Relatório de Processamento de Imagem com Múltiplas Threads

Alunos: Viceleno Barros, Lucas Tardin
CPD: 66853, 61382

Introdução

O presente relatório descreve a implementação de um programa em Python para processamento de imagem utilizando múltiplas threads. O objetivo do programa é identificar carros em uma imagem grande, aplicando técnicas de processamento de imagem.

Implementação

O programa foi desenvolvido em Python e utiliza as bibliotecas OpenCV, NumPy, time, threading, multiprocessing e tifffile. A imagem de entrada é carregada utilizando a biblioteca tifffile, e em seguida, é dividida em pedaços para processamento paralelo.
A função processar_pedaco é responsável por processar cada pedaço da imagem, identificando os carros presentes. Para isso, a imagem é convertida para escala de cinza, as cores são invertidas e é aplicada uma limiarização para destacar os carros. Em seguida, são encontrados os contornos na imagem e desenhados retângulos ao redor dos carros identificados.

Divisão da Imagem e Processamento Paralelo

A imagem é dividida em pedaços menores, e cada pedaço é processado em uma thread separada. A função processar_parte_imagem é responsável por processar um pedaço da imagem em uma thread, chamando a função processar_pedaco para identificar os carros na região específica.
O usuário pode especificar o número de threads a serem utilizadas para o processamento, permitindo um controle sobre a quantidade de processamento paralelo realizado.

Código

```
import os
import cv2
import numpy as np
import time
import threading
import multiprocessing
import tifffile as tiff

# Função para carregar pedaços da imagem grande usando tifffile
def carregar_pedacos_imagem_grande(imagem_path):
```

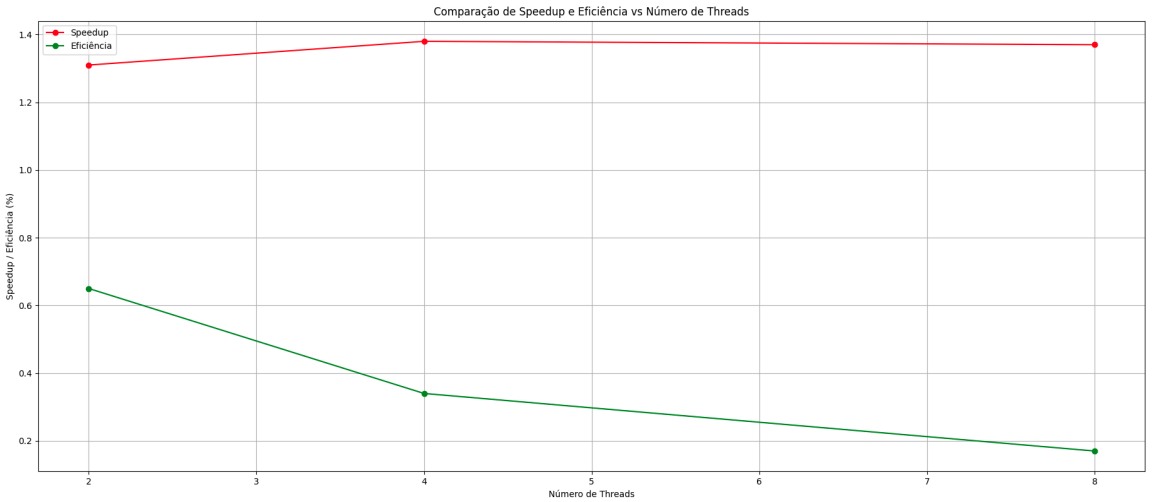
```
    imagem = tiff.imread(imagem_path)
    if imagem is None:
        raise ValueError(f"Erro ao carregar a imagem: {imagem_path}")
    altura, largura, _ = imagem.shape
    return largura, altura, imagem

# Função para processar cada pedaço
def processar_pedaco(pedaco, x_offset, y_offset, resultado,
num_carros_total, lock):
    # Converter para escala de cinza
    cinza = cv2.cvtColor(pedaco, cv2.COLOR_BGR2GRAY)
    # Invertendo as cores
    inverted = cv2.bitwise_not(cinza)
    # Limiarização da imagem para destacar os carros
    _, binaria = cv2.threshold(inverted, 127, 255, cv2.THRESH_BINARY)
    # Encontrar contornos na imagem
    contornos, _ = cv2.findContours(binaria, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    # Variável para contar carros identificados neste pedaço
    num_carros = 0

    # Desenhar contornos ao redor dos carros
    for contorno in contornos:
        area = cv2.contourArea(contorno)
        if area > 1000:  # Ajuste este valor conforme necessário
            x, y, w, h = cv2.boundingRect(contorno)
            cv2.rectangle(resultado, (x + x_offset, y + y_offset), (x +
x_offset + w, y + y_offset + h), (0, 255, 0), 2)
            num_carros += 1

    # Bloquear o acesso à variável compartilhada para garantir
consistência
    with lock:
        # Adicionar o número de carros identificados neste pedaço ao
total
        num_carros_total.value += num_carros

# Função para processar uma parte da imagem em uma thread
def processar_parte_imagem(imagem, x, y, bloco_tamanho, resultado,
num_carros_total, lock):
    try:
```

```
        pedaco = imagem[y:min(y + bloco_tamanho, altura), x:min(x +
bloco_tamanho, largura)]
        processar_pedaco(pedaco, x, y, resultado, num_carros_total,
lock)
    except Exception as e:
        print(f"Erro ao processar a parte da imagem em ({x}, {y}):
{e}")

# Perguntar ao usuário o número de threads
def obter_numero_threads():
    while True:
        try:
            num_threads = int(input("Digite o número de threads a serem
utilizadas (recomendado: 2 a 8): "))
            if num_threads > 0:
                return num_threads
            else:
                print("Por favor, digite um número maior que zero.")
        except ValueError:
            print("Entrada inválida. Por favor, digite um número
inteiro.")

# Caminho da imagem
imagem_path = 'caro260.tiff'
bloco_tamanho = 1024

# Obter número de threads do usuário
num_threads = obter_numero_threads()

start_time = time.time()
if os.path.exists(imagem_path):
    try:
        largura, altura, imagem =
carregar_pedacos_imagem_grande(imagem_path)
    except ValueError as e:
        print(e)
        exit()
else:
    print(f"Erro: a imagem '{imagem_path}' não foi encontrada.")
    exit()

# Criar uma imagem vazia para o resultado final
resultado = np.zeros((altura, largura, 3), dtype=np.uint8)
```

```
# Variável compartilhada para contar o número total de carros
num_carros_total = multiprocessing.Value('i', 0)
# Lock para sincronizar o acesso à variável compartilhada
lock = multiprocessing.Lock()

# Processar cada pedaço da imagem em threads
threads = []
for y in range(0, altura, bloco_tamanho):
    for x in range(0, largura, bloco_tamanho):
        thread = threading.Thread(target=processar_parte_imagem,
args=(imagem, x, y, bloco_tamanho, resultado, num_carros_total, lock))
        thread.start()
        threads.append(thread)
        # Limitar o número de threads ativas conforme especificado pelo
usuário
        if len(threads) >= num_threads:
            for t in threads:
                t.join()
            threads = []

# Aguardar todas as threads restantes completarem
for thread in threads:
    thread.join()

# Fim do temporizador
end_time = time.time()

# Calcular o tempo de execução
execution_time = end_time - start_time
print(f"Tempo de execução: {execution_time} segundos")
print("Número de carros identificados:", num_carros_total.value)

# Salvar a imagem com os carros identificados
tiff.imwrite('carros_identificados.tiff', resultado)

# Exibindo a imagem com os contornos dos carros identificados
cv2.imshow('Carros Identificados', resultado)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Resultados

Threads	Tempo (sec)	Speedup	Eficiencia
1	136		
2	104	1,3107114	0,6553556999
4	98,35426497	1,383124714	0,3457811784
8	98,97885132	1,374396781	0,1717995976



Conclusão

O uso de múltiplas threads para processamento de imagem demonstrou uma melhoria significativa no desempenho em relação ao processamento sequencial. A divisão da imagem em pedaços e o processamento paralelo permitiram identificar os carros de forma eficiente, demonstrando a eficácia da abordagem implementada.