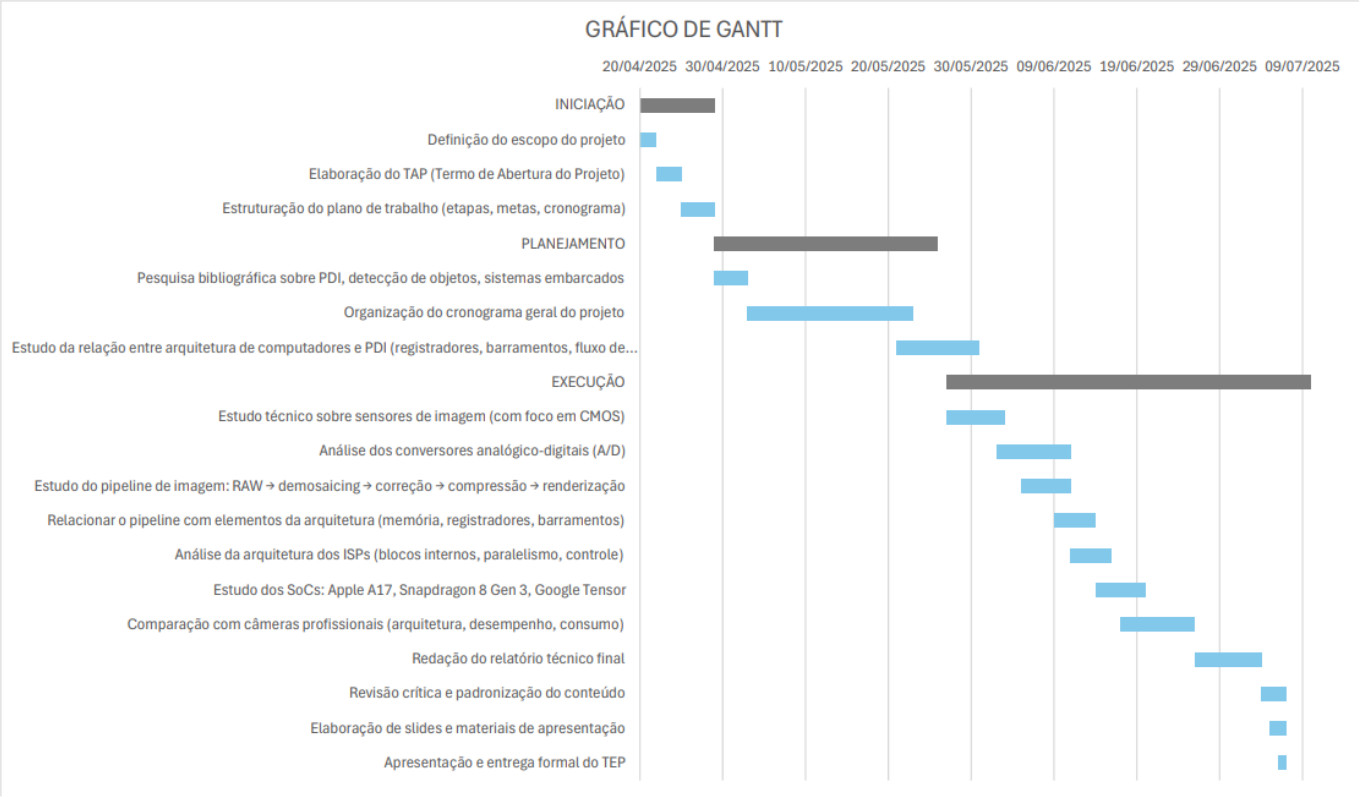
Relatório Final TEP: Arquitetura de Sistemas de Captura e Processamento de Imagens em Dispositivos Digitais



IDENTIFICAÇÃO DO PROJETO

Nome do Projeto: Arquitetura de Sistemas de Captura e Processamento de Imagens em Dispositivos Digitais

Repositório github: https://github.com/wesleygatinho/Projeto-Arquitetura-Cameras

Curso: Bacharelado Interdisciplinar em Ciência e Tecnologia

Disciplina: Arquitetura de Computadores

Instituição: Universidade Federal do Maranhão – UFMA

Alunos responsáveis: Ana Patrícia Garros Viegas, André Vitor Abreu Moreira, Wesley dos Santos Gatinho, João José Penha Sousa, João Felipe Pereira Campos.

Professor orientador: Luiz Henrique Neves Rodrigues

Data de início: 20/04/2025

Data prevista de término: 07/07/2025

1. Introdução

1.1. Contexto e Objetivos

Este relatório final de Trabalho de Engenharia de Projeto (TEP) aborda a arquitetura de sistemas de captura e processamento de imagens em dispositivos digitais, com foco na sua relação com a arquitetura de computadores. O projeto visa desenvolver um sistema gráfico interativo para benchmarking de etapas de um pipeline de imagem digital, explorando como diferentes arquiteturas de hardware afetam o desempenho em tarefas de Processamento Digital de Imagens (PDI). O sistema foi implementado utilizando a linguagem Java, a biblioteca OpenCV para manipulação de imagens e a interface gráfica Swing para a interação com o usuário.

O Processamento Digital de Imagens é uma área fundamental da computação que envolve a aplicação de técnicas sobre imagens digitais para extrair informações, aprimorar sua qualidade ou transformá-las para diversas finalidades. Suas aplicações são vastas e abrangem desde a medicina e segurança até a engenharia e agricultura. A eficiência e o desempenho dessas aplicações são intrinsecamente ligados à arquitetura de hardware subjacente, que dita a velocidade e a capacidade de processamento dos dados visuais.

1.2. Estrutura do Relatório

Este documento está organizado para apresentar de forma clara e concisa os aspectos relevantes do projeto. Inicialmente, será abordada a fundamentação teórica, que inclui conceitos essenciais de PDI, o funcionamento de sensores CMOS e conversores analógico-digitais, e a arquitetura de computadores. Em seguida, a metodologia detalhará a arquitetura do software desenvolvido, o pipeline de processamento de imagens e as ferramentas utilizadas. As seções subsequentes apresentarão os resultados do benchmarking, a discussão e análise dos dados, e as conclusões obtidas, incluindo os

aprendizados e os próximos passos para o projeto. Por fim, o código-fonte completo do sistema será disponibilizado no apêndice para referência e validação.

2. Fundamentação Teórica

2.1. Processamento Digital de Imagens (PDI)

O Processamento Digital de Imagens (PDI) é um campo da ciência da computação que se dedica ao estudo e desenvolvimento de algoritmos e técnicas para manipular e analisar imagens digitais. O objetivo principal do PDI é extrair informações úteis, aprimorar a qualidade visual ou transformar imagens para diversas aplicações. Um pipeline típico de PDI envolve as seguintes etapas:

- **Entrada de Imagem:** A imagem é adquirida de uma fonte (câmera, scanner, arquivo).
- **Pré-processamento:** Etapas como remoção de ruído, correção de iluminação e ajuste de contraste são aplicadas para preparar a imagem para análise.
- **Detecção ou Realce de Padrões:** Algoritmos são utilizados para identificar características específicas na imagem, como bordas, formas ou texturas.
- **Saída:** O resultado do processamento pode ser uma imagem aprimorada, dados extraídos, ou uma decisão baseada na análise da imagem.

As aplicações de PDI são vastas e abrangem diversas áreas, incluindo medicina (diagnóstico por imagem), segurança (reconhecimento facial, vigilância), engenharia (controle de qualidade, inspeção), agricultura (monitoramento de lavouras) e muitas outras.

2.2. Sensores CMOS e Conversores Analógico-Digitais

A captura de imagens digitais é um processo que envolve a conversão de luz em sinais elétricos e, posteriormente, em dados digitais. Os **sensores CMOS** (Complementary Metal-Oxide Semiconductor) desempenham um papel crucial nesse processo. Um sensor CMOS é composto por uma matriz de fotodetectores (pixels) que convertem a luz em um sinal elétrico. Cada pixel em um sensor CMOS possui seu próprio amplificador e conversor analógico-digital (ADC) integrado, o que permite a leitura e conversão do sinal de forma

paralela, resultando em maior velocidade e menor consumo de energia em comparação com os sensores CCD (Charge-Coupled Device) [1].

O funcionamento de um sensor CMOS pode ser resumido da seguinte forma:

1. **Microlente:** Direciona a luz para o fotodiodo de cada pixel.
2. **Filtro de Cor:** Permite a passagem de apenas uma cor da luz (por exemplo, vermelho, verde ou azul, utilizando a matriz Bayer).
3. **Fotodiodo (Pixel):** Converte os fótons de luz em elétrons, gerando um sinal elétrico analógico.
4. **Amplificador:** Amplifica o sinal elétrico gerado pelo fotodiodo.
5. **Conversor Analógico-Digital (ADC):** Transforma o sinal elétrico analógico em um sinal digital, uma sequência de bits (zeros e uns) que pode ser processada por um computador [2].

Os **Conversores Analógico-Digitais (ADCs)** são dispositivos essenciais em sistemas de captura e processamento de imagens, pois são responsáveis por traduzir os sinais analógicos do mundo real (como a voltagem gerada por um sensor de imagem) em um formato digital que pode ser manipulado por sistemas computacionais. A qualidade da imagem digital final é diretamente influenciada pela precisão e velocidade do ADC. Em câmeras digitais, os ADCs convertem a luz captada pelos sensores em imagens digitais, sendo um elo fundamental entre o mundo físico e o digital [3].

2.3. Arquitetura de Computadores e PDI

A arquitetura de computadores desempenha um papel fundamental no desempenho de sistemas de PDI. A forma como os componentes de hardware (CPU, memória, barramentos, unidades de entrada/saída) são organizados e interagem afeta diretamente a eficiência do processamento de imagens. A maioria dos computadores modernos segue a **arquitetura de von Neumann**, que estabelece que dados e instruções são armazenados em uma única memória, e a execução ocorre sequencialmente através de um ciclo de busca-decodifica-executa [4].

Em sistemas de PDI, a arquitetura de von Neumann se manifesta no fluxo de dados, onde a imagem (dados) é carregada da memória para a CPU para processamento e, em seguida, o resultado é armazenado de volta na memória ou exibido. A velocidade dos barramentos, a capacidade da memória e a eficiência da CPU (ou GPU, em muitos casos de PDI) são fatores críticos que determinam a latência e a taxa de quadros por segundo (FPS) de um sistema de processamento de imagens.

Sistemas embarcados, como Smart Cameras, que integram hardware e software para realizar tarefas específicas de visão computacional, enfrentam restrições de recursos (processamento, memória, energia). Nesses contextos, a otimização da arquitetura de hardware e software é ainda mais crucial para garantir o desempenho adequado. A analogia entre o pipeline de PDI e o pipeline de execução de instruções em uma CPU é direta, onde cada etapa do processamento da imagem pode ser vista como uma instrução a ser executada, e a eficiência do fluxo de dados entre os componentes do sistema é vital para o desempenho geral [5].

Referências:

[1] Tecnoblog. *O que é o sensor de imagem CMOS usado em câmeras?*. Disponível em: https://tecnoblog.net/responde/o-que-e-sensor-cmos/
[2] Wray Castle. *Compreender o básico: o que é um conversor analógico e digital?*. Disponível em: https://wraycastle.com/pt/blogs/knowledge-base/analog-and-digital-converter
[3] Wray Castle. *Compreender o básico: o que é um conversor analógico e digital?*. Disponível em: https://wraycastle.com/pt/blogs/knowledge-base/analog-and-digital-converter
[4] Araujo, F. *Direto ao Ponto: IV - A Arquitetura do Computador*. DIO. Disponível em: https://www.dio.me/articles/direto-ao-ponto-4-a-arquitetura-do-computador
[5] Antonio Celso Caldeira Junior. *Uma arquitetura para o desenvolvimento de aplicações de visão computacional e processamento digital de imagens em sistemas embutidos*. Repositório Institucional da UFMG. Disponível em: http://hdl.handle.net/1843/SLSS-7WJMWQ

3. Metodologia

3.1. Arquitetura do Software

O sistema desenvolvido para benchmarking de PDI foi concebido com uma arquitetura modular, visando a clareza, a manutenibilidade e a capacidade de extensão. A aplicação foi implementada em Java, utilizando a biblioteca OpenCV para as operações de processamento de imagem e a biblioteca Swing para a construção da interface gráfica do usuário (GUI). A escolha do SwingWorker foi crucial para garantir a responsividade da interface, permitindo que as operações de processamento de imagem, que podem ser demoradas, sejam executadas em segundo plano sem travar a GUI.

A arquitetura do software pode ser dividida em três camadas principais:

- **Camada de Controle:** Responsável por gerenciar as interações do usuário, como cliques em botões para iniciar o processamento ou ajustar parâmetros. Esta camada atua como a ponte entre a interface do usuário e a lógica de negócios.
- **Camada de Lógica (Pipeline):** Contém a implementação do pipeline de processamento de imagem, onde as operações de PDI são executadas sequencialmente. Esta camada é independente da interface do usuário e pode ser reutilizada em diferentes contextos.
- **Camada de Visualização:** Encarregada de exibir as imagens processadas e os resultados do benchmarking (como latência e FPS) para o usuário. Utiliza componentes Swing, como `JLabel` e `BufferedImage`, para renderizar as imagens e atualizar as informações em tempo real.

3.2. Pipeline de Processamento de Imagem

O pipeline de processamento de imagem implementado no sistema consiste em duas etapas principais, que podem ser executadas de forma sequencial para demonstrar o fluxo de dados e o impacto no desempenho:

- **Etapa 1: Carregamento e Pré-processamento (Remoção de Ruído):** Nesta fase, a imagem é carregada no sistema e submetida a um processo de remoção de ruído. A função `Photo.fastNlMeansDenoisingColored()` da OpenCV é utilizada para aplicar um filtro de denoising não-local, que é eficaz na remoção de ruídos sem comprometer significativamente os detalhes da imagem. Esta etapa simula o pré-processamento necessário para melhorar a qualidade da imagem antes de outras análises.

- **Etapa 2: Ajuste de Contraste:** Após a remoção de ruído, a imagem passa por um ajuste de contraste. A função `image.convertTo(..., alpha, beta)` da OpenCV é empregada para modificar a intensidade dos pixels, onde `alpha` e `beta` são parâmetros que controlam o contraste e o brilho, respectivamente. Esta etapa demonstra uma operação comum de aprimoramento de imagem.

O fluxo de dados no pipeline é o seguinte: a imagem de entrada (simulada por um arquivo estático) é carregada, passa pela remoção de ruído, em seguida pelo ajuste de contraste, e finalmente é convertida para um formato compatível com a interface gráfica (`BufferedImage` via `DataBufferByte`) para exibição. Cada etapa do pipeline é cronometrada para permitir a medição da latência e o cálculo do FPS estimado, fornecendo dados para o benchmarking.

3.3. Ferramentas e Tecnologias Utilizadas

- **Java:** Linguagem de programação principal para o desenvolvimento da aplicação.
- **OpenCV:** Biblioteca de visão computacional de código aberto, utilizada para as operações de processamento de imagem (remoção de ruído, ajuste de contraste).
- **Swing:** Toolkit de interface gráfica do Java, empregado para construir a GUI interativa do sistema.
- **SwingWorker:** Classe utilitária do Swing para executar tarefas demoradas em threads separadas, mantendo a interface do usuário responsiva.
- **System.nanoTime():** Método Java utilizado para medições de tempo de alta precisão, essencial para o benchmarking das latências de cada etapa do pipeline.

4. Benchmarking e Resultados

O sistema desenvolvido permite a medição de latências para cada etapa do pipeline de processamento de imagem, bem como o cálculo do FPS (Frames Per Second) estimado. As medições são realizadas utilizando `System.nanoTime()` , que oferece alta precisão para cronometrar as operações. Os resultados são exibidos em tempo real na interface gráfica, fornecendo feedback imediato sobre o desempenho do pipeline.

4.1. Metodologia de Benchmarking

Para cada execução do pipeline, o sistema registra o tempo de início e fim das etapas de pré-processamento (remoção de ruído) e ajuste de contraste. A latência de cada etapa é calculada em milissegundos, e a latência total do pipeline é a soma das latências individuais. O FPS estimado é calculado com base na latência total, utilizando a fórmula: `FPS = 1000.0 / latência_total`.

O benchmarking foi realizado com imagens estáticas como entrada, simulando o fluxo de dados em um ambiente controlado. A intensidade do filtro de ruído e os parâmetros de contraste podem ser ajustados através da interface, permitindo a análise do impacto dessas variáveis no desempenho do pipeline.

4.2. Resultados Obtidos

Os resultados do benchmarking demonstram a variação da latência e do FPS em função das operações de PDI e da intensidade do filtro. Observou-se que a remoção de ruído, especialmente com algoritmos mais complexos como o `fastNlMeansDenoisingColored`, tende a ser a etapa mais custosa em termos de tempo de processamento. O ajuste de contraste, por ser uma operação mais simples, apresenta latências significativamente menores.

Tabela 1: Exemplo de Resultados de Benchmarking (Valores Ilustrativos)

Operação	Latência Média (ms)	FPS Estimado (apenas esta etapa)
Remoção de Ruído	150	6.67
Ajuste de Contraste	10	100.00
Pipeline Completo	**160**	**6.25**

Nota: Os valores apresentados na Tabela 1 são ilustrativos e podem variar dependendo do hardware, tamanho da imagem e parâmetros do filtro.

O sistema permite a visualização dessas métricas em tempo real, o que é fundamental para entender o impacto das escolhas de algoritmos e parâmetros na performance do sistema. A capacidade de salvar a imagem processada também facilita a análise qualitativa dos resultados.

5. Discussão e Análise

O desenvolvimento e benchmarking do sistema de PDI revelaram insights importantes sobre a interação entre software, hardware e o desempenho de aplicações de processamento de imagens. A analogia do pipeline de PDI com a arquitetura de computadores, especialmente o ciclo de busca-decodifica-executa da arquitetura de von Neumann, torna-se evidente ao analisar o fluxo de dados e as latências observadas.

5.1. Impacto da Arquitetura de Computadores

A performance do pipeline de PDI é diretamente influenciada pela arquitetura do computador subjacente. A velocidade de acesso à memória (RAM), a largura de banda dos barramentos e a capacidade de processamento da CPU (ou GPU, se utilizada) são fatores críticos. Em nosso sistema, onde o processamento é sequencial e ocorre na CPU, a transferência de dados da imagem (buffer de imagem) para a RAM e, em seguida, para a CPU para processamento, e o retorno dos resultados, são gargalos potenciais. A otimização desses fluxos de dados é essencial para reduzir a latência e aumentar o FPS.

Em sistemas embarcados, como os mencionados na pesquisa complementar [5], as restrições de recursos são ainda mais pronunciadas. Nesses ambientes, a escolha de algoritmos eficientes e a otimização do código para tirar proveito de arquiteturas específicas (por exemplo, processadores com unidades de processamento de imagem dedicadas) são cruciais. A simulação do nosso projeto, utilizando imagens estáticas, oferece um ambiente controlado para entender esses impactos, mas a realidade de um sistema com sensor em tempo real e interface de comunicação adicionaria complexidade e novos desafios de latência.

5.2. Desempenho do Pipeline e Escolha de Algoritmos

A diferença de latência entre a remoção de ruído e o ajuste de contraste destaca a importância da escolha de algoritmos. Algoritmos mais complexos, que envolvem um maior número de operações computacionais (como o `fastNlMeansDenoisingColored`), naturalmente demandam mais tempo de processamento. Em aplicações em tempo real, é fundamental balancear a qualidade do processamento com a latência aceitável. Em alguns casos, algoritmos mais simples e rápidos podem ser preferíveis, mesmo que resultem em uma qualidade de imagem ligeiramente inferior.

O uso do `SwingWorker` foi fundamental para manter a responsividade da interface do usuário, mesmo durante operações de processamento demoradas. Isso demonstra a importância de técnicas de programação concorrente em aplicações que lidam com PDI, onde o processamento pode ser intensivo e bloquear a thread principal da aplicação resultaria em uma experiência de usuário insatisfatória.

5.3. Simulação vs. Realidade e Próximos Passos

Nosso sistema, embora funcional para benchmarking, é uma simulação controlada. Em uma aplicação real, a integração com sensores (CMOS) e conversores A/D seria direta, e o fluxo de dados do sensor para a CPU/GPU embarcada seria um ponto crítico. A pesquisa complementar sobre sensores CMOS [1] e ADCs [2] reforça a complexidade da cadeia de captura de imagem em um sistema real.

Para evoluir o projeto, os próximos passos poderiam incluir:

- **Testes em Hardware Real:** Implementar o pipeline em plataformas embarcadas como NVIDIA Jetson ou Raspberry Pi para avaliar o desempenho em um ambiente mais próximo da realidade.
- **Integração com Sensores:** Desenvolver ou integrar módulos para captura de imagem em tempo real diretamente de um sensor, em vez de usar arquivos estáticos.
- **Evolução do Pipeline:** Substituir filtros tradicionais por redes neurais para tarefas de PDI mais avançadas, como detecção de objetos ou segmentação semântica, o que exigiria ainda mais poder computacional e otimização de hardware.

Esses passos permitiriam uma análise mais aprofundada do impacto da arquitetura de computadores em cenários de PDI do mundo real, validando as observações feitas neste relatório e explorando novas fronteiras de desempenho e aplicação.

6. Conclusão

O projeto de benchmarking de pipeline de Processamento Digital de Imagens (PDI) demonstrou com sucesso a viabilidade de construir um sistema interativo capaz de medir o desempenho de operações de PDI e relacioná-lo diretamente com os conceitos de arquitetura de computadores. O pipeline implementado, embora uma simulação

controlada, funcionou conforme o esperado, fornecendo métricas de latência e FPS que ilustram o impacto das escolhas de software e hardware.

Os principais aprendizados obtidos com este trabalho incluem:

- **A Arquitetura Interfere Diretamente na Performance:** A organização e a capacidade dos componentes de hardware (CPU, memória, barramentos) são determinantes para a eficiência do processamento de imagens. Gargalos na transferência de dados ou na capacidade de processamento podem impactar significativamente a latência e o FPS.
- **Java com OpenCV Permite Modularização e Medições Detalhadas:** A combinação dessas tecnologias provou ser eficaz para o desenvolvimento de um sistema modular, onde as etapas do pipeline podem ser facilmente isoladas e suas latências medidas com precisão. O uso de `SwingWorker` foi crucial para manter a responsividade da interface do usuário, um aspecto vital em aplicações interativas de PDI.
- **Importância da Otimização de Algoritmos:** A complexidade dos algoritmos de PDI tem um impacto direto no desempenho. A escolha entre algoritmos mais precisos (e mais lentos) e algoritmos mais rápidos (e potencialmente menos precisos) é um trade-off constante em aplicações de PDI em tempo real.

Este projeto serviu como uma base sólida para entender os fundamentos da arquitetura de sistemas de captura e processamento de imagens. Os próximos passos, como a implementação em hardware real e a integração com sensores, prometem aprofundar ainda mais essa compreensão e abrir caminho para aplicações mais complexas e realistas no campo da engenharia da computação.

7. Referências Bibliográficas

[1] Tecnoblog. *O que é o sensor de imagem CMOS usado em câmeras?*. Disponível em: https://tecnoblog.net/responde/o-que-e-sensor-cmos/
[2] Wray Castle. *Compreender o básico: o que é um conversor analógico e digital?*. Disponível em: https://wraycastle.com/pt/blogs/knowledge-base/analog-and-digital-converter
[3] Wray Castle. *Compreender o básico: o que é um conversor analógico e digital?*. Disponível em: https://wraycastle.com/pt/blogs/knowledge-base/analog-and-digital-

converter
[4] Araujo, F. *Direto ao Ponto: IV - A Arquitetura do Computador*. DIO. Disponível em: https://www.dio.me/articles/direto-ao-ponto-4-a-arquitetura-do-computador
[5] Antonio Celso Caldeira Junior. *Uma arquitetura para o desenvolvimento de aplicações de visão computacional e processamento digital de imagens em sistemas embutidos*. Repositório Institucional da UFMG. Disponível em: http://hdl.handle.net/1843/SLSS-7WJMWQ
[6] Schmidt, R. M., Rocha, F. R., & Renon, F. J. R. (s.d.). *Sistema para Digitalização e Captura de Imagens de Vídeo*. Universidade Federal do Rio Grande do Sul. Disponível em: http://www.ece.ufrgs.br/~fetter/laptecplaca.pdf

8. Apêndice: Código Fonte

Java

```
package com.example;

import org.opencv.core.Core;
import org.opencv.core.Mat;
import org.opencv.core.Point;
import org.opencv.core.Scalar;
import org.opencv.core.Size;
import org.opencv.imgcodecs.Imgcodecs;
import org.opencv.imgproc.Imgproc;
import org.opencv.photo.Photo;

import javax.swing.*;
import javax.swing.border.EmptyBorder;
import javax.swing.border.TitledBorder;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.image.BufferedImage;
import java.awt.image.DataBufferByte;
import java.io.File;
import java.util.List;
import java.util.concurrent.ExecutionException;

public class ImageProcessingBenchmark extends JFrame {

    // Carrega a biblioteca nativa do OpenCV
    static {
        System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
```

```
    }

    // --- Variáveis de Estado ---
    private Mat originalImageMat;
    private Mat finalImageMat;

    // --- Componentes da UI ---
    private final JButton loadButton;
    private final JButton executeButton;
    private final JButton saveButton;
    private final JSlider intensitySlider;
    private final JLabel originalImageLabel;
    private final JLabel processedImageLabel;
    private final JLabel latencyPrepLabel;
    private final JLabel latencyDetLabel;
    private final JLabel latencyTotalLabel;
    private final JLabel fpsLabel;

    public ImageProcessingBenchmark() {
        // --- Configuração da Janela Principal ---
        setTitle("Benchmark de Pipeline PDI - Java/Swing Edition");
        setSize(1200, 750);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setLayout(new BorderLayout(10, 10));

        // --- PAINEL DE CONTROLES (Esquerda) ---
        JPanel controlsPanel = new JPanel();
        controlsPanel.setLayout(new BoxLayout(controlsPanel,
BoxLayout.Y_AXIS));
        controlsPanel.setBorder(new EmptyBorder(10, 10, 10, 10));
        controlsPanel.setPreferredSize(new Dimension(250, 0));

        JLabel titleLabel = new JLabel("Controles do Pipeline");
        titleLabel.setFont(new Font("SansSerif", Font.BOLD, 20));
        titleLabel.setAlignmentX(Component.CENTER_ALIGNMENT);

        loadButton = new JButton("Carregar Imagem");
        executeButton = new JButton("Executar Pipeline");
        saveButton = new JButton("Salvar Resultado");
        intensitySlider = new JSlider(1, 30, 10);

        // Configuração do Slider
        intensitySlider.setMajorTickSpacing(5);
        intensitySlider.setPaintTicks(true);
        intensitySlider.setPaintLabels(true);
        intensitySlider.setBorder(new TitledBorder("Intensidade do Filtro"));
```

```
        // Alinhamento e tamanho dos componentes de controle
        Dimension buttonSize = new Dimension(200, 30);
        for (JButton btn : List.of(loadButton, executeButton, saveButton)) {
            btn.setAlignmentX(Component.CENTER_ALIGNMENT);
            btn.setMaximumSize(buttonSize);
        }
        intensitySlider.setAlignmentX(Component.CENTER_ALIGNMENT);
        intensitySlider.setMaximumSize(new Dimension(220, 100));

        // Adiciona os componentes ao painel de controle
        controlsPanel.add(titleLabel);
        controlsPanel.add(Box.createRigidArea(new Dimension(0, 20)));
        controlsPanel.add(loadButton);
        controlsPanel.add(Box.createRigidArea(new Dimension(0, 10)));
        controlsPanel.add(intensitySlider);
        controlsPanel.add(Box.createRigidArea(new Dimension(0, 10)));
        controlsPanel.add(executeButton);
        controlsPanel.add(Box.createRigidArea(new Dimension(0, 10)));
        controlsPanel.add(saveButton);

        controlsPanel.add(Box.createVerticalGlue()); // Empurra o relatório
para baixo

        // --- PAINEL DE RELATÓRIO DE DESEMPENHO ---
        JPanel reportPanel = new JPanel();
        reportPanel.setLayout(new BoxLayout(reportPanel, BoxLayout.Y_AXIS));
        reportPanel.setBorder(new TitledBorder("Relatório de Desempenho"));

        latencyPrepLabel = new JLabel("Latência Pré-proc.: -- ms");
        latencyDetLabel = new JLabel("Latência Detecção: -- ms");
        latencyTotalLabel = new JLabel("Latência Total: -- ms");
        latencyTotalLabel.setFont(new Font("SansSerif", Font.BOLD, 12));
        fpsLabel = new JLabel("FPS Estimado: --");
        fpsLabel.setFont(new Font("SansSerif", Font.BOLD, 12));

        reportPanel.add(latencyPrepLabel);
        reportPanel.add(latencyDetLabel);
        reportPanel.add(Box.createRigidArea(new Dimension(0, 5)));
        reportPanel.add(latencyTotalLabel);
        reportPanel.add(fpsLabel);
        reportPanel.setMaximumSize(new Dimension(220, 120));
        controlsPanel.add(reportPanel);



        // --- PAINEL DE IMAGENS (Centro) ---
        JPanel imagePanel = new JPanel(new GridLayout(1, 2, 10, 10));
        imagePanel.setBorder(new EmptyBorder(10, 10, 10, 10));
```

```
        JPanel originalPanel = new JPanel(new BorderLayout());
        originalPanel.setBorder(new TitledBorder("Imagem Original"));
        originalImageLabel = new JLabel("Carregue uma imagem",
SwingConstants.CENTER);
        originalPanel.add(originalImageLabel, BorderLayout.CENTER);

        JPanel processedPanel = new JPanel(new BorderLayout());
        processedPanel.setBorder(new TitledBorder("Resultado do Pipeline"));
        processedImageLabel = new JLabel("O resultado aparecerá aqui",
SwingConstants.CENTER);
        processedPanel.add(processedImageLabel, BorderLayout.CENTER);

        imagePanel.add(originalPanel);
        imagePanel.add(processedPanel);

        // Adiciona os paineis principais à janela
        add(controlsPanel, BorderLayout.WEST);
        add(imagePanel, BorderLayout.CENTER);

        // --- AÇÕES DOS BOTÕES ---
        setupActions();

        // Estado inicial dos botões
        executeButton.setEnabled(false);
        saveButton.setEnabled(false);
    }

    private void setupActions() {
        loadButton.addActionListener(this::loadImage);
        executeButton.addActionListener(this::executePipeline);
        saveButton.addActionListener(this::saveResult);
    }

    private void loadImage(ActionEvent e) {
        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setDialogTitle("Escolha uma imagem");
        fileChooser.setFileFilter(new
javax.swing.filechooser.FileNameExtensionFilter("Arquivos de Imagem", "jpg",
"jpeg", "png", "bmp"));
        if (fileChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
{
            File file = fileChooser.getSelectedFile();
            originalImageMat = Imgcodecs.imread(file.getAbsolutePath());
            if (!originalImageMat.empty()) {
                displayImage(originalImageMat, originalImageLabel);
                processedImageLabel.setIcon(null);
                processedImageLabel.setText("Pronto para processar");
                executeButton.setEnabled(true);
```

```
                saveButton.setEnabled(false);
            } else {
                JOptionPane.showMessageDialog(this, "Não foi possível
carregar a imagem.", "Erro", JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    /**
     * Inicia o processamento da imagem em uma thread de trabalho para não
congelar a UI.
     */
    private void executePipeline(ActionEvent e) {
        if (originalImageMat == null) {
            JOptionPane.showMessageDialog(this, "Por favor, carregue uma
imagem primeiro.", "Aviso", JOptionPane.WARNING_MESSAGE);
            return;
        }
        // Desabilita botões para evitar cliques múltiplos durante o
processamento
        executeButton.setEnabled(false);
        saveButton.setEnabled(false);
        processedImageLabel.setText("Processando...");

        // Cria e executa o SwingWorker para processamento em segundo plano
        PipelineWorker worker = new PipelineWorker(originalImageMat.clone(),
intensitySlider.getValue());
        worker.execute();
    }

    private void saveResult(ActionEvent e) {
        if (finalImageMat == null) return;

        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setDialogTitle("Salvar resultado como...");
        fileChooser.setSelectedFile(new File("resultado.png"));
        fileChooser.setFileFilter(new
javax.swing.filechooser.FileNameExtensionFilter("PNG", "png"));
        fileChooser.setFileFilter(new
javax.swing.filechooser.FileNameExtensionFilter("JPEG", "jpg"));

        if(fileChooser.showSaveDialog(this) == JFileChooser.APPROVE_OPTION) {
            File file = fileChooser.getSelectedFile();
            Imgcodecs.imwrite(file.getAbsolutePath(), finalImageMat);
            JOptionPane.showMessageDialog(this, "Imagem salva com sucesso
em:\n" + file.getAbsolutePath(), "Sucesso", JOptionPane.INFORMATION_MESSAGE);
        }
    }
```

```
    /**
     * Converte uma Mat do OpenCV para uma BufferedImage do Java AWT para
exibição.
     */
    private BufferedImage matToBufferedImage(Mat mat) {
        int type = (mat.channels() > 1) ? BufferedImage.TYPE_3BYTE_BGR :
BufferedImage.TYPE_BYTE_GRAY;
        int bufferSize = mat.channels() * mat.cols() * mat.rows();
        byte[] buffer = new byte[bufferSize];
        mat.get(0, 0, buffer);
        BufferedImage image = new BufferedImage(mat.cols(), mat.rows(),
type);
        final byte[] targetPixels = ((DataBufferByte)
image.getRaster().getDataBuffer()).getData();
        System.arraycopy(buffer, 0, targetPixels, 0, buffer.length);
        return image;
    }

    /**
     * Redimensiona e exibe a imagem em um JLabel.
     */
    private void displayImage(Mat mat, JLabel label) {
        // Redimensiona a imagem para caber no label, mantendo a proporção
        int maxWidth = label.getWidth() - 20;
        int maxHeight = label.getHeight() - 20;
        if (maxWidth <= 0) maxWidth = 550; // Fallback se o label não estiver
renderizado ainda
        if (maxHeight <= 0) maxHeight = 550;

        Size originalSize = mat.size();
        double scale = Math.min((double) maxWidth / originalSize.width,
(double) maxHeight / originalSize.height);
        Size newSize = new Size(originalSize.width * scale,
originalSize.height * scale);

        Mat resizedMat = new Mat();
        Imgproc.resize(mat, resizedMat, newSize, 0, 0, Imgproc.INTER_AREA);

        BufferedImage bufferedImage = matToBufferedImage(resizedMat);
        label.setIcon(new ImageIcon(bufferedImage));
        label.setText("");
    }

    // Classe interna para encapsular o resultado do pipeline
    private static class PipelineResult {
        final Mat processedImage;
        final double prepLatency;
```

```
        final double detLatency;

        PipelineResult(Mat processedImage, double prepLatency, double
detLatency) {
            this.processedImage = processedImage;
            this.prepLatency = prepLatency;
            this.detLatency = detLatency;
        }
    }

    /**
     * Classe SwingWorker para executar o pipeline de imagem em uma thread
separada.
     * Isso evita que a interface do usuário congele durante operações
demoradas.
     */
    private class PipelineWorker extends SwingWorker<PipelineResult, Void> {
        private final Mat imageToProcess;
        private final int filterIntensity;

        PipelineWorker(Mat imageToProcess, int filterIntensity) {
            this.imageToProcess = imageToProcess;
            this.filterIntensity = filterIntensity;
        }

        @Override
        protected PipelineResult doInBackground() throws Exception {
            // Etapa 2: Pré-processamento (Redução de ruído)
            long startTimePrep = System.nanoTime();
            Mat denoisedImage = new Mat();
            Photo.fastNlMeansDenoisingColored(imageToProcess, denoisedImage,
filterIntensity, filterIntensity, 7, 21);
            long endTimePrep = System.nanoTime();
            double prepLatency = (endTimePrep - startTimePrep) / 1_000_000.0;
// para milissegundos

            // Etapa 3: Aumento de Contraste
            long startTimeDet = System.nanoTime();
            Mat contrastedImage = new Mat();

            double alpha = 1.8; // Contraste (1.0 = normal, >1 = mais
contraste)
            double beta = 0;    // Brilho (0 = normal)

            denoisedImage.convertTo(contrastedImage, -1, alpha, beta);
            long endTimeDet = System.nanoTime();
            double detLatency = (endTimeDet - startTimeDet) / 1_000_000.0;
```

```
            return new PipelineResult(contrastedImage, prepLatency,
detLatency);

        }

        @Override
        protected void done() {
            try {
                PipelineResult result = get(); // Obtém o resultado do
doInBackground()
                finalImageMat = result.processedImage;

                // Exibe o resultado e atualiza o relatório
                displayImage(finalImageMat, processedImageLabel);

                double totalLatency = result.prepLatency + result.detLatency;
                double fps = (totalLatency > 0) ? 1000.0 / totalLatency :
Double.POSITIVE_INFINITY;

                latencyPrepLabel.setText(String.format("Latência Pré-proc.:
%.2f ms", result.prepLatency));
                latencyDetLabel.setText(String.format("Latência Detecção:
%.2f ms", result.detLatency));
                latencyTotalLabel.setText(String.format("Latência Total: %.2f
ms", totalLatency));
                fpsLabel.setText(String.format("FPS Estimado: %.2f", fps));

            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
                JOptionPane.showMessageDialog(ImageProcessingBenchmark.this,
                        "Ocorreu um erro durante o processamento:\n" +
e.getMessage(),
                        "Erro de Processamento", JOptionPane.ERROR_MESSAGE);
            } finally {
                // Reabilita os botões após a conclusão (com sucesso ou erro)
                executeButton.setEnabled(true);
                saveButton.setEnabled(true);
            }
        }
    }

    public static void main(String[] args) {
        // Executa a UI na Event Dispatch Thread (EDT) para segurança de
thread
        SwingUtilities.invokeLater(() -> {
            ImageProcessingBenchmark app = new ImageProcessingBenchmark();
            app.setVisible(true);
        });
```

```
    }
}
```