e da base da dados COCO. Nos testes realizados, o algoritmo em GPU apresentou um speedup superior a $5,5\times$ em relação à implementação feita em CPU. Isso mostra a viabilidade do uso de GPUs na detecção e rastreamento de objetos em imagens digitais.

Palavras-chave: GPU, CPU, detecção de objetos, rastreamento de objetos.

Abstract

Despite its wide applicability and usability, using a computer to efficiently recognize objects in digital images remains a computing challenge. This work proposes the implementation and comparison in CPU and GPU of a method of recognition and tracking of objects in images or image sequences, captured from a video, using the processing time metric. Therefore, digital image processing techniques and the YOLO algorithm were applied. To validate the functioning of the proposed method, an algorithm was developed in Python using resources from the OpenCV library and from the COCO database. In the tests performed, the GPU algorithm presented a speedup higher than $5.5\times$ compared to the CPU implementation. This shows the feasibility of using GPUs in detecting and tracking objects in digital images.

Keywords: GPU, CPU, object detection, object tracking.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

BLOB	*Binary Large Object*
COCO	*Microsoft Common Objects in Context*
CPU	*Central Process Unit*
CUDA	*Compute Unified Device Architecture*
FPGA	*Field-Programmable Gate Array*
FPS	*Frames per Second*
GPU	*Graphics Processing Unit*
PDI	Processamento Digital de Imagens
TR	Tempo de Resposta
TRM	Tempo de Resposta Médio
YOLO	*You Only Look Once*

Sumário

1 Introdução

“A resposta certa, não importa nada:
o essencial é que as perguntas estejam certas.”
Mário Quintana

A visão computacional pode ser descrita como um conjunto de teorias e técnicas para obtenção de dados por meio de imagens, sendo de modo geral, o estudo da capacidade das máquinas de visualizarem o mundo real, por meio de sensores, câmeras e outros dispositivos que extraem informação dos ambientes (ALVES; GATTASS; CARVALHO, 2005). Neste contexto, a detecção de um objeto em uma imagem consiste em utilizar um meio de visão, que envolve a análise, categorização e reconhecimento dos componentes que constituem tal imagem (GRACIANO, 2007). Mas o problema de rastrear um objeto pode se mostrar bastante complexo e computacionalmente custoso, uma vez que as variáveis envolvidas no sistema, como a intensidade luminosa da cena, o ruído presente na captação da imagem, o comportamento e a própria forma do objeto a ser rastreado não podem ser controlados ou conhecidos previamente (OLIVEIRA, 2008).

Técnicas de reconhecimento de padrões tem como principal objetivo a classificação de objetos dentro de um número de categorias ou classes (DUDA; HART; D.G, 2000), como por exemplo o Viola-Jones, também conhecido como Haar-Cascade, que é um algoritmo utilizado em problemas de detecção de objetos (VIOLA; JONES, 2001). No entanto, tanto a fase de treinamento do algoritmo, quanto a detecção de objetos são tarefas computacionalmente caras, em que as soluções de *software* de detecção em CPU (*Central Process Unit*) podem fornecer baixas taxas de quadros por segundo (HARVEY, 2009a). O problema na parte do treinamento pode piorar em ferramentas de classificação que possuem grandes bases de dados, ou mesmo naquelas onde o conjunto de dados não é tão grande, mas o treinamento dos mesmos precisa ser realizado constantemente, o que pode tornar o tempo necessário para executá-lo inviável. Já a velocidade de detecção do objeto influenciará o desempenho de processamento posterior conforme o tempo de detecção aumenta linearmente com o tamanho das imagens (Wai; Tahir; Chang, 2015).

Companhias que produzem GPU (*Graphic Processor Unit*), tais como a NVIDIA e AMD, empenham-se em produzir dispositivos cada vez mais poderosos, as quais são projetadas para executar operações em paralelo, possibilitando, um aumento de desempenho quando comparado às CPUs. Existem duas principais arquiteturas para desenvolvimento de aplicações em processadores gráficos: CUDA (*Compute Unified Device Architecture*) e CTM (*Close To Metal*). Enquanto CUDA é uma tecnologia de programação baseada na linguagem C++ utilizada para explorar os recursos da GPU NVIDIA (NVIDIA, 2015), orientada para placas gráficas da NVIDIA, CTM segue os padrões da empresa ATI.

Trabalhos que propõem a implementação de técnicas de reconhecimento de padrões nestas GPUs podem ser encontrados na literatura, tais como:

1. Implementação das SVMs (*Support Vector Machines*) em GPU (CATANZARO; SUNDARAM; KEUTZER, 2008);
2. Detector facial Viola-Jones usando a estrutura GPUCV (Wai; Tahir; Chang, 2015);
3. Implementação de redes neurais combinando CUDA e OpenMP (*Open Multi-Processing*) (JANG; PARK; JUNG, 2008).

Cada um destes exemplos apresenta restrições e características específicas com as quais diferentes técnicas de reconhecimento de padrão devem lidar. Por exemplo, os dois primeiro trabalhos apresentam resultados de ganhos de velocidade em uma GPU quando comparados à execução em uma CPU tradicional. E o terceiro aborda o problema do custo de treinamento de sua base de dados.

Com o desenvolvimento e evolução das técnicas de reconhecimento de padrão, pesquisas foram desenvolvidas com o objetivo de criar bibliotecas para otimizar e facilitar as análises de imagens em GPU. Revisando a literatura, algumas ferramentas foram encontradas:

- OpenCV — Visão computacional open source (OPENCV, 2020);
- OpenCL — Arquitetura para escrever programas que funcionam em plataformas heterogêneas (NVIDIA, 2021a);
- TensorFlow — Plataforma de reconhecimento de padrão *open source* (GOOGLE, 2021);
- OpenACC — Padrão de programação para computação paralela (NVIDIA, 2021b).

Essas ferramentas fornecem métodos que podem ser empregados no desenvolvimento de soluções eficientes em diferentes abordagens. No contexto de reconhecimento e rastreamento de objetos em imagens em GPU, estão presentes o uso de Redes Neurais Artificiais (RNA), técnicas de descritores estatísticos, métodos exatos e outros (OH; JUNG, 2004). A seleção da técnica ideal deve ser formulada de acordo com o problema a ser trabalhado.

1.1 Justificativa e Problema

A visão computacional vem sendo utilizada para extrair informações, classificar e identificar objetos, pessoas ou processos específicos. Normalmente, esses sistemas envolvem a aplicação de técnicas de inteligência artificial, com foco em tomada de decisão e aplicação de probabilidade. O processo de reconhecimento de objetos, entretanto, não é trivial. O reconhecimento genérico com invariância à perspectiva, iluminação e a presença de ruídos é um dos desafios da área de visão computacional (LeCun; Fu Jie Huang; Bottou, 2004). Devido a complexidade da informação que deve ser interpretada e a necessidade de processamento rápido, e ainda a necessidade de resultados precisos, são requeridos algoritmos eficientes

no tratamento e processamento de imagens, o que torna relevante o estudo de técnicas que possam suprir essa demanda.

Nesse contexto se inserem as GPUs, que podem ser utilizadas não apenas para renderizar gráficos 2D e 3D, mas também para outras tarefas de computação científica. As GPUs surgiram como uma fonte maior de poder de computação que podem ser usadas para cálculos de uso geral em comparação com uma CPU clássica, como sugerem os trabalhos previamente citados, em que Catanzaro et al. (CATANZARO; SUNDARAM; KEUTZER, 2008) obteve um ganho de 9 até 35 vezes de velocidade quando comparada a execução em uma CPU tradicional. Enquanto Wai et al. (Wai; Tahir; Chang, 2015) concluíram que resultados experimentais mostram um speedup da GPU proposta de até 22 vezes em comparação com uma CPU.

Visto que todos os trabalhos citados acima obtiveram significativos ganhos de desempenho ao implementarem soluções em GPU, espera-se neste trabalho também alcançar aumentos de desempenho em relação à soluções de CPU.

1.2 Objetivos

Este trabalho tem como principal objetivo especificar e implementar um método para detecção e rastreamento de objetos tanto em CPU quanto em GPU.

Também, objetiva-se mais especificamente:

1. Identificar métodos de detecção de objetos que possam ser utilizados na detecção e classificação de objetos em imagens digitais;
2. Implementar um dos métodos para uma execução paralela em GPU;
3. Avaliar o desempenho do método utilizando a métrica de tempo de processamento.

2 Fundamentação Teórica

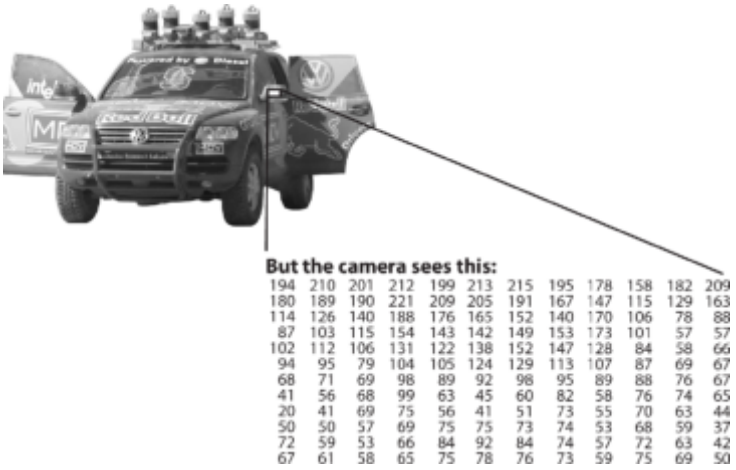
“A tarefa não é tanto ver aquilo que ninguém viu, mas pensar o que ninguém ainda pensou sobre aquilo que todo mundo vê.”
Arthur Schopenhauer

Nas próximas seções são introduzidos alguns conceitos fundamentais para o desenvolvimento e entendimento deste trabalho, especificamente nas áreas de visão computacional, processamento de imagens digitais e computação paralela em GPU.

2.1 Visão Computacional

A expressão “Visão Computacional” é entendida como o processo de aquisição de imagens realizado por sensores e interpretados por sistemas computacionais. Gonzalez e Woods (GONZALEZ; WOODS, 2008) descrevem como um conjunto de técnicas que tem como objetivo auxiliar o computador a interpretar o conteúdo da imagem. Já Szeliski (SZELISKI, 2011), complementa dizendo que a visão computacional é a transformação do dado da imagem em uma nova representação, ou seja, busca-se descrever o mundo a partir de imagens e reconstruí-lo em suas propriedades, como forma, iluminação e distribuição de cores. A Figura 1 mostra um exemplo de como uma máquina interpreta uma imagem.

Figura 1 – Matriz de dados de uma imagem.



Fonte: (BRADSKI, 2008)

No âmbito da visão computacional, o objetivo é emular a visão humana fazendo com que máquinas possam ter a capacidade de aprender e tomar decisões a partir da captura digital de imagens. Humanos estão limitados a uma faixa visível do espectro eletromagnético, porém máquinas conseguem interpretar imagens geradas a partir de um maior conjunto de informações do espectro eletromagnético, variando de ondas gama a ondas de rádio. As fontes

dessas informações são diversas, como sensores de ultrassom ou infravermelho (GONZALEZ; WOODS, 2008).

Analogamente ao sistema de visão humana, um sistema eletrônico possui um sensor (câmera) que recebe uma informação luminosa, envia essa informação ao microcontrolador que a processa, interpreta e toma a decisão sobre qual ação deve ser realizada. Essa ação é baseada em informações contidas em uma memória, que podem ser comparadas para definir a melhor opção.

As vertentes, desde a visão computacional até o processamento da imagem, se complementam, por tanto é difícil definir um limite entre os mesmos, o que, muitas vezes, confunde onde um conceito termina e a outra inicia. Portanto, para uma melhor compreensão sobre o processamento de imagem, é necessário definir alguns conceitos.

2.2 Processamento Digital de Imagens

Através de fotos ou vídeos capturados por câmeras digitais, é obtida a informação necessária para se realizar o processamento de imagens. Uma imagem digital é definida como uma função $F(x, y)$ como, por exemplo na Figura 2, onde x e y são as coordenadas espaciais e o valor de f corresponde à intensidade de brilho (ou níveis de cinza) da imagem nesta coordenada. Gonzalez e Woods (GONZALEZ; WOODS, 2008) descreve uma imagem digital como sendo uma matriz cujos índices de linhas e de colunas identificam um ponto na imagem, e o correspondente valor do elemento da matriz identifica o nível de cinza naquele ponto.

Estes pontos que são identificados na imagem são os *pixels*, a menor unidade que representa uma imagem, com seu valor e informação subsequente.

2.2.1 Etapas do Processamento Digital de Imagens

A aquisição da imagem é o passo inicial para o processamento de imagens, o processo começa reduzindo a dimensionalidade da cena; a câmera digital transforma a cena tridimensional em bidimensional. A aquisição de imagens digitais requer basicamente dois elementos (FILHO; NETO, 1999):

1. Um dispositivo físico como um sensor de imagens com dispositivo de carga acoplada (CCD), normalmente encontrada em câmeras digitais compactas ou com um semicondutor metal-oxido complementar (CMOS) encontrada em câmeras digitais semi-profissionais e profissionais;
2. Um conversor de saída elétrica do dispositivo de sensoreamento físico para o formato digital, capaz de digitalizar o sinal produzido pelo sensor.

Deve-se levar em conta a escolha do tipo do sensor CCD ou CMOS, as lentes a serem utilizadas, condições de iluminação da cena, velocidade da aquisição (tempo de resposta de aquisição de imagem da câmera), resolução da imagem, dentre outros (FILHO; NETO, 1999).

Figura 2 – Imagem no plano cartesiano (x, y).

Fonte: (FILHO; NETO, 1999)

Este passo é fundamental para obter uma imagem de boa qualidade e resolução. A Figura 3 representa a captação de imagem, detalhando os *pixels*.

Figura 3 – Captura de Imagem

Fonte: (PORTES, 2000)

Concluída a fase de aquisição de imagem, parte-se então para o pré-processamento,

onde é realizada a melhoria da imagem para aumentar as chances de melhorar o resultado nos processos seguintes. Normalmente nesta fase são feitos ajustes de contraste, redução de ruído entre outras melhorias que podem ser feitas na imagem (GONZALEZ; WOODS, 2008).

O próximo passo é a segmentação, onde a imagem digital original é divida em partes ou objetos constituintes. Marques Filho e Vieira Neto (FILHO; NETO, 1999) definem a tarefa básica da etapa de segmentação como a de dividir uma imagem em suas unidades significativas, ou seja, nos objetos de interesse que a compõem, selecionando as partes que interessam na imagem. Normalmente essa etapa é aplicada na separação da imagem, que contém fundo e objeto, buscando separar por nível de contraste. A Figura 4 representa um exemplo do processo de segmentação.

A extração de atributos comumente é resultado da segmentação, pois torna-se possível a extração de informações, dados ou atributos relevantes para o analisador. Nesta fase é realizada a transformação dos dados obtidos pela fase de segmentação para que o computador possa processar de forma correta. Os dados podem ser representados de duas formas: A primeira é a representação por fronteiras, utilizada quando o interesse se concentra nas características da forma externa, como: pontos ou cantos. A segunda é a representação por região, onde o interesse se concentra na área interna das formas (GONZALEZ; WOODS, 2008).

O processo de atribuir significado ao objeto com base em descritores corresponde à fase de reconhecimento, quando os parâmetros são agrupados a partir de suas semelhanças. Pode-se dizer que a análise de imagens é nada mais que um processo de descobrimento, de identificação e de entendimento dos padrões que são importantes para o desempenho de uma tarefa baseada em imagens. Dotar um computador com a capacidade de se aproximar, em um determinado sentido, da capacidade de visão dos seres humanos é uma das principais metas da análise de imagens (GONZALEZ; WOODS, 2008).

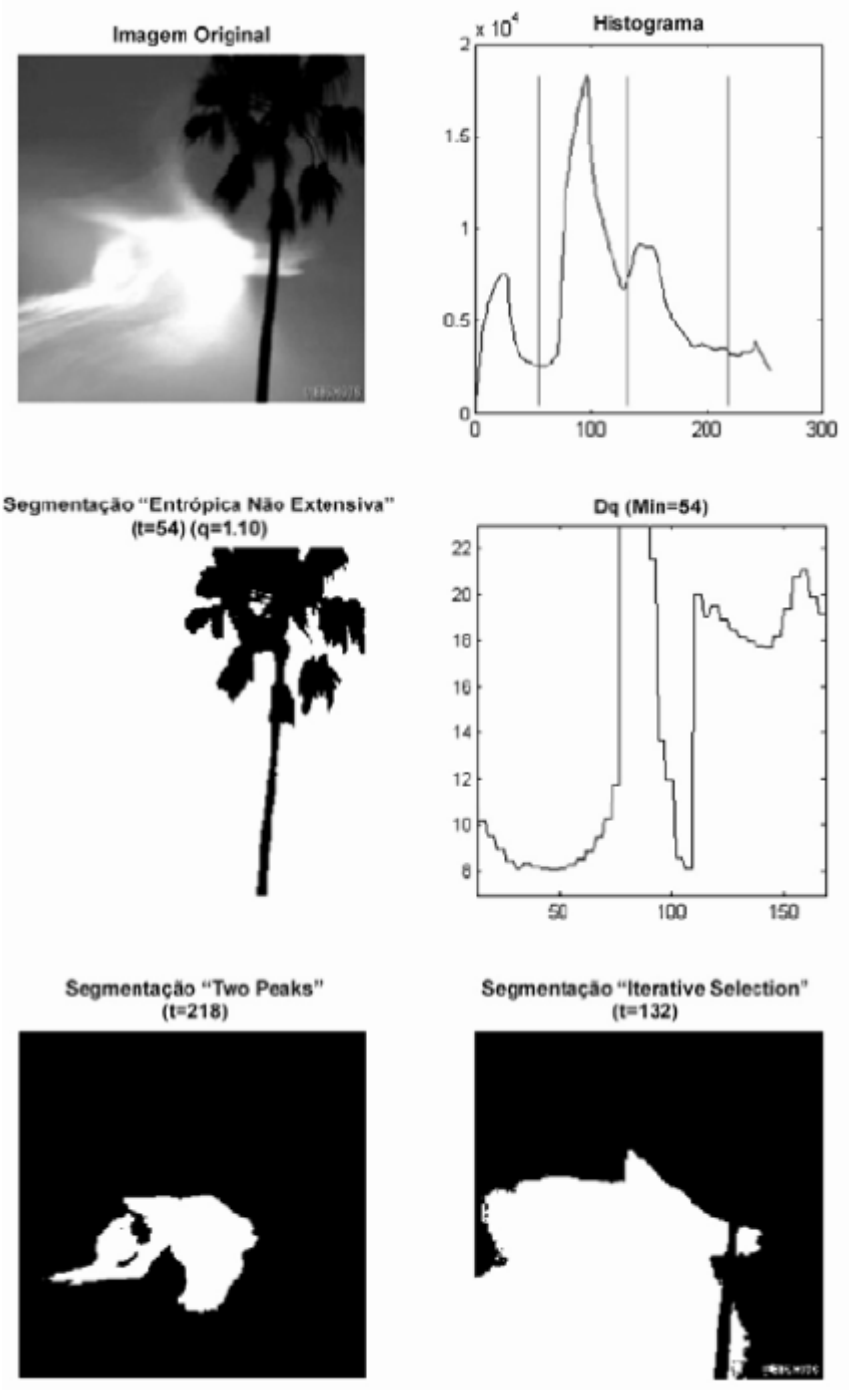
A etapa de decisão retrata o objetivo da visão computacional, pois é responsável por atribuir finalidade ao conjunto de dados obtidos até então. A tomada de decisão pode ser realizada por meio de comparações de resultados ou com uso de técnicas aprimoradas e que envolvem a inteligencia artificial, tornando o sistema mais complexo. Desse modo, a informação não está disponível diretamente e precisa ser obtida.

2.2.2 Reconhecimento de Padrões

Dentro do reconhecimento e interpretação de imagens existem categorias, tais como o reconhecimento de padrões. O reconhecimento de padrões tem como objetivo realizar a classificação de objetos conforme sua categoria ou classe a partir da análise de suas características. A classificação automática pode ser dividida em dois grupos: supervisionada e não supervisionada (SATHYA, 2013). Existe ainda um terceiro grupo chamado de classificação híbrida, onde a classificação supervisionada e não supervisionada atuam de forma conjunta, aumentando a probabilidade de melhorar o resultado das classificações.

A classificação supervisionada utiliza uma base de dados com padrões previamente cadastrados para definir a categoria de cada objeto. Já na classificação não supervisionada,

Figura 4 – Segmentação da Imagem.



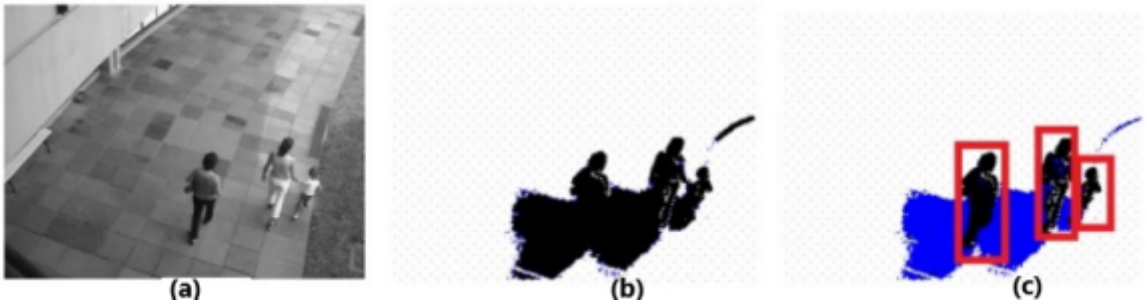
Fonte: (ALBUQUERQUE; ESQUEF, 2008)

não existem informações armazenadas sobre as categorias dos padrões reconhecidos na imagem. Através de métodos não supervisionados, são realizados agrupamentos conforme a disposição dos padrões na tela.

2.2.3 Tipos de Processamento Digital

A imagem digital consiste numa representação de um objeto, de forma que possa ser interpretada por um computador. Para o processamento, a imagem é convertida do mundo real para o digital, podendo então ter uso computacional. São três procedimentos básicos: os processos de baixo nível, médio nível e alto nível, que estão representados na Figura 5.

Figura 5 – Níveis de Processamento Digital.



Fonte: (JUNIOR, 2015)

No processo de baixo nível (Figura 5a) é realizada a restauração quantitativa das imagens com o intuito de corrigir as degradações geométricas inseridas pelo sensor ainda no processo de formação das imagens, além de remover o ruído da imagem. No nível médio (Figura 5b) são incluídas técnicas que acentuam características relevantes da imagem, transformando a imagem de forma que as informações possam ser extraídas com melhor resultado. Para isto, usam-se a manipulação do contraste, isolamento de regiões da imagem entre outras. Já no de alto nível (Figura 5c) ocorre a extração de informação de informações da imagem, a partir de várias técnicas como a segmentação da imagem, repartindo a imagem em regiões com características diferentes, classificando a imagem, reconhecendo objetos e padrões. Os resultados da aplicação destas técnicas resultam em descrições dos objetos da cena.

2.3 Algoritmos de Detecção de Objetos

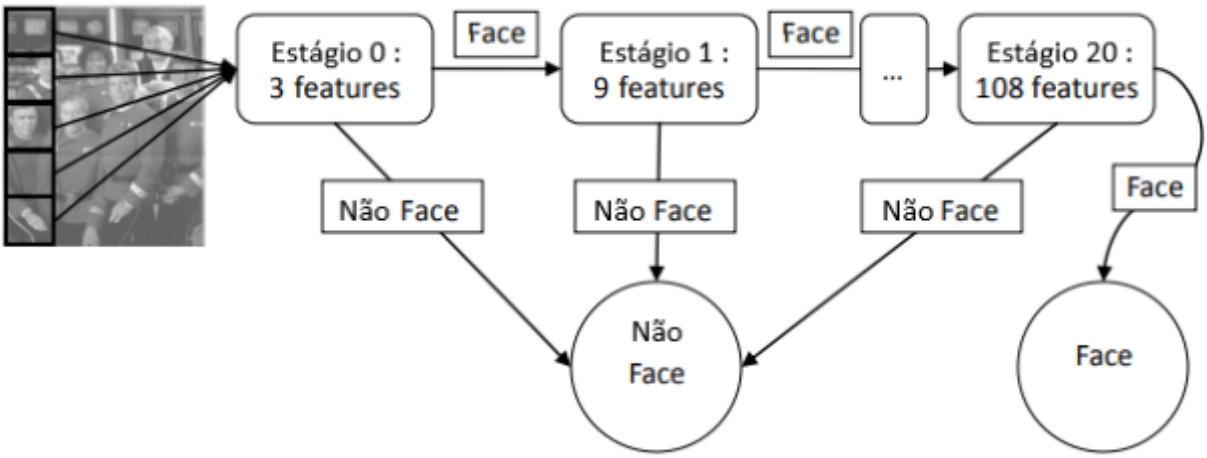
2.3.1 Algoritmo Viola-Jones

O algoritmo de Viola-Jones, ou *Haar Cascade*, é empregado para detecção de faces e pode ser dividido em duas partes. A primeira parte é o treinamento de um conjunto de classificadores fracos com base nas características (*features*) e a formação de um classificador em cascata de estágios com todos os classificadores fracos promissores usando Adaboost (VIOLA; JONES, 2001). O classificador é treinado com alguns milhares de visualizações de amostra de rosto (imagens positivas) e imagens arbitrárias (imagens negativas) que são redimensionadas para o mesmo tamanho. A segunda parte será a detecção, quando o algoritmo pesquisará em cada local do *frame* de entrada aplicando o classificador em cascata de cada estágio treinado em uma janela de pesquisa dinâmica para procurar características de um rosto humano (VIOLA; JONES, 2001).

O detector de objetos Viola e Jones é um classificador forte composto de muitos classificadores fracos. Esses classificadores fracos são simples para permitir um processamento rápido, realizando apenas uma soma ponderada. Combinando esses classificadores fracos em uma cascata, um classificador final é criado, sendo capaz de eliminar as regiões não faciais rapidamente e mantendo quase todas as regiões da face.

Para processar imagens em tempo real, os estágios anteriores em cascata tentam remover apenas as janelas que têm baixa probabilidade de serem rostos. À medida que os estágios em cascata progridem, um número maior de classificadores, cada vez mais complexos, é usado para reduzir as classificações de falsos positivos. Se um dos estágios em cascata classificar uma janela como não facial, nenhum outro estágio a processará. Isso garante que apenas as regiões com alta probabilidade de serem faces serão submetidas a cálculos mais intensivos, como ilustrado pela Figura 6.

Figura 6 – Viola e Jones: Exemplo de cascata na detecção de faces.



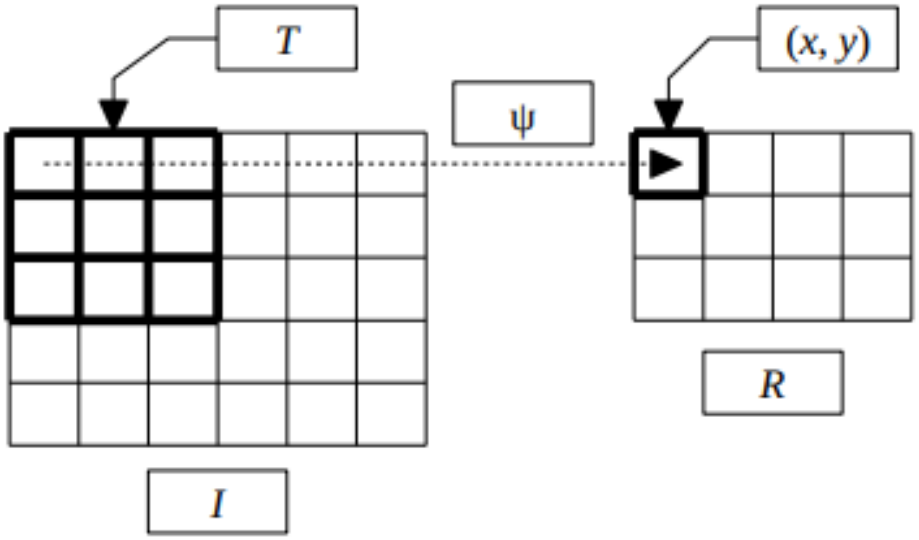
Fonte: Adaptado (HARVEY, 2009b)

2.3.2 *Templating Matching*

Templating Matching é um método simples de ser implementado, flexível, eficiente e de baixo custo de processamento, presente na biblioteca OpenCV. Usando como exemplo a Figura 7, onde I é a imagem de entrada onde a busca será realizada, T é o modelo a ser procurado e R é o resultado. Esse método trabalha de forma a percorrer a imagem I comparando *template* T com cada posição (x, y) de I. O resultado R é menor do que a imagem de busca I, e o resultado do casamento é colocado no canto superior esquerdo da ocorrência do modelo em R (USP, 2017).

O método realiza uma convolução da máscara sobre toda a imagem e retorna a posição onde existe a menor diferença e o valor de tal diferença, semelhante ao cálculo da diferença ponto a ponto entre as imagens. A máscara possui as mesmas dimensões da imagem e a saída dependes do tamanho das entradas, se a imagem de entrada for do tamanho

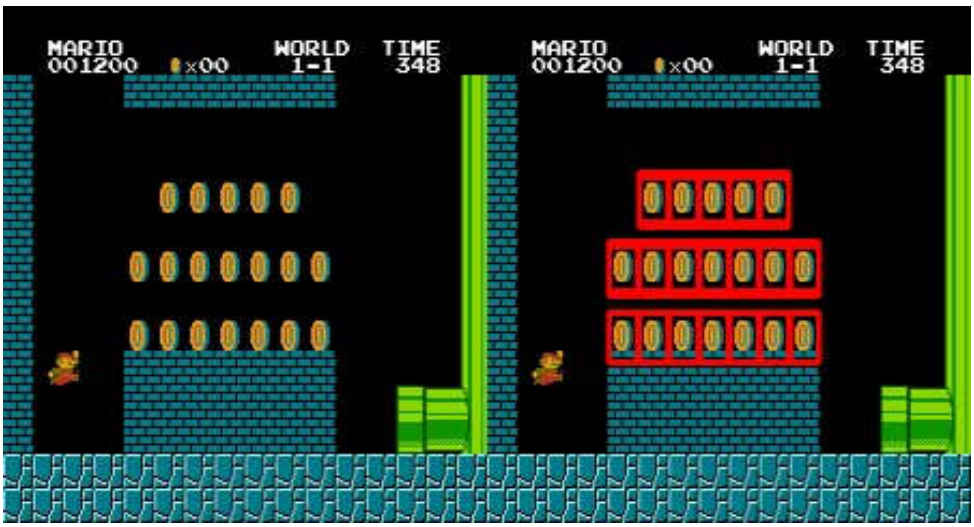
Figura 7 – Passo a passo do método *Templating Matching*.



Fonte: (USP, 2017)

$(w \times h)$ e a imagem do modelo for do tamanho $(w \times h)$, a imagem de saída terá o tamanho de $(W - w + 1, H - h + 1)$. Na Figura 8, é possível observar o método em funcionamento, quando foi lhe passado um um modelo onde o objetivo era identificar as moedas presentes no cenário (OPENCV, 2020).

Figura 8 – Resultado método *Templating Matching*.



Fonte: (OPENCV, 2020)

2.3.3 YOLO (*You Only Look Once*)

Após o seu lançamento em 2015, o YOLO foi logo reconhecido como uma técnica inovadora, pois através de uma abordagem totalmente nova foi capaz de obter uma precisão igual ou superior ao dos outros métodos de detecção de objetos da época, porém com uma velocidade de detecção muito superior. YOLO é um método de detecção de objetos de passada única (*single pass*) que utiliza uma rede neural convolucional como extrator de características (*features*).

Diferente de algoritmos anteriores de detecção de objetos, como R-CNN ou *Faster* R-CNN, ele apenas precisa olhar pela imagem uma única vez para enviar para a rede neural, por isso ele recebe esse nome (You *Only Look Once* – "Você só olha uma vez"). Devido a essa característica, o YOLO foi capaz de conseguir uma velocidade na detecção muito maior do que as técnicas concorrentes, sem perder em acurácia. Seu funcionamento é dado nos seguintes passos, ilustrados pela Figura 9:

1. Primeiro, o algoritmo divide a imagem em um grid de $S \times S$ células. Como exemplo usaremos 13×13, porém esse tamanho pode mudar. Para as versões mais recentes por exemplo tem-se preferido utilizar um *grid* de 19×19 (BOCHKOVSKIY; WANG; LIAO, 2020);

2. Cada uma dessa células é responsável por fazer a predição de 5 caixas delimitadoras (B), para o caso de haver mais de um objeto naquela célula. Também é retornado a pontuação de confiança que nos diz o quanto de certeza ele tem que aquela caixa delimitadora contém um objeto;

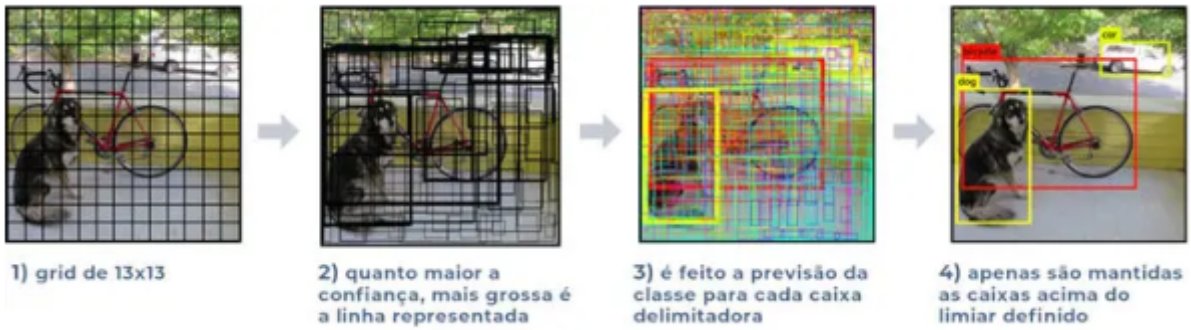
3. Em cada caixa, a célula também faz a previsão de uma classe. Isso funciona como se fosse um classificador: é fornecido um valor de probabilidade para cada uma das classes possíveis. O valor de confiança para a caixa delimitadora e a predição da classe são combinados em uma pontuação final, que vai nos dizer a probabilidade dessa caixa conter um objeto específico. No exemplo, o grid é 13×13, o que no final resulta em 169 células. Para cada uma dessas células são detectados 5 caixas delimitadoras, o que resulta em 845 no total;

4. A maioria dessas caixas terá um valor de confiança extremamente baixo, então por isso geralmente se considera apenas as caixas cuja pontuação final seja 30% ou mais. Esse valor de 30% é o limiar, chamado de *threshold*, e ele pode ser alterado dependendo do quão preciso você quer que o detector seja.

2.4 OpenCV

OpenCV é uma biblioteca *open source*, escrita em C e C++ que funciona em Linux, Windows e Mac OS X e que possui interfaces para C, C++, Python, Ruby, Java, MATLAB e outras linguagens. O OpenCV foi originalmente projetado visando eficiência computacional com foco em aplicações de tempo real (BRADSKI; KAEHLER, 2008). OpenCV é a enorme

Figura 9 – Passo a passo do método YOLO.



Fonte: Adaptado (HOLLERWEGER, 2019)

biblioteca de visão computacional, aprendizado de máquina e processamento de imagem e agora desempenha um papel importante na operação em tempo real, o que é muito importante nos sistemas atuais. Ao usá-lo, pode-se processar imagens e vídeos para identificar objetos, rostos ou até mesmo a escrita de um humano. Quando integrado com várias bibliotecas, como *Numpy*, implementado-o em python, é capaz de processar a estrutura de *array* do OpenCV para análise. Para identificar o padrão de imagem e seus vários recursos, usa-se o espaço vetorial e executa-se operações matemáticas nesses recursos. A biblioteca é estruturada em cinco componentes, sendo eles:

1. CV: possui as funcionalidades básicas de PDI e algoritmos de visão computacional;
2. ML: possui algoritmos de aprendizado de máquina, *clustering*, classificação e análise de dados;
3. HighGUI: possui funções relacionadas à construção de interfaces e a captura e armazenamento de imagens e vídeos;
4. CXCORE: núcleo de estruturas e algoritmos básicos do OpenCV;
5. CVAUX: possui algoritmos de visão computacional em fase de experimentação.

Além disso, o OpenCV é utilizado extensivamente por grandes organizações, como Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda e Toyota, ou por *startups* de visão computacional (BRADSKI, 2000).

2.5 COCO (*Common Objects in Context*)

O conjunto de dados *Common Objects in Context* (COCO) é um dos mais populares bancos de dados de reconhecimento de objeto de código aberto usado para treinar programas de aprendizado profundo. Este banco de dados inclui centenas de milhares de imagens com milhões de objetos já rotulados para treinamento (LIN et al., 2015).

Um elemento importante do aprendizado de máquina supervisionado é o acesso a um conjunto de dados grande e bem documentado para aprender. Patrocinado pela Microsoft, o COCO segmenta as imagens em categorias e objetos, ao mesmo tempo que fornece legendas e *tags* de contexto legíveis por máquina. Isso tudo reduz drasticamente o tempo de treinamento básico para qualquer IA que precise processar imagens.

A base de dados COCO possui mais de 300 mil imagens, 1.5 milhão de instâncias de objetos e 80 categorias diferentes de objetos, dentre elas (LIN et al., 2015) : Pessoas, mochilas, mesas, cadeiras, carros, motos, etc.

2.6 Unidade de Processamento Gráfico (GPU)

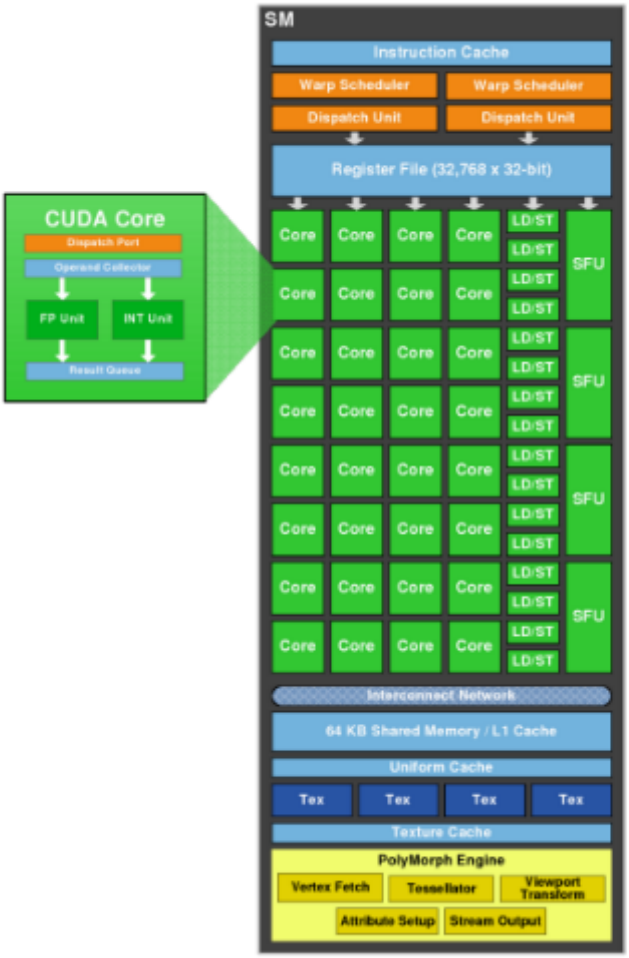
Tradicionalmente, as GPUs (*Graphics Processing Unit*) têm sido usadas como dispositivos de renderização dedicados para computadores e consoles de jogos. Nos últimos anos, tornaram-se capazes de realizar muito mais do que cálculos de gráficos específicos. O *hardware* de renderização especializado oferece uma vantagem para a GPU sobre a CPU ao realizar cálculos altamente paralelos e intensivos em computação.

2.6.1 Arquitetura de uma GPU

Uma distinção relevante entre as GPUs e as CPUs é que, enquanto as CPUs empregam uma significante quantidade de seus circuitos ao controle, a GPU é mais focada em ALUs (*Arithmetic Logical Units*), o que faz com que sejam bem mais eficientes quando executam um *software* paralelo (ZANOTTO; FERREIRA; MATSUMOTO, 2012). Como efeito, a GPU é construída para aplicações com demandas diferentes da CPU: cálculos massivamente paralelos com mais ênfase na taxa de transferência (*throughput*) que na latência.

As GPUs da NVIDIA, por exemplo, são divididas em vários SMs (*Streaming Multiprocessors*), que executam grupos de *threads*, chamados de *warps*. Cada SM é formado por vários núcleos, chamados de CUDA *cores*. Cada CUDA *cores* possui *pipelines* completos de operações aritméticas (ALU) e de pontos flutuantes (FPU - *Floating Pount Unit*). Em um SM, há uma memória *cache* L1 comum somente aos núcleos de um SM, e todos os *cores* de um SM tem acesso à uma memória global. A Figura 10 mostra o SM da arquitetura Fermi da NVIDIA. Uma significativa parte do SM corresponde aos CUDA *cores*, definindo sua especialidade em operações que envolvem muitos cálculos, ao contrário das CPUs, onde a *cache* é predominante.

Figura 10 – *Streaming Multiprocessor* da Arquitetura Fermi.



Fonte: (NVIDIA, 2009)

2.7 Programação GPGPU (*General Purpose Graphics Processing Unit*)

A GPU é um dispositivo poderoso, capaz de processar uma grande quantidade de dados em um curto espaço de tempo. No entanto, é necessário um *software* para programar as ações da GPU e interfaces são necessárias para acessar o *hardware*, ou seja, um *framework*. A seguir serão apresentados alguns *frameworks* utilizados nesta área.

2.7.1 *Frameworks* de Programação GPGPU

Sh e Brook foram duas das primeiras linguagens de alto nível e ambientes de programação para GPUs (BrookGPU). Essas estruturas de programação fornecem um nível de abstração de o *hardware* gráfico para que não exija do programador um conhecimento profundo das texturas da GPU e *shaders* (BUCK et al., 2004).

BrookGPU é um compilador e interpretador em tempo de execução da linguagem de programação *Brook Stream*, e é implementado como uma extensão da linguagem de programação C. Sh é uma linguagem de metaprogramação que é implementada como uma biblioteca C++. BrookGPU e Sh suporta GPUs NVIDIA e ATI em Windows e Linux. Sh foi comercializado e expandido com suporte adicional para CPUs *multicore*, e foi adotado para uso no ATI Stream.

A NVIDIA percebeu a necessidade de um método fácil para programar suas GPUs. Em novembro de 2006, a NVIDIA lançou o CUDA, uma ferramenta de computação paralela de uso geral, com um novo modelo de programação paralela e arquitetura de conjunto de instruções, que aproveita o mecanismo de computação paralela em GPUs NVIDIA para resolver muitos complexos computacionais problemas de uma forma mais eficiente do que em uma CPU (NVIDIA, 2015).

O ATI Stream, desenvolvido pela AMD, é um conceito semelhante ao CUDA da NVIDIA. Ambos fornecem um interface de alto nível para programar em GPUs. ATI Stream utiliza Brook+, um compilador e pacote de tempo de execução para programação GPGPU que fornece controle sobre o dispositivo. ATI Stream é multiplataforma, mas só funciona em GPUs AMDs (AMD, 2021).

A NVIDIA e a AMD oferecem suporte a OpenCL (*Open Computing Language*). OpenCL é o primeiro padrão aberto para programação paralela de uso geral de sistemas heterogêneos. OpenCL suporta não apenas a programação de GPU, mas também uma mistura diversa de CPUs *multi-core*, GPUs e outros processadores paralelos, como DSPs. O OpenCL fornece um estrutura de programação e ambiente mais intimamente relacionado ao CUDA da NVIDIA (NVIDIA, 2021a).

2.7.2 CUDA

Antes das ferramentas de programação explicita para GPUs, era habitual programadores utilizarem do OpenGL (GROUPS, 2021). Trata-se de uma especificação livre para a criação de gráficos, sendo muito utilizada em ambientes UNIX e sistemas operacionais da Apple. Porém, sua estrutura de programação é complicada e provável de gerar erros. Para sanar as dificuldades, a NVidia criou a plataforma CUDA (NVIDIA, 2015), baseada nas linguagens C/C++, o que tornou a plataforma facilmente aceita por programadores familiarizados com a linguagem.

Em programação de CPUs, costuma-se agrupar dados a serem utilizados próximos temporalmente em estruturas, de forma que fiquem próximos espacialmente. Por exemplo, se for feita a soma entre dois vetores x e y, armazenando o resultado em z, cria-se um vetor de estruturas contendo o valor de x, y e z dos elementos correspondentes. Já em programação de GPUs é aconselhável utilizar o formato original dos vetores. Como a soma é executada em paralelo em diversas *threads*, essa organização permite carregar vários elementos de um único vetor com apenas uma transferência, diminuindo, dessa forma, o gargalo. A título de exemplo, na NVIDIA GeForce 285, sete operações de ponto flutuante podem ser executadas por um núcleo de processamento ao mesmo tempo que um *byte* demora para ser transferido da memória externa para a GPU (Aamodt et al., 2018).

Em CUDA, a função a ser executada na GPU recebe o nome de *kernel*. Essa função é responsável por acessar o *hardware* onde ela é executada, N vezes em paralelo em N diferentes CUDA *threads*. Essa função pode receber argumentos como valores ou ponteiros para memórias globais, locais e também possui uma série de constantes definidas que permite uma *thread* identificar qual elemento deve ser processado por ela. Uma função *kernel* é definida utilizando a declaração global e o número de *threads* que serão executadas.

Cada *thread* tem seu ID, que a identifica de forma única e é acessível pelo *kernel* através da variável `threaIdx`. O comando `blockIdx.x` disponibiliza o identificador único do bloco da *thread* atual. A variável `threadIdx.x` possui o identificador único da *thread* atual dentro do bloco e `blockDim.x` é a dimensão do bloco atual. Temos, dessa forma, um identificador único de uma *thread* que permite a ela identificar que elemento processar. Esse identificador é calculado da seguinte maneira:

$$IDthread = blockIdx.x \times blockDim.x + threadIdx.x$$

Cada uma dessas constantes podem ter valores em x, y e z, facilitando o trabalho do programador e sendo organizadas internamente da maneira mais eficiente.

Com CUDA, pode-se criar um número de *threads* muito maior do que o suportado pelo *hardware*. Um controlador de *software* gerencia e cria as *threads* que serão executadas. Como não se tem controle sobre o gerenciador de *threads*, não se pode afirmar que uma *thread* executará primeiro, e por isso deve-se utilizar funções de sincronização dentro do *kernel* quando necessário, fazendo com que seja criada uma barreira para todas as *threads* até que todas cheguem ao ponto de sincronização.

Deve-se seguir alguns comandos e passos para programação em CUDA, a invocação do feita utilizando $<<<>>>$. Depois, definem-se as variáveis como a `threadIdx.x` ou `blockIdx.x`, dependendo do que se deseja fazer. Para se alocar a memória na GPU, primeiramente usa-se a alocação da memória no host (CPU) utilizando o comando em C `malloc()`. Depois, aloca-se a GPU utilizando o `cudaMalloc()`. Em seguida, copia-se o que está na CPU para a GPU utilizando o comando `cudaMemcpy()`. Assim que essas variáveis forem alocadas, deve-se liberar a memória para seu uso futuro. Em C temos comando `free()`, e no CUDA, temos o `cudaFree()`. As *threads* precisam ser sincronizadas se estão trabalhando em um bloco e precisam trabalhar com dados de um iteração anterior. Para isso, utiliza-se a função `synchthreads()` para se colocar uma barreira até que todas as *threads* de um mesmo bloco terminem suas tarefas.

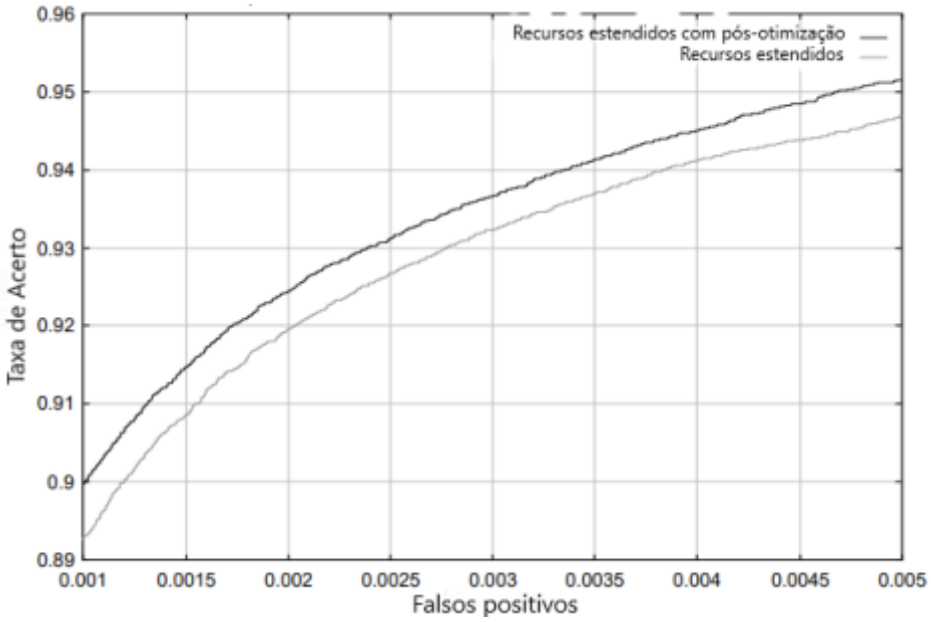
3 Trabalhos Relacionados

“O período de maior ganho em conhecimento e experiência
é o período mais difícil da vida.”
Dalai Lama

As atividades de pesquisa em processamento digital de imagens, mais especificamente na área de detecção de objetos em imagens, ganharam popularidade desde o final dos anos noventa (Wai; Tahir; Chang, 2015). Devido ao potencial e as possibilidades de atribuir ao computador a capacidade do ser humano de interpretar imagens, cresce o número de trabalhos que tentam realizar tal feito. Na literatura, foram encontrados vários trabalhos e alguns são descritos a seguir.

Muitos pesquisadores propuseram abordagens para identificar rostos humanos. Destas abordagens, a detecção de rosto Viola-Jones, que é baseada no Adaboost, mostrou um resultado promissor com 15 quadros por segundo (*fps*) em imagens de resolução 320×288 *pixels* em comparação com as abordagens anteriores, mantendo a mesma a precisão de detecção (VIOLA; JONES, 2001). Posteriormente, Lienhart (LIENHART; MAYDT, 2002) propôs o uso de um conjunto de recursos rotativos do tipo Haar que enriquece os recursos simples do algoritmo de Viola-Jones original, como pode ser observado na Figura 11.

Figura 11 – O estágio de pós-otimização proposto por Lienhart melhora o desempenho da cascata de detecção intensificada em cerca de $12,5\%$



Fonte: Adaptado (LIENHART; MAYDT, 2002)

Outros trabalhos têm sido relatados na literatura na tentativa de acelerar o processo de

detecção de objetos, principalmente na face humana. Isso é para atender a demanda atual por aplicações, como sistema de vigilância por vídeo em tempo real. No entanto, a detecção de faces é uma tarefa demorada, especialmente porque o tamanho da imagem a ser processada aumenta. A abordagem de *software* que usa *multi-threading* na implementação otimizada do OpenCV em sistema baseado em CPU é capaz de atingir 1,78 *frames* por segundo (fps) em imagens de tamanho VGA (640×480) *pixels* (HARVEY, 2009b) e 14,2 fps em imagens menores de resolução 256×192 *pixels*, segundo testes realizados por Shrestha et al. (SHRESTHA; KANSAKAR, 2009).

Por outro lado, trocar a plataforma para *hardware* é um método alternativo para acelerar o algoritmo. Por exemplo, Theocharides et al. (Theocharides; Vijaykrishnan; Irwin, 2006) propuseram uma arquitetura ASIC (*Application-Specific Integrated Circuit*) que explora fortemente o paralelismo do algoritmo de Viola-Jones, paralelizando os acessos aos dados da imagem. Como resultado, ele mostra uma taxa de computação de 52 fps, mas as resoluções da imagem de entrada não são mencionadas. Além disso, Cho et al. (CHO et al., 2009) também apresentaram um sistema de detecção de face baseado em FPGA (*Field Programmable Gate Array*) com classificadores Haar usando *buffers* e captadores de quadros especiais para acelerar o processamento, que é capaz de obter 6,55 fps para imagem VGA, como ilustrado na Tabela 1.

Tabela 1 – Desempenho do sistema FPGA de detecção de rosto com imagens de resolução 640×480 *pixels*.

# de Faces	**Classificador em *Software***	**Classificador em *Hardware***	
		Classificador Único	**Classificador Triplo**
1	2.165 ms (0,46 fps)	189,199 ms (5,28 fps)	133,143 (7,51 fps)
6	2.919 ms (0.34 fps)	254,254 ms (3,93 fps)	146,745 (6,81 fps)
11	3.129 ms (0.31 fps)	260,169 ms (3,84 fps)	152,664 (6,55 fps)

Fonte: Adaptado (CHO et al., 2009)

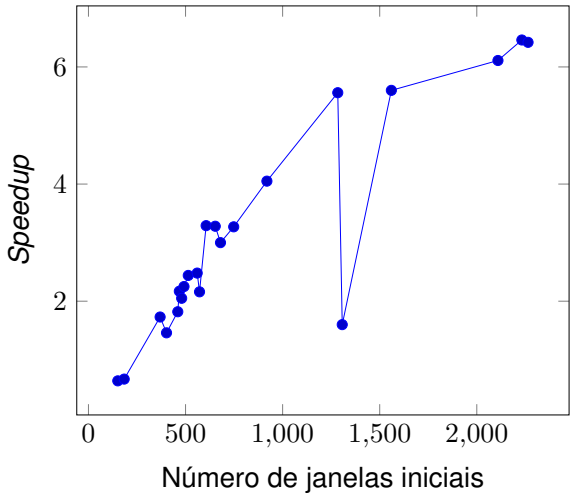
A linguagem de programação em GPU da NVidia (CUDA) está se tornando popular para uso em paralelização massiva devido à sua arquitetura que mantém o código original em C++, o que torna fácil de compreender seu uso. Já existem vários trabalhos relacionados à aceleração do algoritmo de detecção facial Viola-Jones usando CUDA. Por exemplo, as principais implementações de GPU no algoritmo estão rodando a 2,8 fps em um NVidia GTX285 e 4,3 fps em duas NVidia GTX295, com um aumento de desempenho no tempo de detecção de faces de até 6,42x comparado à implementação em CPU tradicional, desenvolvida em OpenCV (HARVEY, 2009b), como pode ser visto pela Tabela 2 e pela Figura 12.

Tabela 2 – Comparação da execução em CPU *versus* GPU do algoritmo Viola e Jones

Largura da Imagem	**Altura da Imagem**	**# de Janelas Iniciais**	**OpenCV 1.1 (ms)**	**Duas GPUs GTX295 (ms)**	***Speedup* Vs. OpenCV**
141	201	151	67,55	106,15	0,64
233	174	184	65,08	96,57	0,67
306	472	369	288,04	166,86	1,73
469	375	402	283,31	194,27	1,46
580	380	460	390,80	214,71	1,82
580	396	468	447,28	205,92	2,17
500	500	480	465,29	227,21	2,05
512	512	492	509,61	226,32	2,25
627	441	514	561,15	230,26	2,44
560	600	560	783,08	316,22	2,48
900	284	572	484,51	224,77	2,16
689	563	606	811,18	246,85	3,29
800	545	653	959,44	292,27	3,28
716	684	680	943,30	314,41	3,00
662	874	748	1119,01	342,56	3,27
1413	465	919	1455,59	359,47	4,05
1500	1107	1284	3885,84	698,38	5,56
1100	1553	1307	12899,67	8080,34	1,60
2022	1138	1560	5472,63	977,19	5,60
2922	1266	2109	8991,16	1471,54	6,11
2916	1587	2232	10970,75	1697,83	6,46
2903	1664	2264	11686,09	1820,16	6,42

Fonte: Adaptado (HARVEY, 2009b)

Figura 12 – *Speedup* CPU versus GPU em relação ao número de janelas iniciais.



Fonte: Adaptado (HARVEY, 2009b)

Analisando-se os resultados obtidos por Harvey, pode-se observar que a implementação CUDA nem sempre aprensenta resultados melhores em OpenCV. O aumento de velocidade depende muito do número inicial de subjanelas de faces detectadas a serem processadas. Conforme este número de subjanelas aumenta, mais trabalho pode ser feito em paralelo em cada estágio. O *speedup* torna-se maior do que $1x$ quando há aproximadamente 275 subjanelas ou um tamanho de imagem de 300×300 *pixels*. Para imagens menores do que isso, não existe paralelismo suficiente para superar a sobrecarga associada à cópia de memória para o dispositivo, compressão de fluxo, lançamentos de *kernel* e sincronização.

Li-chao Sun et. al (Sun et al., 2013) apresentaram um sistema de detecção facial em tempo real baseado no classificador em cascata de Viola-Jones na plataforma CUDA. Os resultados experimentais mostram que o programa CUDA rodando em uma placa de vídeo Nvidia GTX 570 para uma imagem de entrada VGA pode atingir até $6x$ de *speedup* com 9,35 fps em comparação com a versão do CPU, como ilustrado pela Tabela 3.

Tabela 3 – Tempo de execução da detecção facial

Tamanho da Imagem	**CUDA**	**OpenCV**	**CPU**
3680x2070	0,874	3,98	19,599
1920x1080	0,299	0,963	5,097
1280x720	1,71	0,462	2,222
1024x768	0,165	0,414	1,979
800x600	0,129	0,241	1,139
640x480	0,107	0,169	0,715

Fonte: Adaptado (Sun et al., 2013)

Também, Shivashankar J. Bhutekar et. al (BHUTEKAR; MANJARAMKAR, 2018) propuseram uma técnica que processa imagens para detecção e reconhecimento de faces em paralelo na GPU NVIDIA GeForce GTX 770. O algoritmo de detecção de rosto Viola-Jones mostra aproximadamente 3 fps em uma imagem de 700×580 *pixels* na estrutura CUDA em comparação com 0,71 fps na CPU, aprensentado um *speedup* de aproximadamente $4x$ no tempo de reconhecimento facial entre as duas arquiteturas, como pode ser observado pela Tabela 4.

Tabela 4 – Tempo gasto por componente em GPU versus CPU

Tempo (ms)	**Tamanho da Imagem**					
	480x360		**650x400**		**700x580**	
	GPU	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**
Detecção da Face	102	780	160	900	350	1406
Reconhecimento da Face	90	207	127	279	312	300

Fonte: Adaptado (BHUTEKAR; MANJARAMKAR, 2018)

Apesar de todas essas melhorias, ainda supõe-se ser possível ocorrer um aumento na velocidade de detecção de faces em GPU, utilizando-se de *hardwares* mais recentes e versões mais novas de *frameworks* e linguagens citadas.

4 Materiais e Método

“Um bom começo é a metade.”
Aristóteles

Este trabalho tem por característica ser experimental devido à característica da comparação e avaliação de um processo em GPU, com o objetivo de um suposto ganho de desempenho em relação a este processo em CPU. Além disso, é primordial o entendimento dos fundamentos técnicos de visão computacional e processamento de imagens em uma GPU. Para se entender os conceitos necessários, foram utilizados livros teóricos, trabalhos de conclusão de curso (TCCs), dissertações de mestrado, doutorado, e artigos relacionados ao tema. Para uma melhor assimilação das técnicas necessárias foi essencial a busca por materiais, vídeos e tutoriais em repositórios disponíveis na internet.

A fim de se realizar os objetivos propostos por este trabalho foi utilizada a biblioteca OpenCV, que é uma das bibliotecas referências nesta área (Seção 2). Podendo ela ser implementada em diversas linguagens, optou-se pela linguagem de programação Python. Por isso foi utilizada a plataforma de desenvolvimento Pycharme, a plataforma Anaconda para gerenciamento da biblioteca OpenCV e demais bibliotecas discorridas posteriormente.

A codificação, compilação e execução de todo *software* produzido foi realizada em um computador com as seguintes especificações:

- Processador AMD Ryzen 5 5600X 6-Core 3.70 GHz;
- Memória RAM 32GB 3200 MHz;
- SSD Asgard AN3 1TB NVMe;
- EVGA NVIDIA GeForce RTX 2060 KO Gaming 6GB;
- Sistema Operacional Windows 10 x64.

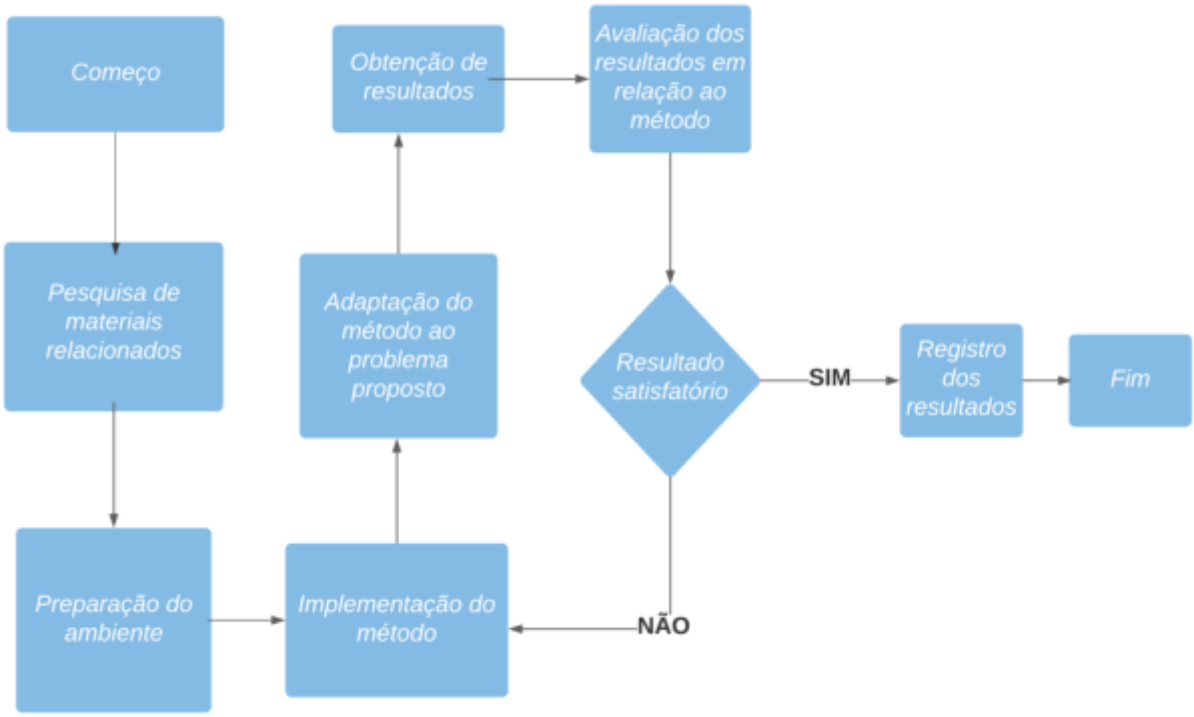
Os procedimentos metodológicos são organizados nas seguintes etapas:

1. Pesquisar e reunir trabalhos relacionados com o tema proposto para que seja realizada uma estruturação de informações sobre os principais métodos e técnicas sobre detecção de objetos em GPU;
2. Pesquisa e obtenção de bases de dados pertinentes ao tema proposto em repositórios *online*;
3. Escolha e implementação o método para detecção e rastreamento de objetos que será utilizado no processo em GPU e CPU;

4. Comparação dos resultados em CPU e GPU;
5. Avaliação dos resultados obtidos.

Na Figura 13 são ilustrados os passos das atividades realizadas para a conclusão do experimento deste trabalho.

Figura 13 – Procedimentos metodológicos



Fonte: elaborada pelo autor

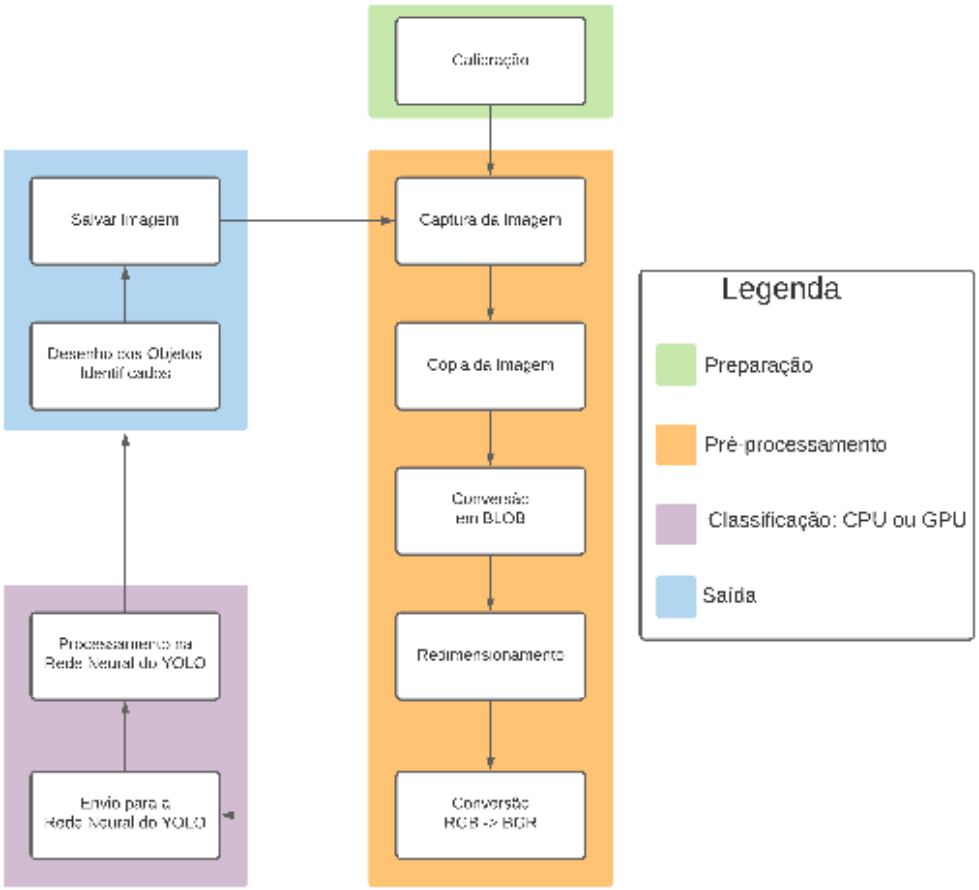
5 Especificação do Método

“Descobrir consiste em olhar para o que todo mundo está vendo e pensar uma coisa diferente.”
Roger Von Oech

Partindo do ponto da análise de todo o referencial teórico e os trabalhos já realizados na área de detecção de objetos em imagens digitais, foi escolhido e implementado um método capaz de reconhecer objetos de até 80 classes diferentes utilizando a biblioteca YOLO. Também foi utilizado a biblioteca OpenCV que permite a utilização de CUDA na detecção.

Um visão geral deste método pode ser visualizado no fluxograma da Figura 14. Nas seções a seguir serão descritas todas as etapas deste fluxograma com o objetivo de esclarecer o funcionamento do método proposto.

Figura 14 – Fluxograma do método implementado.



Fonte: elaborada pelo autor

5.1 Preparação

5.1.1 Calibração

Na Calibração foi definido quais são os conjuntos de objetos a serem detectados na imagem digital. Tendo como base que a etapa de treinamento de algoritmos de detecção de objetos é custosa e toma muito tempo (ZHAO et al., 2019), optou-se por utilizar o modelo já existente e pré-treinado do YOLO, que foi treinado com a base de dados COCO e é capaz de detectar até 80 classes de objetos diferentes.

5.2 Pré-Processamento

Após a preparação do ambiente e definição das classes de objetos a serem identificadas, é necessário definir qual imagem será analisada a fim de se reconhecer os objetos presentes nela.

O principal objetivo desta etapa é capturar e manipular a imagem de forma que o resultado obtido seja mais adequado do que o original para uma aplicação específica. É importante ressaltar que em processamento digital de imagens, “ser adequado” é algo bastante subjetivo, pois o processo de adequação de uma imagem está diretamente relacionamento ao problema em que ela está envolvida (GONZALEZ; WOODS, 2008).

5.2.1 Captura da Imagem

Neste passo é o momento que se escolhe qual o padrão de imagem que será analisada e, por convenção, espera-se uma imagem no padrão RGB, que é o tipo mais comum dentre o meio de imagens digitais (FILHO; NETO, 1999). Esta imagem pode ser individual ou pertencer a um conjunto de imagens sequenciais, o que caracteriza o rastreamento de objetos em um vídeo, por exemplo.

Após a obtenção desta imagem é necessário que sejam aplicadas algumas transformações para atender aos padrões necessários do YOLO e OpenCV antes de avançar para as próximas etapas do método.

5.2.2 Cópia da Imagem

Após a etapa de aquisição, tendo em vista que algumas transformações serão aplicadas na imagem, uma cópia deste imagem é feita para que as informações essenciais da imagem não se percam e se possa ter acesso à imagem original durante a fase de desenho das caixas delimitadoras dos objetos identificados pelo YOLO.

5.2.3 Conversão em BLOB (*Binary Large Object*)

Para que a imagem se adeque aos padrões convencionados pelo YOLO, uma primeira transformação deve ser feita, de modo que o formato da imagem seja do tipo BLOB (*Binary Large Object*) ou Objeto Grande Binário. Um blob é um tipo de dados que pode armazenar dados binários. Isso é diferente da maioria dos outros tipos de dados usados em bancos de

dados, como inteiros, números de ponto flutuante, caracteres e *strings*, que armazenam letras e números (ORACLE, 1977).

Como os blobs podem armazenar dados binários, eles podem ser usados para armazenar imagens ou outros arquivos multimídia. Por exemplo, um álbum de fotos pode ser armazenado em um banco de dados usando um tipo de dados de blob para as imagens e um tipo de dados de *string* para as legendas. Além disso, como os blobs são usados para armazenar objetos como imagens, arquivos de áudio e videoclipes, eles geralmente requerem muito mais espaço do que outros tipos de dados. A quantidade de dados que um blob pode armazenar varia dependendo do tipo de banco de dados, mas alguns bancos de dados permitem tamanhos de blob de vários *gigabytes*.

5.2.4 Redimensionamento

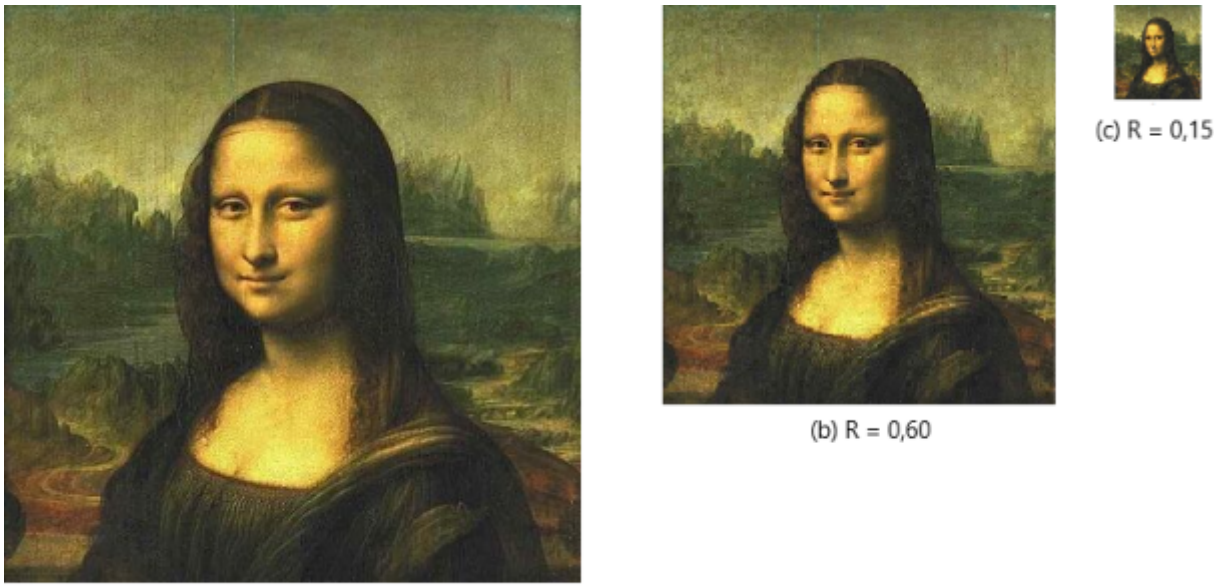
Grande parte das operações de redimensionamento de imagens envolvem funções aplicadas em todos os seus *pixels*. Em geral, estas operações possuem uma complexidade de no mínimo $O(N \times M)$ onde N é o número de *pixels* verticais e M é o número de *pixels* horizontais da imagem (GONZALEZ; WOODS, 2008). Dependendo do tamanho da imagem ou da complexidade das operações realizadas, o custo computacional pode exceder o desejado.

Uma das formas mais usuais de se diminuir este custo é redimensionar a imagem por um fator R que esteja entre 0 e 1, ou seja, reduzir o tamanho da imagem. Uma imagem de tamanho $N \times M$ que for redimensionada por um fator R passa a ter as dimensões $(N.R) \times (M.R)$ consequentemente, as operações aplicadas à imagem redimensionada passam a ter complexidade de no mínimo $O((N.R) \times (M.R))$, diminuindo em $(1-R)\%$ o custo computacional.

A escolha de um valor para o fator R é uma tarefa delicada. Uma redução brusca do tamanho da imagem, onde ser atrativa por diminuir o custo computacional, mas ao mesmo tempo pode-se perder detalhes de interesse na imagem resultante por ter ficado pequena demais. Analisando a Figura 15, percebe-se que a imagem reduzida por um fator maior (Figura 15c) ficou pequena a ponto de não ser possível distinguir suas características a olho nu, o mesmo se estende ao computador durante a fase de classificação. Enquanto isso, na imagem reduzida por um fator menor (Figura 15b) ainda é possível identificar essas características. Dessa maneira, o ideal deve ser tentar encontrar um determinado fator R pequeno o suficiente que reduza ao máximo o custo computacional, desde que na imagem reduzida ainda seja possível identificar plenamente suas características de interesse. Por fim, o fator R deve ser armazenado para que no fim da execução do algoritmo, as posições dos objetos encontrados possam ser recalculadas e relacionadas à imagem original e não à imagem redimensionada.

Dessa forma, após vários testes, um padrão empírico de $R = 1/255$ foi adotado e aplicado neste trabalho, de forma que o custo computacional é reduzido e a precisão do algoritmo é mantida.

Figura 15 – Redimensionamentos por fatores R diferentes.



Fonte: Elaborada pelo autor

5.2.5 Conversão para BGR

Após serem feitas as etapas de obtenção da imagem e do seu redimensionamento, é preciso convertê-la do padrão RGB *(Red, Green, Blue)* para o padrão BGR *(Blue, Green, Red)*. Esta conversão é necessária pois o padrão adotado pela biblioteca OpenCV assume que as imagens estão utilizando a ordem de canal BGR (INTEL; GARAGE; ITSEEZ, 2000). Para resolver essa discrepância, é feita uma troca entre os canais R e B na imagem.

5.3 Classificação: CPU ou GPU

5.3.1 Envio para o YOLO

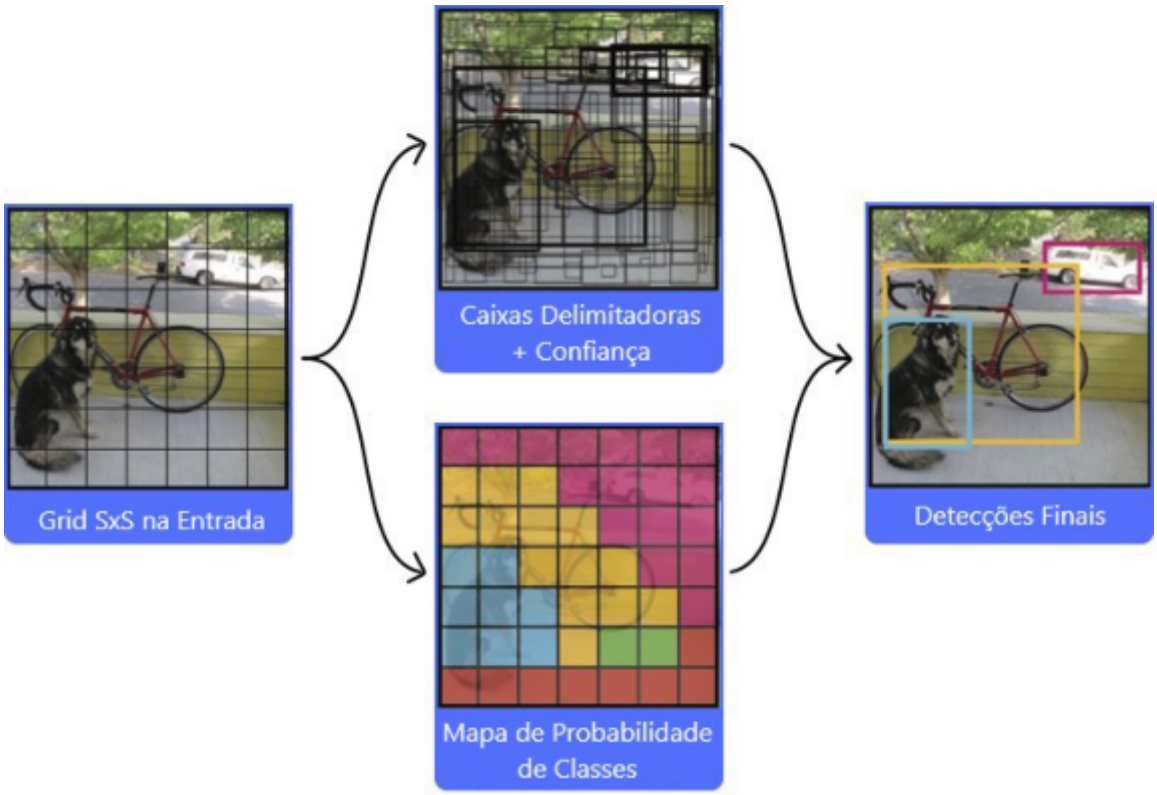
Após a fase de pré-processamento da imagem ter sido concluída e a imagem ter sido devidamente pré-processada, chega-se ao passo do envio à rede neural do YOLO para reconhecimento e classificação dos objetos contidos nela.

5.3.2 Processamento do YOLO

Como visto anteriormente na Seção 2, o funcionamento do processamento do YOLO é feito de forma que uma única rede neural é aplicada na imagem completa (passada única). Exemplificado pela Figura 16, essa rede primeiramente divide a imagem em regiões (*Grid* $S \times S$) e prevê caixas delimitadoras e probabilidades para cada região da imagem de entrada. Essas caixas delimitadoras são ponderadas pelas probabilidades previstas e por fim é executada o NMS (*Non-Max Suppression*) através do OpenCV, ou seja, como a maioria dessas

caixas terá um valor de confiança extremamente baixo, é executado um limiar de probabilidade cuja as caixas que serão consideradas devem ter sua probabilidade acima deste limiar.

Figura 16 – Processamento do YOLO



Fonte: Adaptado (BLAGOJEVIC, 2018)

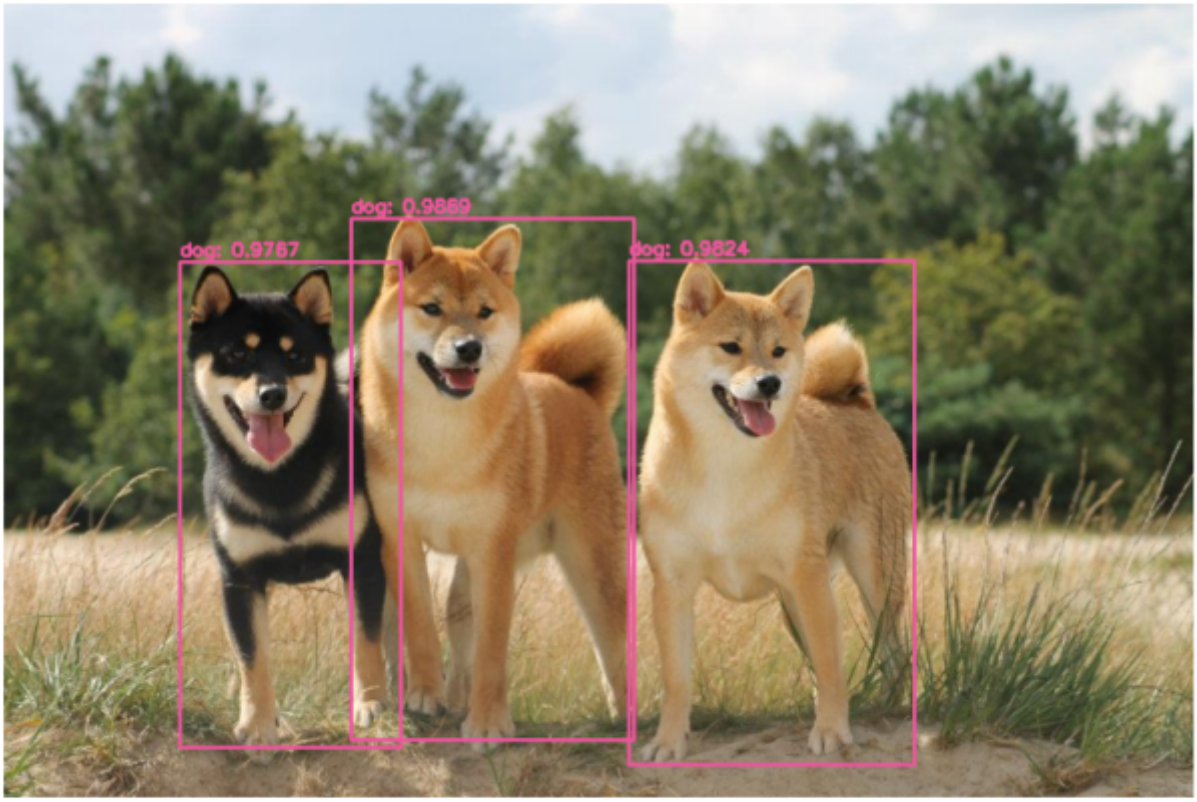
5.4 Saída

5.4.1 Desenho dos Objetos Identificados

Ao fim do processamento do algoritmo do YOLO, é gerado uma lista com as informações de todos os objetos identificados, ou seja, sua altura, largura, e posição (x, y) dentro da imagem original, bem como sua confiança de pertencer às 80 classes previamente definidas dentro do conjunto da base de dados do COCO.

Dessa forma, as informações de cada coordenada do objeto é convertida para coincidir com os tamanhos da imagem original, re-utilizando o fator de redimensionamento R. Após obter a localização do objeto na imagem original, as caixas delimitadoras são desenhadas na imagem, bem como a probabilidade do objeto, utilizando as funções de manipulação de imagem presentes no OpenCV, como pode ser observado e exemplificado pela Figura 17.

Figura 17 – Exemplo de detecção com YOLO.



Fonte: Elaborado pelo autor

5.5 Rastreamento de Objetos

Partindo do pressuposto que o rastreamento de objetos equivalente ao reconhecimento de objetos de forma contínua, ou seja, para realizá-lo basta que todas as etapas do método especificado (exceto as de preparação) sejam realizadas continuamente utilizando uma sequência de imagens como fonte de dados. Estas podem ser em formato de vídeo ou providas por alguma fonte de imagens em tempo real (câmera, *webcam*, etc).

Especificadamente para o segundo caso, existe a necessidade de que o método especificado apresente um certo nível de desempenho que atenda a uma determinada frequência de captura de imagens (*frames* por segundo). Dessa forma, para facilitar a visualização do resultado do processamento dos *frames* do vídeo de entrada, os *frames* analisados no método especificado são salvos em um novo arquivo de vídeo que contém a quantidade frames por segundo gravada no topo esquerdo de cada *frame*, calculada por $1/(tempo)$, em que o $tempo$ é a quantidade de segundos que o YOLO levou para processá-lo, como demonstra a Figura 18.

Figura 18 – Exemplo de detecção em Vídeos com YOLO

Fonte: Elaborado pelo autor

5.6 Processamento em GPU

Devido ao fato de se ter escolhido a biblioteca OpenCV neste método, a conversão da implementação feita em CPU para GPU é feito de forma que após a recompilação da biblioteca para adicionar os recursos necessários para a execução em CUDA, é preciso adicionar as funcionalidades do contidas no OpenCV: *setPreferableBackend()* e *setPreferableTarget()* no algoritmo, que faz com que o sistema operacional execute o processamento da rede neural do YOLO na GPU.

6 Implementação e Avaliação do *softwares* por Método: CPU e GPU

“A persistência é o menor caminho do êxito.”
Charles Chaplin

6.1 Implementação do Método

Objetivando validar o funcionamento e a eficiência do método de reconhecimento e rastreamento de objetos proposto por este trabalho, desenvolveu-se duas versões do *software*, implementados tanto em CPU quanto GPU, seguindo todos os passos e recomendações especificados. Estes *softwares* são capaz de analisar imagens únicas ou imagens sequenciais em formato de vídeo. De início foi feita toda a configuração do ambiente, de forma que o método escolhido possa ser implementado e executado da maneira correta. Os seguintes *softwares* e bibliotecas foram utilizadas em seu desenvolvimento:

- Anaconda versão 3.2020.11, que provê um ambiente de execução em Python (ANACONDAINC, 2012);
- CUDA Toolkit versão 11.2, que permite a execução de CUDA no ambiente (NVIDIA, 2007);
- NVIDIA CUDA *Deep Neural Network* (cuDNN) versão 11.0, que é uma biblioteca acelerada por GPU para redes neurais profundas (NVIDIA, 2014);
- OpenCV versão 4.4.0, uma biblioteca para visão computacional (INTEL; GARAGE; ITSEEZ, 2000);
- OpenCV Contrib versão 4.4.0, para implementação de módulos extras ao OpenCV original, que são necessários para a execução da rede neural do YOLO (OPENCV, 2014);
- Pycharm versão 2020.3.5, uma IDE voltada para a linguagem Python (JETBRAINS, 2010);
- CMake versão 3.20, que visa facilitar a compilação de *softwares* (KITWARE, 2000).

Por padrão, a biblioteca OpenCV não contém os módulos de suporte à GPU instalados, então foi necessário fazer a recompilação toda a biblioteca, adicionando os módulos necessários para o funcionamento do CUDA, através do CMake. A escolha do OpenCV deu-se pelo fato dele ser umas das biblioteca de processamento de imagens mais bem estabelecidas e suportadas disponíveis atualmente na computação. Todos os algoritmos necessários para realizar as etapas do processo especificado estão implementados e documentados na biblioteca.

A linguagem Python foi escolhida por ser uma das linguagens mais documentadas do OpenCV e o Anaconda devido à possibilidade de criar diferentes ambientes virtuais de desenvolvimento em Python.

Os parâmetros do software desenvolvido são:

- Coeficiente de redimensionamento $R = 1/255$;
- *Threshold*: 0.5;
- Formato dos vídeos de entrada: MP4;
- Formato dos vídeos de saída: AVI;
- Tamanho padrão da imagem de entrada na rede neural: 416×416.

6.2 Testes em Imagens

Para validar os software desenvolvidos, foram realizados testes utilizando diversas imagens com resoluções diferentes em padrão RGB, providas pelo repositório Unsplash que dispõe de imagens grátis para *download* de resoluções em até 4k (3840×2160 *pixels*) (UNSPLASH, 2019). Estes vídeos são compostos por objetos aleatórios de interesse, que estão contidos nos conjuntos de objetos da base de dados COCO. A relação dos objetos utilizados nos testes pode ser vista na Tabela 5.

Tabela 5 – Quantidade de imagens analisados por resolução.

Categoria	**Resolução (pixels)**	**Quantidade**
4k	3840 × 2160	3
1080p	1920 × 1080	3
720p	1280 × 720	3
480p	852 × 480	3
240p	426 × 240	3

Fonte: Elaborada pelo autor.

Para cada imagem (IMG) testada registra-se o tempo de resposta médio (TRM), tanto em GPU quanto em CPU, que consiste no tempo entre a cópia da imagem até o término da criação da lista de objetos reconhecidos na imagem. Este valor é utilizado para estimar o desempenho do processo implementado. Para que esta avaliação de desempenho seja confiável, cada imagem foi processada dez vezes afim de se obter um tempo de resposta médio. Além disso, todas as análises incluem tentativas de classificação de qualquer tipo de objeto presente na base de dados COCO, como mostra o exemplo da Figura 19.

Figura 19 – Exemplo de detecção em imagens com YOLO utilizando a base de dados COCO

Fonte: Elaborado pelo autor

6.2.1 Resultados

Os resultados dos processamentos obtidos nas execuções das versões em CPU e GPU são exibidos nas Tabelas 6 e 7.

Tabela 6 – Comparação do TRM entre o processamento de imagens em CPU e GPU.

Categoria	**IMG**	**TRM CPU (ms)**	**TRM GPU (ms)**
4k	1	168,63	30,10
	2	169,49	30,47
	3	168,75	30,63
1080p	1	168,06	29,94
	2	166,38	30,04
	3	168,72	29,58
720p	1	167,22	28,96
	2	167,50	28,88
	3	168,35	28,95
480p	1	167,78	26,48
	2	168,06	25,75
	3	167,76	26,55
240p	1	167,22	25,38
	2	167,50	24,89
	3	167,50	25,08

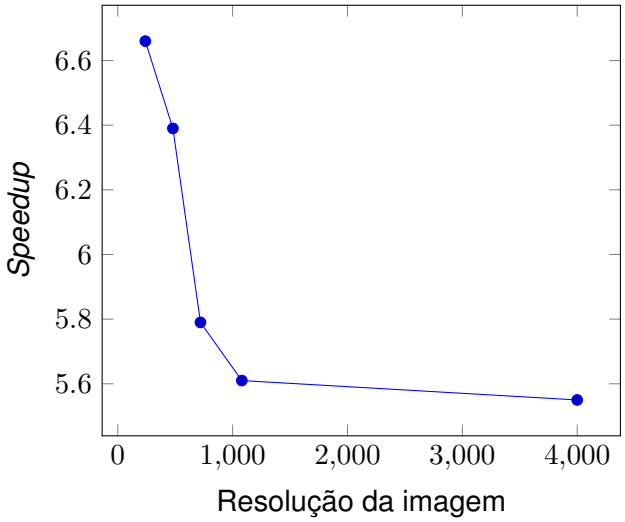
Fonte: Elaborada pelo autor.

Tabela 7 – Comparação da média do TRM entre o processamento de imagens em CPU e GPU.

Categoria	**TRM Médio CPU(ms)**	**TRM Médio GPU(ms)**	***Speedup***
4k	168,95	30,40	5,55 (555%)
1080p	167,72	29,85	5,61 (561%)
720p	167,69	28,93	5,79 (579%)
480p	167,86	26,26	6,39 (639%)
240p	167,40	25,11	6,66 (666%)

Fonte: Elaborada pelo autor.

Figura 20 – *Speedup* GPU versus CPU em relação à resolução da imagem.



Fonte: Elaborado pelo autor

Analisando os dados apresentados na Tabela 7 e na Figura 20, nota-se que o *software* em GPU obteve resultados satisfatórios dado ao que se estava proposto a fazer. A implementação em GPU conseguiu reduzir acima de $5,5\times$ a quantidade de tempo para cada categoria de objeto e obter até $6,6\times$ de *speedup* em relação à implementação tradicional feita em CPU.

Além disso, explorando mais à fundo os resultado obtidos, percebe-se uma peculiaridade nos *speedups*, de forma que o TRM médio de detecção em CPU se mantém quase constante enquanto o tamanho da imagem diminui, já o TRM médio de detecção em GPU diminui enquanto o tamanho da imagem diminui. Isso se deve ao fato de que a implementação feita em Python e OpenCV do YOLO sempre redimensionar a imagem antes de enviá-la para a rede neural. Este processo sendo feito em CPU sempre será sequencial e portanto, em geral, levará aproximadamente o mesmo tempo independente do tamanho da imagem de entrada. Já a execução em GPU paraleliza este redimensionamento, o que faz com que o tempo de redimensionamento da imagem seja menor à medida que o tamanho da imagem diminui, já que haverá cada vez menos *pixels* necessários para calcular, e com isso o *speedup* da GPU versus CPU aumenta conforme o tamanho da imagem diminui.

6.3 Testes em Imagens Sequenciais

Para demonstrar que os algoritmos desenvolvidos e implementados em CPU e GPU são capazes de reconhecer imagens continuamente, foram realizados testes em imagens sequenciais. Essas imagens equivalem aos *frames* capturados em tempo real por um vídeo, provido pelo repositório Videvo que dispõe de vídeos grátis para *download* de resoluções em até 4k (3840×2160 *pixels*) (VIDEVO, 2015). As mesmas configurações de resolução dos testes em imagem foram utilizadas neste caso, como mostra a Tabela 8.

Tabela 8 – Quantidade de vídeos analisados por resolução.

Categoria	**Resolução (pixels)**	**Quantidade**
4k	3840 × 2160	3
1080p	1920 × 1080	3
720p	1280 × 720	3
480p	852 × 480	3
240p	426 × 240	3

Fonte: Elaborada pelo autor.

Neste contexto, um conjunto de *frames* capturados no vídeo em um determinado período de tempo será denominado de sequência de imagens. Então, para cada vídeo (VID), registra-se:

- *Frames* Processados (FP): número de *frames* contidos no vídeo;
- Tempo de Processamento (TP): o tempo (em segundos) de duração do processamento completo do vídeo;
- *Frames* por Segundo (FPS): média de *frames* processados por segundo do vídeo, calculado por FP/TP.

No total foram analisadas 5 categorias de vídeo, cada uma com características equivalentes a uma das 5 primeiras categorias dos testes em imagem. Em cada *frame* do vídeo foi gravado qual a atual contagem de FPS para manter o registro do tempo de processamento dos *frames*, como pode ser observado no canto superior esquerdo da Figura 21.

Figura 21 – Exemplo de detecção em um *frame* do vídeo com YOLO utilizando a base de dados COCO.

Fonte: Elaborado pelo autor

6.3.1 Resultados

Os resultados dos processamentos obtidos nas execuções das versões em CPU e GPU são exibidos nas Tabelas 9 e 10.

Tabela 9 – Comparação de FPS entre o processamento de vídeos em CPU e GPU.

Categoria	**VID**	**FP**	**TP CPU(s)**	**TP GPU(s)**	**FPS CPU**	**FPS GPU**
4k	1	375	62,81	11,66	5,97	32,16
	2	370	61,72	11,14	5,99	33,20
	3	357	59,67	10,77	5,98	33,15
1080p	1	598	99,88	17,91	5,99	33,39
	2	226	37,62	6,82	6,01	33,14
	3	191	32,19	5,68	5,93	33,60
720p	1	191	31,87	5,48	5,99	34,87
	2	275	46,05	7,93	5,97	34,67
	3	328	54,89	9,48	5,98	34,59
480p	1	312	53,39	7,99	5,84	39,05
	2	317	53,22	8,17	5,96	38,81
	3	421	70,55	10,88	5,97	38,68
240p	1	312	52,36	7,80	5,96	40
	2	317	53,08	8,01	5,97	39,58
	3	421	70,76	10,55	5,95	39,90

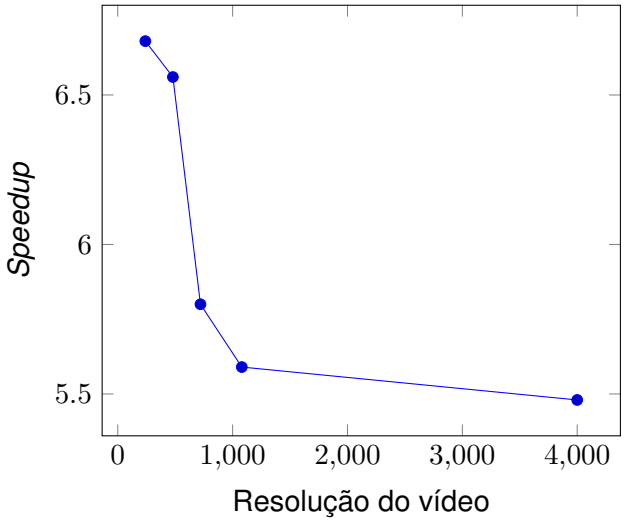
Fonte: Elaborada pelo autor.

Tabela 10 – Comparação de FPS médio entre o processamento de vídeos em CPU e GPU

Categoria	**FPS Médio CPU(ms)**	**FPS Médio GPU(ms)**	***Speedup***
4k	5,98	32,83	5,48 (548%)
1080p	5,97	33,37	5,59 (559%)
720p	5,98	34,71	5,80 (580%)
480p	5,92	38,85	6,56 (656%)
240p	5,96	39,82	6,68 (668%)

Fonte: Elaborada pelo autor.

Figura 22 – *Speedup* GPU versus CPU em relação à resolução do vídeo.



Fonte: Elaborado pelo autor

Observando os resultados apresentados na Tabela 10 e na Figura 22, nota-se que o algoritmo realiza o rastreamento de objetos de forma satisfatória e esperada, já que previamente o resultado do processamento de imagens foi positivo. A mesma inferência para o *speedup* citada no processamento de imagens é válida para o processamento de vídeos, em que o tempo do processamento em CPU se mantém praticamente constante e na GPU aumenta conforme o tamanho da imagem diminui, e dessa forma o *speedup* aumenta quando a resolução do vídeo diminui.

Das 5 categorias analisadas, todas conseguiram manter um ganho acima de $5,48\times$ em FPS, chegando em até $6,68\times$ de *speedup* em relação à implementação feita em CPU do algoritmo. O algoritmo em questão apresentou bons resultados em GPU, pois o FPS dos testes ultrapassaram, em todos as categorias analisadas, a quantidade de FPS usualmente utilizadas em capturas de vídeos em tempo real, que é de 30 FPS.

7 Conclusão

“As palavras fogem quando precisamos delas e sobram quando não pretendemos usá-las.“
Carlos Drummond de Andrade

Com base nos estudos e nas pesquisas realizadas, este trabalho propôs uma implementação de um método capaz de reconhecer e rastrear objetos e a compara suas execuções em CPU e GPU. Este método descreve um mecanismo que utiliza do algoritmo YOLO para a detecção de objetos e é capaz de reconhecer até 80 classes diferentes de objetos em uma imagem ou em uma sequência de imagens, através do conjunto de dados COCO. Deste modo, o objetivo geral deste trabalho foi alcançado.

Os objetivos específicos também foram alcançados. A implementação do método especificado foi capaz de reconhecer objetos destas 80 classes, tanto em imagens quanto em vídeo. A eficiência do método proposto foi demonstradas por meio dos testes realizados na Seção 6, que evidenciaram o fato da implementação do método em GPU obter um ganho de processamento de no mínimo $5,48\times$ em relação a implementação do mesmo método em CPU.

Entretanto, o desenvolvimento desta pesquisa apresentou algumas dificuldades. Uma das mais evidentes está relacionada à configuração do ambiente local para que a execução dos algoritmos implementados tivesse êxito, mais especificamente durante a recompilação da biblioteca OpenCV, que foi complexa devido à alguns erros de versão de bibliotecas do Python e Anaconda, para que a mesma tivesse suporte à GPU e acesso aos módulos do CUDA.

Além disso, a preocupação em implementar o método de maneira eficiente em GPU levou à investigação de técnicas que reduzem a complexidade do reconhecimento de objetos e consequentemente, seu custo computacional. Entretanto, a busca de soluções para esses problemas enriqueceram o desenvolvimento deste trabalho.

De forma geral, a especificação de um método de natureza eficiente em GPU é relevante para incentivar e alavancar trabalhos futuros e assim, ampliar a quantidade de problemas e situações capazes de serem beneficiados por este recurso computacional. Deste modo, as principais contribuições deste trabalho são:

- Levantamento, avaliação e documentação das etapas do processamento digital de imagens;
- Especificação e implementação de um método de reconhecimento e rastreamento de objetos de baixo custo computacional;
- Avaliação do YOLO, OpenCV e COCO como ferramentas computacionais auxiliares no processamento digital de imagens;

- Implementação de algoritmos do método de reconhecimento de objetos que podem ser aplicados em cenários reais.

7.1 Trabalhos Futuros

Entre os trabalhos futuros, considerados para a extensão deste projeto, estão:

- Propor alternativas para a detecção e rastreamento de objetos em imagens e sequência de imagens;
- Implementar e avaliar a implementação em GPU do método utilizando outras ferramentas computacionais;
- Propor uma implementação que torne o tempo de treinamento do método mais eficiente;
- Realizar estudos de caso em cenários reais da utilização do método em GPU para a detecção e rastreamento de objetos;
- Realizar a avaliação da métrica: precisão dos objetos detectados.

Referências

Aamodt, T. M. et al. *General-Purpose Graphics Processor Architecture*. [S.l.: s.n.], 2018. Citado na página 29.

ALBUQUERQUE, M.; ESQUEF, I. Image segmentation using nonextensive relative entropy. *Latin America Transactions, IEEE (Revista IEEE America Latina)*, v. 6, p. 477 – 483, 10 2008. Citado na página 21.

ALVES, G. T. M.; GATTASS, M.; CARVALHO, P. C. P. Um estudo das técnicas de obtenção de forma a partir de estéreo e luz estruturada para engenharia. In: DEPARTAMENTO DE INFORMáTICA, PONTIFíCIA UNIVERSIDADE CATóLICA DO RIO DE JANEIRO, Rio de Janeiro. [S.l.], 2005. Citado na página 14.

AMD. *ATI Stream*. 2021. Disponível em: <https://developer.amd.com/>. Citado na página 29.

ANACONDAINC. *Anaconda*. 2012. Disponível em: <https://www.anaconda.com/>. Citado na página 44.

BHUTEKAR, S. J.; MANJARAMKAR, A. K. Parallel face detection and recognition on gpu. *Department of Information Technology (Nvidia Research Lab)*, p. 6, 2018. Citado na página 34.

BLAGOJEVIC, B. *How to (actually) easily detect objects with deep learning*. [S.l.], 2018. Disponível em: <https://medium.com/ml-everything/how-to-actually-easily-detect-objects-with-deep-learning-on-raspberry-pi-4fd40af84fee>. Acesso em: 05 abr. 2021. Citado na página 41.

BOCHKOVSKIY, A.; WANG, C.-Y.; LIAO, H.-Y. M. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. Citado na página 25.

BRADSKI, A. *Learning OpenCV, [Computer Vision with OpenCV Library ; software that sees]*. 1. ed.. ed. [S.l.]: O'Reilly Media, 2008. Gary Bradski and Adrian Kaehler. ISBN 0-596-51613-4. Citado na página 17.

BRADSKI, G. The opencv library. *Dr. Dobb's Journal of Software Tools*, v. 25, 01 2000. Citado na página 26.

BRADSKI, G.; KAEHLER, A. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008. ISBN 9780596554040. Disponível em: <https://books.google.com.br/books?id=seAgiOfu2EIC>. Citado na página 25.

BUCK, I. et al. Brook for gpus: Stream computing on graphics hardware. *ACM Trans. Graph.*, v. 23, p. 777–786, 08 2004. Citado na página 28.

CATANZARO, B.; SUNDARAM, N.; KEUTZER, K. Fast support vector machine training and classification on graphics processors. In: ACM, New York, NY, USA. [S.l.], 2008. Citado nas páginas 15 e 16.

CHO, J. et al. Fpga-based face detection system using haar classifiers. In: . [S.l.: s.n.], 2009. p. 103–112. Citado na página 32.

DUDA, R.; HART, P.; D.G. Pattern classification. In: *Wiley-Interscience, 2 edition*. [S.l.: s.n.], 2000. Citado na página 14.

FILHO, O. M.; NETO, H. V. *Processamento Digital de Imagens*. [S.l.: s.n.], 1999. v. 21/03. ISSN 0102261X. Citado nas páginas 18, 19, 20 e 38.

GONZALEZ, R. C.; WOODS, R. E. *Digital image processing*. Upper Saddle River, N.J.: Prentice Hall, 2008. ISBN 9780131687288 013168728X 9780135052679 013505267X. Disponível em: <http://www.amazon.com/Digital-Image-Processing-3rd-Edition/dp/013168728X>. Citado nas páginas 17, 18, 20, 38 e 39.

GOOGLE. *TensorFlow*. 2021. Disponível em: <https://www.tensorflow.org/>. Citado na página 15.

GRACIANO, A. B. V. *Rastreamento de objetos baseado em reconhecimento estrutural de padrões*. 2007. 138 p. Dissertação (Dissertação (Mestrado em Ciências)) — Instituto de Matemática e Estatística, Universidade de São Paulo, 2007. Citado na página 14.

GROUPS, K. *OpenGL*. 2021. Disponível em: <https://www.opengl.org//>. Citado na página 29.

HARVEY, J. P. Gpu acceleration of object classification algorithms using nvidia cuda. 2009. Citado na página 14.

HARVEY, J. P. *GPU acceleration of object classification algorithms using NVIDIA CUDA*. 2009. 94 p. Dissertação (Master of Science in Computer Engineering) — Rochester Institute of Technology Kate Gleason College of Engineering, 2009. Citado nas páginas 23, 32 e 33.

HOLLERWEGER, M. M. Aplicacao de visao computacional no auxilio ao levantamento de defeitos em pavimento rodoviariomostrar registro completo. *Universidade Federal de Santa Catarina*, 2019. Citado na página 26.

INTEL; GARAGE, W.; ITSEEZ. *OpenCV*. 2000. Disponível em: <https://opencv.org/>. Citado nas páginas 40 e 44.

JANG, H.; PARK, A.; JUNG, K. Neural network implementation using cuda and openmp. In: IEEE COMPUTER SOCIETY, Washington, DC, USA. [S.l.], 2008. Citado na página 15.

JETBRAINS. *The Python IDE for Professional Developers*. 2010. Disponível em: <https://www.jetbrains.com/pycharm/>. Citado na página 44.

JUNIOR, J. C. S. J. *Introdução do Processamento de Imagens. Julio C. S. Jacques Junior*. 2015. Disponível em: <https://docplayer.com.br/5760888-Introducao-do-processamento-de-imagens-julio-c-s-jacques-junior-juliojj-gmail-com.html>. Citado na página 22.

KITWARE. *CMake*. 2000. Disponível em: <https://cmake.org/>. Citado na página 44.

LeCun, Y.; Fu Jie Huang; Bottou, L. Learning methods for generic object recognition with invariance to pose and lighting. In: *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.* [S.l.: s.n.], 2004. v. 2, p. II–104 Vol.2. Citado na página 15.

LIENHART, R.; MAYDT, J. An extended set of haar-like features for rapid object detection. In: . [S.l.: s.n.], 2002. v. 1, p. I–900. Citado na página 31.

LIN, T.-Y. et al. *Microsoft COCO: Common Objects in Context*. 2015. Citado nas páginas 26 e 27.

NVIDIA. *CUDA Toolkit*. 2007. Disponível em: <https://developer.nvidia.com/cuda-toolkit>. Citado na página 44.

NVIDIA. *Fermi computer architecture whitepaper*. 2009. Disponível em: <https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Citado na página 28.

NVIDIA. *NVIDIA cuDNN*. 2014. Disponível em: <https://developer.nvidia.com/cudnn>. Citado na página 44.

NVIDIA. *NVIDIA CUDA Zone*. [S.l.], 2015. Disponível em: <http://developer.nvidia.com/category/zone/cuda-zone>. Acesso em: 13 out. 2020. Citado nas páginas 14 e 29.

NVIDIA. *NVIDIA OpenCL*. 2021. Disponível em: <https://developer.nvidia.com/opencl>. Citado nas páginas 15 e 29.

NVIDIA. *OpenACC*. 2021. Disponível em: <https://www.openacc.org/>. Citado na página 15.

OH, K.-S.; JUNG, K. Gpu implementation of neural networks. *Pattern Recognition*, v. 37, n. 6, p. 1311–1314, 2004. ISSN 0031-3203. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0031320304000524>. Citado na página 15.

OLIVEIRA, A. B. *Filtro de partículas adaptativo para o tratamento de oclusões no rastreamento de objetos em vídeos*. 2008. 58 p. Dissertação (Dissertação (Mestrado em Computação)) — Universidade Federal do Rio Grande do Sul, 2008. Citado na página 14.

OPENCV. *OpenCV Contrib*. 2014. Disponível em: <https://github.com/opencv/opencv_contrib>. Citado na página 44.

OPENCV. *Template Matching*. 2020. Disponível em: <https://docs.opencv.org/master/d4/dc6/tutorial_py_template_matching.html>. Citado nas páginas 15 e 24.

ORACLE. *BLOB data type*. [S.l.], 1977. Disponível em: <https://docs.oracle.com/javadb/10.8.3.0/ref/rrefblob.html>. Acesso em: 05 abr. 2021. Citado na página 39.

PORTES, M. Processamento de imagens : Métodos e análises. *Revista Científica*, 2000. ISSN 15198022. Citado na página 19.

SATHYA, R. *Comparison of Supervised and Unsupervised Learning Algorithms for Pattern Classification*. 2013. Citado na página 20.

SHRESTHA, B.; KANSAKAR, L. Real time face tracking and recognition (rtftr). In: . [S.l.: s.n.], 2009. Citado na página 32.

Sun, L. et al. Acceleration algorithm for cuda-based face detection. In: *2013 IEEE International Conference on Signal Processing, Communication and Computing (ICSPCC 2013)*. [S.l.: s.n.], 2013. p. 1–5. Citado na página 34.

SZELISKI, R. *Computer vision algorithms and applications*. London; New York: Springer, 2011. Disponível em: <http://dx.doi.org/10.1007/978-1-84882-935-0>. Citado na página 17.

Theocharides, T.; Vijaykrishnan, N.; Irwin, M. J. A parallel architecture for hardware face detection. In: *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*. [S.l.: s.n.], 2006. p. 2 pp.–. Citado na página 32.

UNSPLASH. *Unsplash*. [S.l.], 2019. Disponível em: <https://unsplash.com/>. Acesso em: 05 abr. 2021. Citado na página 45.

USP. *Template Matching*. 2017. Disponível em: <http://www.lps.usp.br/hae/apostilaraspi/matching.pdf>. Citado nas páginas 23 e 24.

VIDEVO. *Videvo*. [S.l.], 2015. Disponível em: <https://www.videvo.net/>. Acesso em: 05 abr. 2021. Citado na página 48.

VIOLA, P.; JONES, M. Robust real-time object detection. In: *International Journal of Computer Vision*. [S.l.: s.n.], 2001. Citado nas páginas 14, 22 e 31.

Wai, A. W. Y.; Tahir, S. M.; Chang, Y. C. Gpu acceleration of real time viola-jones face detection. In: *2015 IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*. [S.l.: s.n.], 2015. p. 183–188. Citado nas páginas 14, 15, 16 e 31.

ZANOTTO, L.; FERREIRA, A.; MATSUMOTO, M. *Arquitetura e Programação de GPU Nvidia*. 2012. 7 p. Citado na página 27.

ZHAO, Z.-Q. et al. *Object Detection with Deep Learning: A Review*. 2019. Citado na página 38.