Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais

Campus Ouro Branco

DOCUMENTAÇÃO DO TRABALHO PRÁTICO 2

Trabalho apresentado ao professor Saulo Cabral, da disciplina de Arquitetura e Organização de Computadores, pelos alunos Antônio Carlos, Henrique Leão, Rhuan Victor, do 3º período de Sistemas de Informação, como parte das exigências da disciplina.

Minas Gerais

Ouro Branco, 21 de junho de 2025

Introdução

Este documento detalha a implementação de um software para processamento e restauração de vídeos digitais, desenvolvido no âmbito da disciplina de Arquitetura e Organização de Computadores. O projeto foca na aplicação de técnicas de programação paralela para otimizar a correção de defeitos comuns em vídeos antigos, como ruídos e borrões, utilizando a linguagem de programação Java e a biblioteca OpenCV.

Descrição do problema a ser resolvido

O problema central deste trabalho é inspirado em uma narrativa: Seu José, um morador da cidade de Ouro Branco, encontrou uma fita VHS de valor sentimental inestimável, registrando o casamento de seus tios-avós. Contudo, a ação do tempo degradou a qualidade da gravação. O vídeo apresenta dois defeitos principais:

1. Ruído "Salt & Pepper": A imagem está repleta de ruídos que se assemelham a "chuviscos de antena mal sintonizada", com pixels pretos e brancos aleatórios poluindo a tela.
2. Borrões (Blur): Diversos quadros (frames) do vídeo estão borrados a ponto de os convidados se parecerem com "vultos atravessando a festa".

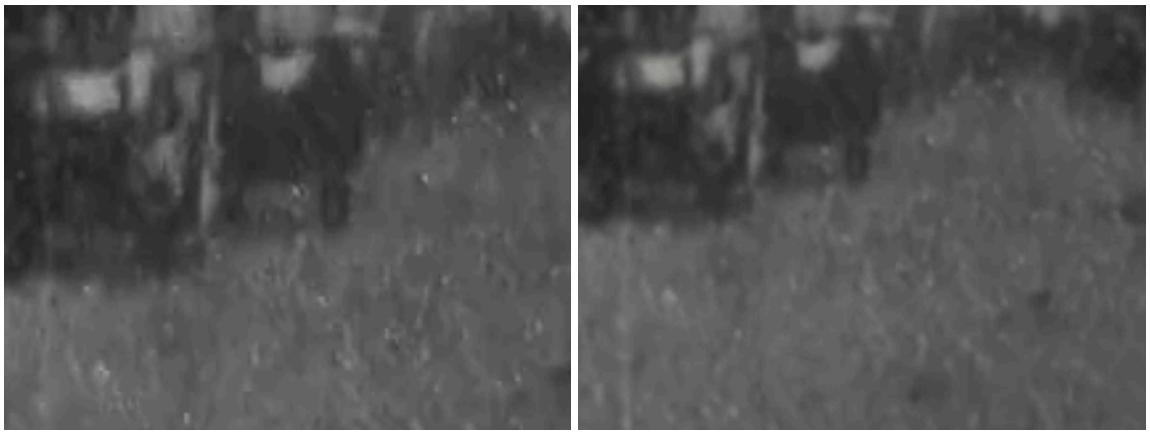
O objetivo do trabalho é, portanto, desenvolver uma solução computacional para corrigir esses defeitos. Uma restrição fundamental é a otimização do tempo de processamento, que deve ser alcançada através do uso de múltiplos núcleos de processamento (paralelismo) para acelerar a execução das tarefas de correção.

Visão geral sobre o funcionamento do programa

Funcionamento dos Filtros

A correção dos defeitos do vídeo é realizada por dois algoritmos principais:

1. **Filtro Espacial (*removerSalPimenta()*):** Um filtro de vizinhança que itera sobre cada pixel de um frame, analisando seus arredores para identificar e corrigir os ruídos pontuais característicos do "Salt & Pepper".

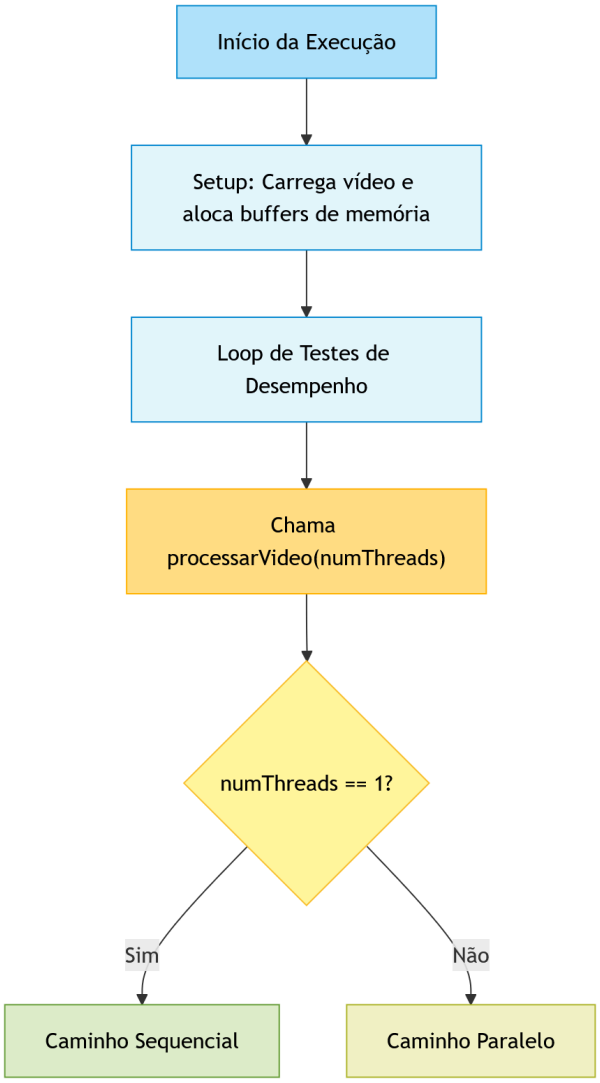


Ao analisar o mesmo frame do vídeo processado (apenas pelo algoritmo Salt & Pepper) com o original, é perceptível uma leve suavização dos “chapiscos”.

2. **Filtro Temporal (*removerBorroesTempo()*):** Um filtro que analisa cada pixel ao longo do tempo, utilizando uma janela de frames adjacentes para suavizar anomalias temporais, como os flashes de sinal, aumentando a estabilidade da imagem.

Fluxo de Execução

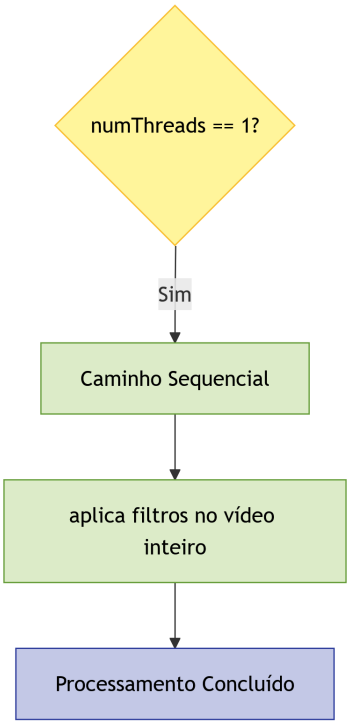
1.Início da Execução



O programa inicia no método *main*, onde define os caminhos dos arquivos e chama a função *carregarVideo*. Esta função usa a biblioteca OpenCV para ler o vídeo e armazená-lo como uma matriz 3D em memória. Em seguida, o método ***processarVideo*** é invocado. Dentro dele, um objeto **VideoDTO** é criado para encapsular a matriz do vídeo, os buffers de saída e todos os parâmetros dos filtros. Com este DTO preparado, o programa avança para a verificação que decidirá se o processamento continuará de forma sequencial ou paralela.

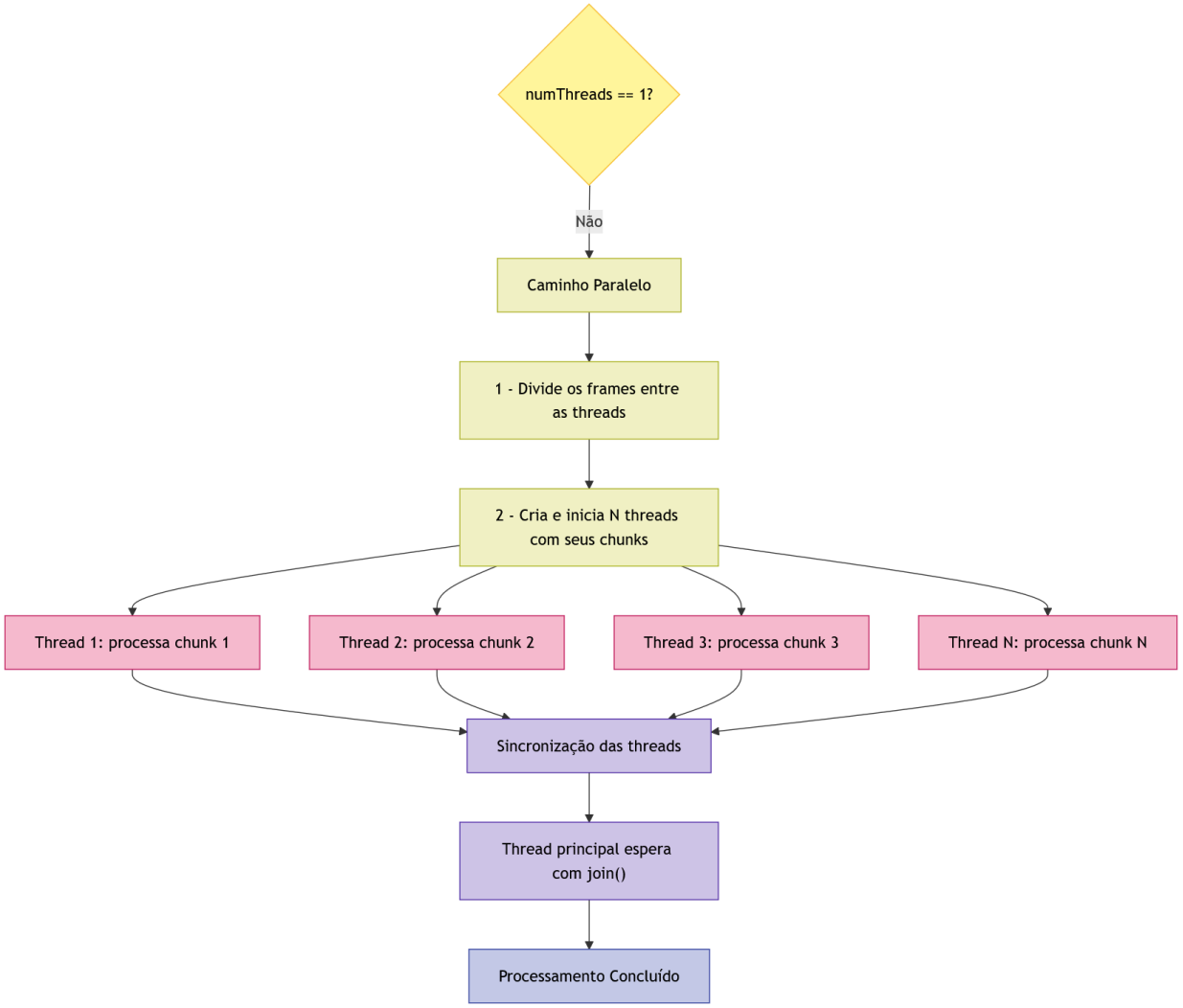
2.Tipo de Processamento

2.1. Processamento Sequencial



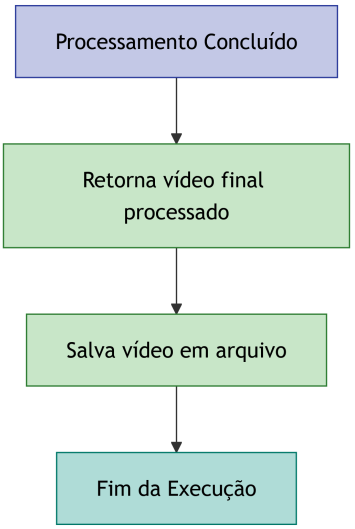
O caminho sequencial é ativado quando o programa roda configurado com apenas uma thread. Neste modo, nenhuma nova thread é criada e a thread principal assume todo o trabalho. Ela ajusta os limites de processamento para abranger o vídeo inteiro, do primeiro ao último frame, e então aplica os filtros de correção (***removerSalPimenta*** e ***removerBorroesTempo***) um após o outro em toda a sequência de frames. Essa execução serve como o benchmark de desempenho, o tempo base para comparar o ganho de velocidade obtido com o processamento paralelo.

2.2. Processamento Paralelo e Desempenho



O núcleo da otimização de desempenho reside na implementação de uma estratégia de **paralelismo de dados**. O método orquestrador (***processarVideo***) divide o número total de frames do vídeo pelo número de threads solicitado, distribuindo fatias de trabalho (chunks) para cada uma. As threads executam os algoritmos de forma concorrente em seus respectivos segmentos de vídeo. A sincronização é um ponto crítico, garantido pelo uso do método ***thread.join()***, que força a thread principal a aguardar a finalização de todas as threads trabalhadoras antes de prosseguir. Esta abordagem visa reduzir o tempo total de processamento, conforme o número de threads se aproxima do número de núcleos físicos do processador.

3.Finalização da Execução



A finalização da execução ocorre quando o processamento, seja ele sequencial ou paralelo, é concluído. O método ***processarVideo*** retorna a matriz tridimensional (*byte[][][]*) com o vídeo corrigido para o método ***main***. Neste momento, o cronômetro de desempenho é parado e o tempo total da execução é registrado no console. Em seguida, a ***main*** chama a função *gravarVideo*, que é responsável por usar a biblioteca OpenCV para codificar a matriz de dados e salvar o resultado em um novo arquivo de vídeo .mp4. Após a gravação ser bem-sucedida, o programa exibe uma mensagem final e encerra sua execução.

Implementação - Detalhamento da Estrutura do Projeto:

Classe: VideoDTO.java

A classe VideoDTO implementa o padrão de projeto *Data Transfer Object*. Sua finalidade é encapsular todos os dados e parâmetros relacionados ao processamento do vídeo em um único objeto coeso. Essa abordagem simplifica a passagem de informações entre os diferentes métodos e, principalmente, para as threads de trabalho, resultando em um código mais limpo e de fácil manutenção.

Atributos

```
// DTO (Data Transfer Object) para encapsular os dados do vídeo e os parâmetros do processamento
// Facilita passar todas as informações necessárias entre os métodos e para as threads
public class VideoDTO {  12 usages  Henrique Leão *
    // Matriz 3D com o vídeo original, que servirá como fonte de leitura
    // É 'final' para garantir que não seja acidentalmente sobrescrito
    private final byte[][][] videoPreProcessamento;  3 usages
    private byte[][][] videoPosSalPimenta; // resultado após a aplicação do primeiro filtro (Sal e Pimenta)  3 usages
    private byte[][][] videoPosCorrecaoBorroes; // resultado final, após o segundo filtro (temporal/borrões)  3 usages
    private int limiteInferior; // O índice do primeiro frame do chunk que a thread vai processar  3 usages
    private int limiteSuperior;// O índice do último frame (não inclusivo) do chunk que a thread vai processar  3 usages
    private int tamanhoLadoMascara; // Tamanho do lado da máscara (kernel) do filtro espacial, ex: 3 para uma máscara 3x3
    private TipoDeCalculo tipoDeCalculoSalPimenta; // define se o filtro de sal e pimenta usará Média ou Mediana  3 usages
    private TipoDeCalculo tipoDeCalculoBorrao; // define o cálculo para o filtro temporal  3 usages
```

- **private final byte[][][] videoPreProcessamento**: Este atributo armazena a matriz tridimensional que serve como fonte de dados para uma etapa de filtragem, representando o vídeo original. A declaração final é uma medida de segurança para garantir que a referência a este buffer de leitura não seja alterada após a inicialização do objeto, prevenindo a modificação acidental dos dados de origem.

- **private byte[][][] videoPosSalPimenta**: Este atributo é o buffer de saída para o resultado após a aplicação do primeiro filtro, que corrige o ruído "Salt & Pepper".

- **private byte[][][] videoPosCorrecaoBorroes**: Este buffer armazena o resultado do segundo filtro (temporal), representando o vídeo final corrigido dos borrões e flashes. O uso de buffers de entrada e saída separados é

fundamental para a estratégia de processamento out-of-place, que evita a corrupção de dados que ocorreria se os filtros lessem e escrevessem na mesma matriz simultaneamente.

- **private int limiteInferior; e private int limiteSuperior**: Esses atributos são cruciais para a implementação do paralelismo. Eles definem o segmento (ou "chunk") de frames pelo qual uma thread específica é responsável. limiteInferior marca o índice do primeiro frame, e limiteSuperior o índice final (não inclusivo) do segmento. Essa divisão permite que o trabalho de processar o vídeo seja distribuído entre múltiplos núcleos de forma eficiente.

- **private int tamanhoLadoMascara**: Este atributo configura o tamanho da máscara utilizada pelo filtro *Salt & Pepper*. Por exemplo, um valor de 3 corresponde a uma máscara de 3x3 pixels. Tornar este valor um parâmetro do DTO permite ajustar o *range* do filtro sem modificar o código do algoritmo.

- **private TipoDeCalculo tipoDeCalculoSalPimenta; e private TipoDeCalculo tipoDeCalculoBorrao**: Utilizando um enum (*TipoDeCalculo*), estes atributos definem qual algoritmo específico (**Média** ou **Mediana**) será aplicado em cada uma das duas etapas de filtragem. Esta abordagem desacopla a configuração da lógica de execução, permitindo que o comportamento dos filtros seja alterado de forma flexível e segura.

Construtores

```java
// Construtor principal, usado para criar o DTO inicial com o vídeo original
public VideoDTO(byte[][][] videoPreProcessamento) {  1 usage  Henrique Leão
    this.videoPreProcessamento = videoPreProcessamento;
}

// Construtor de cópia para as threads
// Herda as referências dos buffers e parâmetros do DTO principal,
// mas define limites (inferior/superior) específicos para o chunk daquela thread
public VideoDTO(VideoDTO videoModelo, int limiteInferior, int limiteSuperior) {
    this.videoPreProcessamento = videoModelo.getVideoPreProcessamento();
    this.videoPosSalPimenta = videoModelo.getVideoPosSalPimenta();
    this.videoPosCorrecaoBorroes = videoModelo.getVideoPosCorrecaoBorroes();
    this.tamanhoLadoMascara = videoModelo.getTamanhoLadoMascara();
    this.tipoDeCalculoSalPimenta = videoModelo.getTipoDeCalculoSalPimenta();
    this.tipoDeCalculoBorrao = videoModelo.getTipoDeCalculoBorrao();
    this.limiteInferior = limiteInferior;
    this.limiteSuperior = limiteSuperior;
}
```

- **public VideoDTO(byte[][][] videoPreProcessamento)**: Este é o construtor principal da classe, utilizado no início do processamento para criar o objeto DTO inicial, que contém a referência para o vídeo original a ser processado.

- **public VideoDTO(VideoDTO videoModelo, int limiteInferior, int limiteSuperior):** Este é um construtor de cópia, projetado especificamente para a criação de threads. Ele herda as referências para os buffers de vídeo e os parâmetros de filtro do DTO principal, mas recebe valores únicos para *limiteInferior* e *limiteSuperior*. Isso permite que cada thread tenha seu próprio escopo de trabalho bem definido, operando sobre os mesmos dados compartilhados.

Classe: Thread.java

Para implementar o processamento paralelo exigido no trabalho, criamos uma classe Thread customizada que estende a classe *java.lang.Thread*. Esta classe representa a nossa unidade de trabalho, ou "worker", onde cada instância dela é responsável por processar um segmento (chunk) do vídeo de forma independente.

```java
// A nossa thread "worker", a classe que vai fazer o trabalho pesado em paralelo
public class Thread extends java.lang.Thread {  4 usages  Henrique Leão +1

    // Cada thread guarda seu próprio DTO com as infos da tarefa dela
    private final VideoDTO videoDTO;  4 usages

    // Construtor simples pra receber o DTO com o "pedaço" do vídeo que ela vai processar
    public Thread(VideoDTO videoDTO) {  1 usage  Henrique Leão
        this.videoDTO = videoDTO;
    }

    // É aqui que a mágica acontece quando a gente dá um thread.start()
    // A única coisa que ela faz é chamar os métodos de processamento, passando o DTO dela
    @Override  Henrique Leão
    public void run() {
        VideoProcessingMethods.processarVideo(videoDTO);
    }
```

O componente central desta classe é o atributo ***videoDTO***, que armazena todas as informações que a thread precisa para executar sua tarefa. Ao criar a thread, passamos este VideoDTO através de seu construtor. Isso é fundamental para a nossa estratégia de paralelismo, pois é assim que cada thread "sabe" qual é o seu pedaço específico do vídeo a ser processado, definido pelos ***limiteInferior*** e ***limiteSuperior*** contidos no DTO.

O coração da execução da thread é o método ***run()***. Quando o método ***start()*** é chamado em uma de nossas threads, a JVM inicia um novo fluxo de execução e invoca o conteúdo deste método. Decidimos manter a classe *Thread* enxuta e com uma única responsabilidade: executar a tarefa. Por isso, dentro do run(), nós

simplesmente chamamos o método ***VideoProcessingMethods.processarVideo()***, delegando a ele toda a lógica complexa de aplicação dos filtros.

Por fim, adicionamos um método auxiliar, ***getQuantFrames***(), que simplesmente retorna o número de frames que foram designados para aquela thread. Isso se mostrou útil durante o desenvolvimento e depuração para verificar se a carga de trabalho estava sendo distribuída corretamente entre as threads.

Classe: VideoProcessing.java

Método: *main()*

O método main atua como o orquestrador central do programa. Ele é responsável pela inicialização, gerenciamento dos testes de desempenho e finalização do processo de restauração do vídeo.

```java
public static void main(String[] args) {  Rhuan Azevedo +3 *
    // Vendo o tamanho da heap que a JVM alocou, só pra ter uma ideia do consumo de memória
    System.out.println("Heap inicial (Xms): " + Runtime.getRuntime().totalMemory() / (1024 * 1024) + " MB");
    System.out.println("Heap máximo  (Xmx): " + Runtime.getRuntime().maxMemory() / (1024 * 1024) + " MB");

    String caminhoVideo = "src/main/videos/video-cortado.mp4";
    String caminhoGravar = "src/main/videos/video-pos-borrao.mp4";
    double fps = 24.0;

    System.out.println("Carregando o vídeo do diretório: " + caminhoVideo);
    byte video[][][] = carregarVideo(caminhoVideo);

    System.out.printf("N° de Frames: %d   Resolução: %d x %d \n",
            video.length, video[0][0].length, video[0].length);
```

No início da execução, é inspecionado e exibido o tamanho inicial e máximo da memória Heap alocada pela Máquina Virtual Java (JVM). Essa verificação é relevante no contexto de Arquitetura de Computadores, pois o processamento de vídeo é uma tarefa que consome uma quantidade significativa de memória, e entender seus limites é fundamental.

- **Configurando os parâmetros do processamento:**

```java
byte[][][] videoProcessado = null;
// Definindo os parâmetros dos filtros -> 'final' pra garantir que ninguém mude sem querer
final int tamanhoLadoMascara = 3;
final TipoDeCalculo tipoDeCalculoSalPimenta = TipoDeCalculo.MEDIA;
final TipoDeCalculo tipoDeCalculoBorrao = TipoDeCalculo.MEDIANA;

// Usando um DTO pra não passar 500 parâmetros na chamada dos métodos de correção
VideoDTO videoDTO = new VideoDTO(video);
// Criamos os arrays de resultado aqui fora. Assim a gente passa eles como referência
// e as threads os modificam. Evita criar um monte de array gigante dentro do loop.
videoDTO.setVideoPosSalPimenta(new byte[video.length][video[0].length][video[0][0].length]);
videoDTO.setVideoPosCorrecaoBorroes(new byte[video.length][video[0].length][video[0][0].length]);
videoDTO.setTamanhoLadoMascara(tamanhoLadoMascara);
videoDTO.setTipoDeCalculoSalPimenta(tipoDeCalculoSalPimenta);
videoDTO.setTipoDeCalculoBorrao(tipoDeCalculoBorrao);
```

A configuração dos parâmetros dos filtros foi centralizada diretamente no main. Ao declará-los como *final*, seus valores são definidos como constantes durante toda a execução. Essa abordagem facilita a experimentação, pois permite alterar o comportamento dos filtros (como o tamanho da máscara ou o tipo de cálculo) em um único e acessível local.

Neste bloco, uma instância da classe **VideoDTO** é criada e populada com todas as informações necessárias para o processamento. Isso inclui o vídeo original, os buffers de saída (que são alocados aqui) e todos os parâmetros definidos anteriormente. Ao encapsular tudo em um único objeto, a chamada ao método de processamento se torna mais limpa e a gestão dos dados, mais organizada e robusta.

- **Loop Principal:**

```
System.out.println("\nIniciando Processamento!");
// Loop principal pra testar a performance com 1, 2, 4, 8... threads
// O numThreads dobra a cada iteracao, fazendo todos os testes requisitados
for (int numThreads = 1; numThreads <= 32; numThreads *= 2) {
    System.out.println("Execução com " + numThreads + " thread(s)");
    // Pega o tempo antes...
    long inicio = System.currentTimeMillis();
    // Chamando a função que faz a magica acontecer
    videoProcessado = processarVideo(videoDTO, numThreads);
    // ...e depois, pra gente poder calcular o speedup
    long fim = System.currentTimeMillis();
    System.out.println("Tempo de execução com " + numThreads + " Thread(s): " + (fim - inicio));
    System.out.println("--------------------------\n");
}
```

Este laço de controle é o motor dos testes de desempenho, executando o processamento para cada configuração de threads exigida no enunciado do trabalho. Ele começa com a execução sequencial (1 thread) e dobra o número de threads a cada iteração até 32. Dentro do loop, o tempo de execução da chamada *processarVideo* é medido com precisão. Os resultados são impressos para permitir a análise posterior do ganho de desempenho (speedup).

```java
// Se o video estiver nulo, ele nao sera salvo
if (videoProcessado == null) return;

System.out.println("Salvando...  " + caminhoGravar);
gravarVideo(videoProcessado, caminhoGravar, fps);
System.out.println("Término do processamento");
```

Na etapa final, após a conclusão de todos os testes, o programa realiza uma verificação para garantir que o processamento retornou um resultado válido. Em seguida, a função *gravarVideo* é chamada. Conforme disponibilizado no enunciado, ela recebe a matriz final *videoProcessado*, codifica os frames e salva o vídeo corrigido em um novo arquivo .mp4 no disco, concluindo o trabalho.

Método: *processarVideo()*

Esse método funciona como um despachante. Sua única responsabilidade é direcionar o fluxo de execução para o caminho de processamento correto — sequencial ou paralelo — com base no número de threads solicitado. Ele delega a lógica de processamento real para outros métodos.

```java
private static byte[][][] processarVideo(VideoDTO videoDTO, int numThreads) {
    // caso base: execução sequencial (sem criar threads)
    // -> é o nosso benchmark pra comparar o desempenho
    if (numThreads == 1) return processarSequencial(videoDTO);

    return processarParalelo(videoDTO, numThreads);
}
```

Assim, é verificado se a execução foi solicitada em modo sequencial, ou seja, com apenas uma thread. Se a condição for verdadeira, a execução é imediatamente delegada ao método ***processarSequencial***. Este modo é executado para estabelecer um *benchmark* de desempenho, contra o qual os resultados do processamento paralelo serão comparados para análise de *speedup*.

Caso o número de threads seja maior que um, a execução é delegada ao método ***processarParalelo***. Esta é a implementação do principal objetivo do trabalho, que é otimizar o tempo de correção através do uso de múltiplos núcleos de processamento. O método ***processarParalelo*** contém toda a lógica para dividir o trabalho, criar, executar e sincronizar as threads.

Método: *processarSequencial ()*

Este método encapsula a lógica para a execução puramente sequencial do pipeline de filtros. Todo o processamento ocorre na thread principal da aplicação, sem a criação de novos fluxos de execução.

```
private static byte[][][] processarSequencial(VideoDTO videoDTO) {  1 usage   Rhuan Azevedo +1
    // Configura o DTO para processar o vídeo inteiro em um único "chunk"
    // Define o início do processamento no primeiro frame
    videoDTO.setLimiteInferior(0);
    // Define o fim do processamento no último frame
    videoDTO.setLimiteSuperior(videoDTO.getVideoPreProcessamento().length);
    // Invoca o metodo que contém a logica de aplicação dos filtros
    VideoProcessingMethods.processarVideo(videoDTO);

    // Retorna a referência para o buffer de vídeo com o resultado final
    return videoDTO.getVideoPosCorrecaoBorroes();
}
```

O objeto *videoDTO* é configurado para que o processamento abranja o vídeo completo. O ***limiteInferior*** é definido como 0 (o primeiro frame) e o ***limiteSuperior*** é definido como o comprimento total do array de frames. Isso garante que o método de processamento subsequente itere sobre todos os frames do vídeo em uma única passagem, simulando uma execução sem divisão de tarefas.

Então, é invocado o método que contém a lógica de aplicação dos filtros de imagem. Após a conclusão de todo o processamento, é retornada a referência para a matriz tridimensional que contém o resultado final. O buffer ***videoPosCorrecaoBorroes*** foi modificado diretamente pelo método de processamento e agora contém o vídeo com todos os filtros de correção aplicados.

Método: *processarParalelo()*

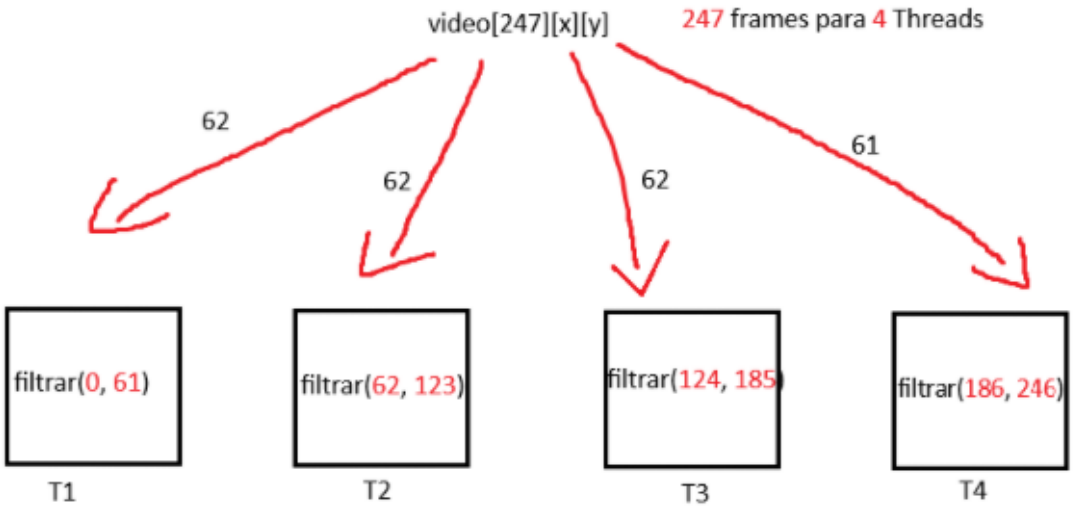
Este método é responsável por implementar a estratégia de processamento paralelo, que visa otimizar e encurtar o tempo de execução da correção do vídeo através do uso de múltiplos núcleos de processamento. Ele coordena a divisão do trabalho, a criação e execução de threads e, por fim, a sincronização dos resultados.

```
private static byte[][][] processarParalelo(VideoDTO videoDTO, int numThreads) {  1 usage  Henri
    // Exemplo para que utilizamos para elaboraçao da logica de divisao de frames para as threa
    //video[247] 246 indices (do 0 ao 246)
    //247 frames para 4 threads
    //numFramesPorThread = 61
    //Resto = 3

    // Variáveis pra controlar a divisão do trabalho entre as threads.
    Thread thread;
    List<Thread> threads = new ArrayList<>();
    int numFramesPorThread = videoDTO.getVideoPreProcessamento().length / numThreads,
            resto = videoDTO.getVideoPreProcessamento().length % numThreads,
            limiteInferior = 0,
            limiteSuperior,
            numFramesAlocados;
```

Neste bloco inicial, são declaradas as variáveis que controlarão todo o processo de paralelização.

- ***threads***: lista que armazenará as instâncias de todas as threads criadas, permitindo que a thread principal possa gerenciá-las e aguardar sua conclusão posteriormente.
- ***numFramesPorThread*** e ***resto***: implementam a lógica de balanceamento de carga. *numFramesPorThread* calcula a quantidade base de frames que cada thread processará através de uma divisão inteira. *resto* armazena o número de frames que sobram dessa divisão, os quais serão distribuídos entre as primeiras threads para garantir que todo o vídeo seja processado e a carga de trabalho seja a mais equilibrada possível.
- ***limiteInferior***, ***limiteSuperior*** e ***numFramesAlocados***: Variáveis auxiliares para calcular e definir o segmento ("chunk") exato de frames para cada thread dentro do laço de criação.




```java
// Loop que cria e configura cada thread pra processar um "pedaço" do vídeo.
for (int i = 0; i < numThreads; i++) {
    numFramesAlocados = numFramesPorThread;
    // Se a divisão não for exata, distribui o resto dos frames pras primeiras threads.
    // Cada uma pega um frame a mais até o resto acabar.
    if (resto > 0) {
        numFramesAlocados++;
        resto--;
    }
    limiteSuperior = limiteInferior + numFramesAlocados;

    // Cria a thread, passando o DTO com os frames que ela vai processar (do limiteInferior ao Superior).
    thread = new Thread(new VideoDTO(videoDTO, limiteInferior, limiteSuperior));
    // Atualiza o limite inferior pra próxima thread já começar do lugar certo.
    limiteInferior = limiteSuperior;

    // Inicia a thread. Agora ela tá rodando "em paralelo" com a main.
    thread.start();
    // Guarda a thread na lista para podermos aguardar ela terminar
    threads.add(thread);
}
```

Este é o loop de criação e execução das threads. Para cada uma das ***numThreads*** solicitadas, o laço executa os seguintes passos:

1. **Alocação de Frames:** Calcula o número exato de frames que a thread atual irá processar, adicionando um frame extra se ainda houver um resto a ser distribuído.

2. **Criação da Thread:** Uma nova instância da classe **Thread** customizada é criada. Crucialmente, ela recebe um novo ***videoDTO*** que, através de um construtor de cópia, herda todos os parâmetros e referências aos buffers, mas possui valores de ***limiteInferior*** e ***limiteSuperior*** únicos, definindo sua fatia de trabalho exclusiva.
3. **Execução:** O método ***thread.start()*** é invocado, chamando o método run() da thread, que por sua vez inicia o processamento dos filtros em seu segmento de vídeo designado. A partir deste ponto, a thread opera de forma concorrente com a thread principal e as demais.
4. **Armazenamento:** A referência da thread recém-criada é adicionada à lista threads para a etapa de sincronização.

```
// esperamos todas as threads finalizarem suas tasks
for (Thread t : threads) {
    try {
        // O .join() faz a thread main 'pausar' aqui até a thread 't' terminar.
        // Sem isso, tentariamos retornar o vídeo antes de ele estar pronto.
        t.join();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
// Agora que todas as threads terminaram, o vídeo de resultado tá completo.
return videoDTO.getVideoPosCorrecaoBorroes();
```

Esta parte é fundamental para garantir a sincronização e a integridade do resultado final. Após ter iniciado todas as threads, a thread principal itera sobre a lista threads e chama o método ***t.join()*** em cada uma. Esse método bloqueia a thread principal, forçando-a a esperar até que a thread ***t*** tenha concluído a execução do seu método ***run()***. Sem este passo, a thread principal continuaria sua execução e retornaria o resultado antes que as threads de trabalho tivessem a chance de processar o vídeo, resultando em um arquivo incompleto ou incorreto.

Após o loop de ***join()*** ser concluído, o ***videoPosCorrecaoBorroes*** foi completamente preenchido por todas as threads, então é retornada sua referência, que contém o vídeo final e totalmente processado.

Classe: VideoProcessingMethods.java

Método: *removerSalPimenta()*

O objetivo do *removerSalPimenta* é aplicar um filtro espacial para corrigir ruídos de pixels que aparecem como pontos brancos ou pretos isolados em um fundo normal em cada frame do vídeo. Esse tipo de filtro é útil para eliminar ruídos sem afetar muito a nitidez da imagem.

Diferente do filtro temporal (*removerBorroesTempo*), que trabalha com a evolução de um pixel ao longo dos frames, o filtro espacial atua sobre os pixels vizinhos dentro do mesmo frame, substituindo os valores por uma média ou mediana dos valores ao redor, a depender do que foi optado pelo usuário.

Explicação do método:

```
private static void removerSalPimenta(VideoDTO videoASerProcessado) {  1 usage   Henrique Leão +1 *

    // Itera sobre cada frame que foi designado para esta thread
    for (int indiceFrame = videoASerProcessado.getLimiteInferior(); indiceFrame < videoASerProcessado.getLimiteSuperior();
         indiceFrame++) {

        // Pega o frame atual do vídeo de entrada para processar
        byte[][] frame = videoASerProcessado.getVideoPreProcessamento()[indiceFrame];

        // Define as dimensões da máscara (kernel) para o filtro, ex: 3x3
        int ladoMascara = videoASerProcessado.getTamanhoLadoMascara(),
                // Calcula o deslocamento para acessar os valores vizinhos
                metadeLado = ladoMascara / 2,
                // Calcula a quantidade de vizinhos pra criar o array com o tamanho certo
                numPixelsVizinhos = ladoMascara * ladoMascara - 1,
                // Variável que vai guardar o novo valor do pixel pós-filtro
                pixelProcessado = 0;

        // Obtém qual cálculo usar (Média ou Mediana) a partir dos parâmetros
        TipoDeCalculo tipoDeCalculo = videoASerProcessado.getTipoDeCalculoSalPimenta();
```

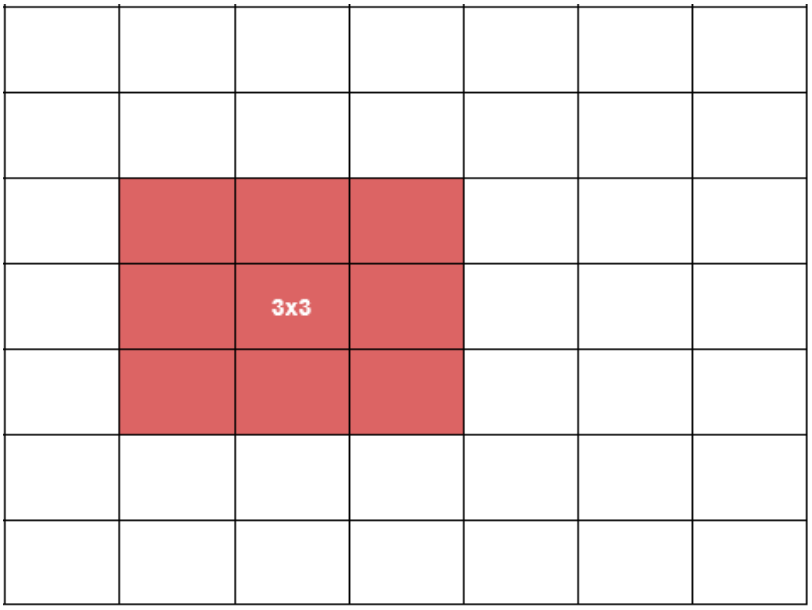
Inicialmente o método começa iterando sobre todos os frames para que o filtro seja aplicado em todos. Para isso, o *for* inicia do limite inferior e vai de um em um até o limite superior de cada thread, sendo executado x vezes baseado na quantidade de frames responsável por cada thread.

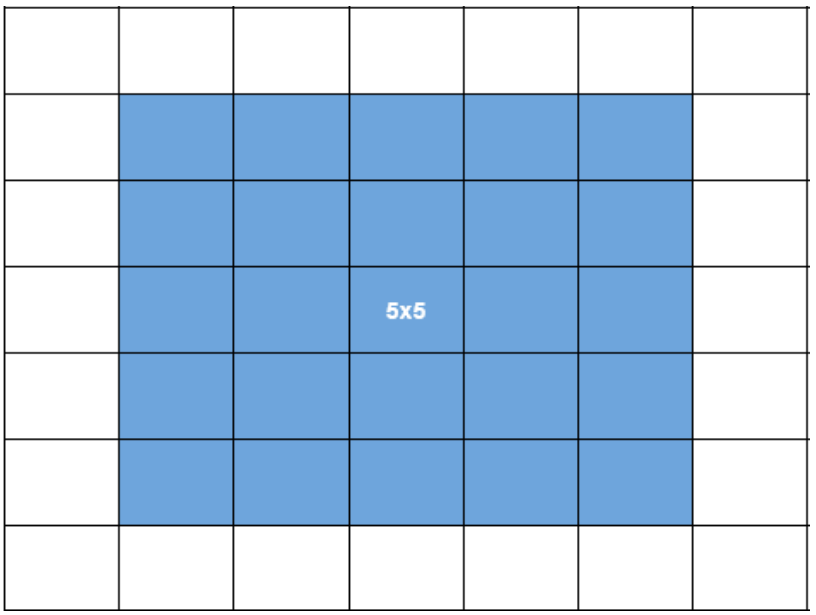
Após isso, pegamos o frame que vai ser processado na iteração e logo em seguida definimos a máscara baseado no valor que foi definido no código (3, no nosso caso). Esses valores são respectivamente armazenados em *frame* e *ladoMascara*.

A variável *ladoMascara* define a dimensão da "janela" de análise, como 3x3 nesse caso. Ela controla a intensidade do efeito do filtro. Uma máscara maior considera uma vizinhança mais ampla, resultando em uma suavização mais forte e agressiva, enquanto uma máscara menor produz um efeito mais sutil e localizado.

A partir da máscara, a variável *metadeLado* é calculada pela divisão inteira de *ladoMascara* por 2, a fim de calcular o deslocamento necessário para acessar todos os pixels vizinhos em relação a um ponto central. Ao iniciar os laços de iteração a uma distância de *metadeLado* da borda, o código cria uma margem que impede o filtro de tentar ler pixels fora da imagem, o que causaria um erro e interromperia o programa.Nessa abordagem, os pixels da borda são ignorados.

Observação: A máscara precisa ser um valor ímpar para que possa haver um pixel central, que armazenará a média ou mediana dos pixels vizinhos.





Em seguida, é calculado a quantidade de valores que serão armazenados para que se possa calcular a média ou mediana que será atribuída ao pixel central da iteração. O cálculo é feito elevando a área da máscara (*ladoMascara)* ao quadrado e subtraindo 1 (o valor do pixel central não é considerado). Esse valor é armazenado em *numPixelsVizinhos*, que será utilizado posteriormente para definir o tamanho do vetor que irá armazenar os valores ao redor do pixel central.

Por fim, *pixelProcessado*, que será utilizada para armazenar o novo valor do pixel central, é inicializada com zero, e *tipoDeCalculo* é obtido a partir dos parâmetros definidos no objeto videoASerProcessado, indicando se o filtro deve aplicar média ou mediana.

```java
// Itera sobre cada pixel do frame, ignorando as bordas
for (int i = metadeLado; i < frame.length - metadeLado; i++) {
    for (int j = metadeLado; j < frame[i].length - metadeLado; j++) {

        // Array pra guardar os valores dos pixels vizinhos
        int[] vizinhos = new int[numPixelsVizinhos];
        int index = 0;

        // Monta a lista de vizinhos do pixel atual
        for (int x = -metadeLado; x <= metadeLado; x++) {
            for (int y = -metadeLado; y <= metadeLado; y++) {
                // Pula o pixel do centro (x=0, y=0) pra pegar só os vizinhos
                if (x != 0 || y != 0) {
                    // '& 0xFF' converte o byte (signed) pra um int "unsigned" (0 a 255)
                    vizinhos[index++] = frame[i + x][j + y] & 0xFF;
                }
            }
        }
```

Após definida a máscara, o método inicia a iteração sobre cada pixel interno do frame, ignorando as bordas para evitar acesso a posições inválidas no array. Isso é feito começando dos índices *metadeLado* e indo até *frame.length* - *metadeLado*, garantindo que a máscara 3x3 sempre tenha vizinhos válidos ao redor do pixel central.

Esse duplo *for* percorre linha por linha (*i*) e coluna por coluna (*j*) do frame, delimitando quais pixels serão processados. Dentro desse laço, é criado um array chamado *vizinhos*, que armazenará os valores dos pixels ao redor do pixel central atual.

Em seguida, mais dois *for* internos percorrem os deslocamentos horizontais (*x*) e verticais (*y*) dentro da janela da máscara (por exemplo, de -1 a 1 para uma máscara 3x3). Durante essa varredura, é feita uma verificação para ignorar o pixel central (quando *x* == 0 e *y* == 0), pois o foco está apenas nos vizinhos.

Os valores dos pixels vizinhos são então coletados diretamente do frame com o deslocamento apropriado (*i* + *x*, *j* + *y*). Para garantir que esses valores fiquem no intervalo de 0 a 255 (e não de -128 a 127, como é padrão em bytes Java), é usado o

operador & 0xFF, que converte o byte assinado para um inteiro sem sinal. Esses valores convertidos são armazenados no array *vizinhos* para posterior cálculo da média ou mediana.

```
// Operador ternário pra escolher o cálculo de forma concisa
pixelProcessado = tipoDeCalculo.equals(TipoDeCalculo.MEDIA)
        ? calcularMediaEntreVizinhos(vizinhos)
        : pegarMedianaEntreVizinhos(vizinhos);

// Escreve o resultado no buffer de saída
videoASerProcessado.getVideoPosSalPimenta()[indiceFrame][i][j] = (byte) pixelProcessado;
```

Após coletar todos os valores dos vizinhos do pixel atual, o próximo passo é decidir como calcular o novo valor do pixel. Para isso, é utilizado um operador ternário, que de forma concisa verifica se o tipo de cálculo definido nos parâmetros é MEDIA.

Caso seja, a função *calcularMediaEntreVizinhos* é chamada e retorna a média dos valores coletados. Caso contrário, a função *pegarMedianaEntreVizinhos* é utilizada, retornando a mediana dos mesmos valores..

Após o cálculo, o valor resultante (*pixelProcessado*) é convertido para byte e escrito no frame de saída, exatamente na mesma posição do pixel original. Essa substituição garante que o novo frame refletirá o efeito do filtro aplicado, mantendo a estrutura geral da imagem.

Método: *removerBorroesTempo()*

O propósito fundamental do *removerBorroesTempo* é aplicar um filtro temporal. Diferente do filtro espacial (*removerSalPimenta*), que trabalha com os pixels vizinhos dentro de um único frame, o filtro temporal trabalha com o mesmo pixel através de múltiplos frames consecutivos. O objetivo é suavizar variações bruscas de um pixel ao longo do tempo, o que na prática ajuda a:

- Reduzir o efeito de "fantasma" ou "rastro" (blur de movimento).
- Estabilizar pixels que podem estar piscando ou tremendo devido à má qualidade da fita.
- Fluxo de Funcionamento Detalhado

O método pode ser dividido em 5 etapas principais:

1. Inicialização e Parâmetros

Primeiro, o método define seus parâmetros e prepara as variáveis necessárias:

```
// Parâmetros de configuração do filtro temporal
int janelaTemporalFrames = 2, // Raio da janela, 2 significa pegar 2 frames antes e 2 depois
        limiteInferior = videoASerProcessado.getLimiteInferior(), // Início do chunk da thread
        limiteSuperior = videoASerProcessado.getLimiteSuperior(); // Fim do chunk da thread

// Buffers de entrada (leitura) e saída (escrita) pra fazer o processamento out-of-place
byte[][][] video = videoASerProcessado.getVideoPosSalPimenta(),
        videoPos = videoASerProcessado.getVideoPosCorrecaoBorroes();
```

- **int janelaTemporalFrames = 2;:** Define o "raio" da janela de análise. Isso significa que, para cada frame, o algoritmo irá olhar para 3 frames anteriores e 3 frames posteriores, totalizando uma janela de 7 frames (3 + 1 atual + 3).
- **limiteInferior, limiteSuperior:** Obtém os limites de processamento da thread atual a partir do *VideoDTO*.
- **video, videoPos:** Pega as referências das matrizes. *video* é a matriz de leitura (que já passou pelo filtro *removerSalPimenta*) e *videoPos* é a matriz de escrita (o resultado final).

2. Tratamento de Bordas (Frames Iniciais e Finais)

Esta é uma parte crítica e sutil do algoritmo, indicada pelos comentários no código. Um filtro temporal precisa de frames "vizinhos" para funcionar. Os frames no início e no fim do vídeo (ou de uma fatia de processamento) não possuem todos os vizinhos necessários.

```
int primeiroFrame = limiteInferior != 0
        ? limiteInferior
        : limiteInferior + janelaTemporalFrames;
int ultimoFrame = limiteSuperior <= video.length - janelaTemporalFrames
        ? limiteSuperior
        : limiteSuperior - janelaTemporalFrames;
TipoDeCalculo tipoDeCalculo = videoASerProcessado.getTipoDeCalculoBorrao();
```

int primeiroFrame = Esta linha decide a partir de qual frame o processamento deve começar.

- Se a thread atual não é a primeira (*limiteInferior* != 0), ela pode começar a processar desde o seu primeiro frame (*limiteInferior*), pois seus "vizinhos" anteriores existem na fatia da thread anterior.
- Se a thread é a primeira (*limiteInferior* == 0), ela não pode processar os frames 0, 1 e 2, pois eles não têm 3 frames anteriores. A solução adotada foi pular esses frames, começando o processamento a partir de *limiteInferior* + *janelaTemporalFrames* (frame 3).

int ultimoFrame = De forma similar, esta linha define até onde o processamento pode ir.

- Se a thread não é a última do vídeo, ela pode processar até o final de sua fatia.
- Se for a última thread, ela precisa parar *janelaTemporalFrames* (3) frames antes do fim do vídeo, pois os últimos frames não têm 3 vizinhos posteriores.

Consequência: Essa abordagem faz com que os primeiros e os últimos 3 frames do vídeo completo não sejam processados por este filtro de borrão, sendo efetivamente copiados da etapa anterior.

3. Iteração por Frames e Pixels

O núcleo do método consiste em três laços for aninhados:

```
for (int indiceFrame = primeiroFrame; indiceFrame < ultimoFrame; indiceFrame++) {

    byte[][] frameAtual = video[indiceFrame];
    int altura = frameAtual.length;      // 720
    int largura = frameAtual[0].length;  // 960

    for (int indicePixelAltura = 0; indicePixelAltura < altura; indicePixelAltura++) {   // 0 -> 720
        for (int indicePixelLargura = 0; indicePixelLargura < largura; indicePixelLargura++) { // 0 -> 960, 720 vezes
```

Essa estrutura garante que o código irá visitar cada pixel de cada frame dentro dos limites de processamento válidos definidos na etapa anterior.

4. Coleta de Dados na Janela Temporal

Para cada pixel visitado, o próximo passo é coletar os valores desse mesmo pixel ao longo da janela de tempo:

```
for (int indicePixelLargura = 0; indicePixelLargura < largura; indicePixelLargura++) { // 0 -> 960, 720 vezes

    int[] valoresTemporais = new int[(2 * janelaTemporalFrames + 1)];
    int index = 0;

    for (int deslocamento = -janelaTemporalFrames; deslocamento <= janelaTemporalFrames; deslocamento++) {
        valoresTemporais[index] = video[indiceFrame + deslocamento][indicePixelAltura][indicePixelLargura] & 0xFF;
        index++;
    }
```

- Um array valoresTemporais é criado.
- Um laço percorre os deslocamentos de -3 a +3.
- Em cada iteração, ele acessa o mesmo pixel ([indicePixelAltura][indicePixelLargura]), mas em um frame diferente ([indiceFrame + deslocamento]).

- A operação & 0xFF é usada para garantir que o valor do byte (que em Java é assinado, de -128 a 127) seja tratado como um inteiro sem sinal (0 a 255), que é a representação correta para uma cor em escala de cinza.

Ao final deste laço, o array valoresTemporais contém 7 valores de intensidade daquele pixel ao longo do tempo.

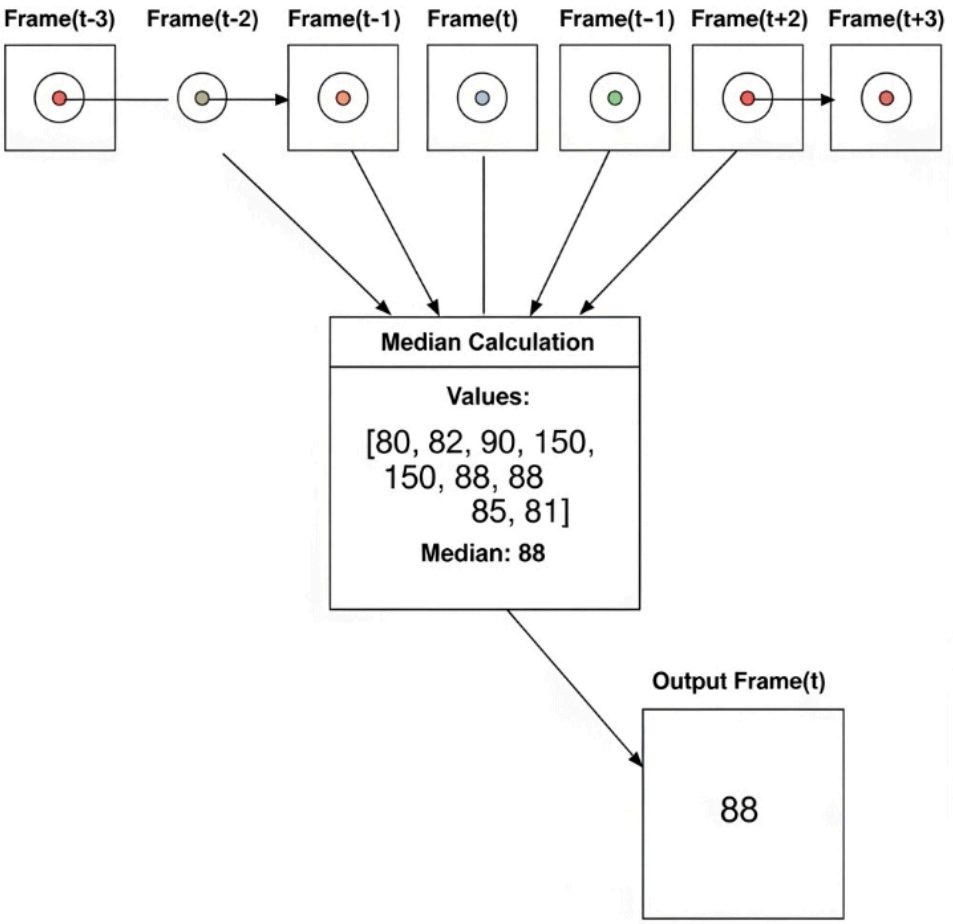
5. Cálculo do Novo Pixel e Atualização

Com os dados coletados, o passo final é calcular o novo valor do pixel e gravá-lo na matriz de saída:

```
int pixelProcessado = tipoDeCalculo.equals(TipoDeCalculo.MEDIA)
        ? calcularMediaEntreVizinhos(valoresTemporais)
        : pegarMedianaEntreVizinhos(valoresTemporais);
videoPos[indiceFrame][indicePixelAltura][indicePixelLargura] = (byte) pixelProcessado;
```

- O programa verifica qual tipo de cálculo deve ser feito (no caso, a MEDIANA, conforme a configuração padrão).
- O método pegarMedianaEntreVizinhos é chamado, que ordena os 7 valores temporais e retorna o valor central.
- Este valor calculado (pixelProcessado) é então gravado na posição correspondente na matriz de saída videoPos.

Ao substituir o valor de um pixel pela mediana de seus valores ao longo do tempo, o algoritmo efetivamente remove picos e vales de intensidade, resultando em uma imagem mais estável e com menos borrões.



Método: *calcularMediaEntreVizinhos()*

```
// Helper pra calcular a média simples de um array de pixels
private static int calcularMediaEntreVizinhos(int[] vizinhos) {
    int soma = 0;
    // Acumula a soma de todos os valores
    for (int valorPixelVizinho : vizinhos) {
        soma += valorPixelVizinho;
    }
    // Retorna a divisão inteira
    return soma / vizinhos.length;
}
```

O método auxiliar *calcularMediaEntreVizinhos* é chamado para processar o array *vizinhos*. Ele recebe como entrada o array vizinhos, que contém os valores dos pixels ao redor do pixel central, e retorna a média aritmética entre eles.

Inicialmente, é criada a variável soma, que será usada para acumular a soma de todos os valores presentes no array. Um laço *for-each* percorre cada elemento (*valorPixelVizinho*) e vai somando esse valor à variável acumuladora. Ao final da iteração, o método realiza a divisão inteira da soma pelo número total de elementos, retornando o valor médio arredondado para baixo (por ser divisão inteira com int).

Esse valor será então usado como o novo valor do pixel no frame de saída. Esse tipo de cálculo tende a suavizar a imagem, reduzindo variações bruscas entre pixels vizinhos, embora possa ser menos eficaz que a mediana para ruídos extremos.

Método: *pegarMedianaEntreVizinhos()*

```java
// Helper pra pegar a mediana, ATENÇÃO: o array de entrada é ordenado e modificado
private static int pegarMedianaEntreVizinhos(int[] vizinhos) {  2 usages  Henrique Leão +1 *
    // Ordena o array pra achar o valor do meio
    Arrays.sort(vizinhos);

    // Tratamento pro caso (improvável) de array com tamanho par
    if (vizinhos.length % 2 == 0) {
        return (vizinhos[(vizinhos.length / 2) - 1] + vizinhos[(vizinhos.length / 2)]) / 2;
    }

    // Se for ímpar, a mediana é só pegar o valor do meio
    return vizinhos[vizinhos.length / 2];
}
```

O método auxiliar *pegarMedianaEntreVizinhos*, assim como o *calcularMediaEntreVizinhos*, processa o array *vizinhos*. Essa função recebe o array contendo os valores dos pixels ao redor e tem como objetivo retornar o valor central (mediana), que representa o dado mais estável dentro do conjunto.

O primeiro passo dentro da função é ordenar o array com *Arrays.sort(vizinhos)*. Isso é essencial, já que a mediana depende da posição relativa dos elementos.

Após ordenado, o método verifica se o tamanho do array é par. Se for, ele retorna a média dos dois valores centrais. Caso contrário (tamanho ímpar), simplesmente retorna o elemento que está exatamente no meio do array.

Resultados e Questões:

Processador: Intel Core i5-10400F (6 núcleos / 12 threads, até 4.3 GHz)

Memória RAM: 16 GB DDR4 2666 MHz

Armazenamento: SSD Lexar NVMe 3.0 de 512 GB (Leitura 3500 MB/s, Gravação 2400 MB/s

Sistema Operacional: Windows 11

	Tempo (ms)	
Configuração	**Média**	**Mediana**
Sequencial	62454	122976
2	31740	64245
4	17466	34020
8	12087	23455
16	11661	20606
32	11854	21076

Q1.: Como o tempo de execução varia conforme aumentamos o número de cores?

Resp: Conforme aumentamos o número de threads, o tempo de execução diminui drasticamente, até um certo ponto. Isso acontece porque o processamento do vídeo está sendo dividido entre múltiplos "trabalhadores" (as threads), ou seja, em vez de um único core processar tudo sequencialmente, vários cores trabalham em paralelo,

cada um em um pedaço do vídeo. Assim, quanto mais threads, menor o tempo de execução.

No entanto, a partir de certo ponto, o aumento no número de threads deixa de trazer ganhos relevantes. Isso ocorre pelo número de threads ultrapassar o número real de cores, e elas começarem a disputar os mesmos recursos, o que gera contenção e não acelera mais o processamento, além do princípio da Lei de Amdahl, que afirma que sempre existe uma parte do código que não pode ser paralelizada, e que o ganho de velocidade máximo que você pode obter ao paralelizar um programa é limitado por essa parte que não pode ser paralelizada.

Além disso, observamos que o tempo de execução da Mediana é consistentemente maior que o da Média, mesmo com o mesmo número de threads. Isso acontece porque o cálculo da mediana exige a ordenação do array de vizinhos (com Arrays.sort), o que tem uma complexidade maior (O(n log n)) em comparação com a média, que apenas soma os valores e divide (O(n)). Esse custo adicional da ordenação, repetido para cada pixel, torna o filtro de mediana naturalmente mais lento.

Em resumo, o tempo de execução é inversamente proporcional ao número de threads/cores até um certo limite, após o qual o ganho se estabiliza ou até piora por conta dos custos paralelos.

Q2.: Qual o comportamento do tempo de execução quando o número de threads excede a quantidade de cores físicos instalados na máquina utilizada para a execução dos testes?

Resp: Quando o número de threads excede a quantidade de cores físicos da máquina, o tempo de execução tende a deixar de diminuir significativamente e, em alguns casos, pode até aumentar ligeiramente.

Isso ocorre pois múltiplas threads passam a disputar os mesmos recursos físicos do processador, como cache, registradores e ciclos de execução, gerando contenção e sobrecarga.

No caso de processadores que possuem hyper-threading (threads lógicas), o ganho de desempenho é limitado, já que essas threads não correspondem a núcleos físicos reais, ou seja, elas compartilham o mesmo hardware interno. Dessa forma, o paralelismo continua ocorrendo, mas com eficiência reduzida.

Em resumo, quando o número de threads ultrapassa o número de cores físicos (ou o ponto ideal de paralelismo), o tempo de execução se estabiliza ou cresce, devido ao limite da capacidade de paralelização real da máquina.

Considerações Finais

O desenvolvimento deste trabalho envolveu a aplicação prática de diversos conceitos fundamentais da Programação Orientada a Objetos, paralelismo e manipulação de estruturas de dados em Java. A construção de uma solução capaz de aplicar filtros de média e mediana em vídeos, de forma sequencial e paralela, exigiu atenção à organização do código, encapsulamento de dados, uso adequado de threads e compreensão do impacto do paralelismo no desempenho computacional.

Durante o processo, enfrentamos desafios como a divisão dos frames para cada thread e a aplicação dos filtros de forma eficaz no vídeo. A análise dos resultados também foi importante para entender limites práticos da paralelização, como a contenção de recursos e os efeitos da Lei de Amdahl.

Por fim, o trabalho foi extremamente enriquecedor. Ele proporcionou uma experiência concreta de desenvolvimento orientado a objetos aliado ao processamento paralelo, permitindo aplicar na prática os conceitos aprendidos em sala de aula. Além disso, contribuiu para o aprimoramento da capacidade de abstração, raciocínio lógico e resolução de problemas, trazendo grande satisfação ao ver o sistema funcionando corretamente e cumprindo todos os requisitos propostos.