Real-Time Neural Style Transfer on 4K Video

Concurrent Programming

Fatima Atwi 6438
Rokaya Alameen 6172

July 2025

Contents

1 Abstract

This project aims to explore the application of parallel programming in Java to accelerate real-time neural style transfer on 4K video frames. Traditional style transfer is exhaustive, especially when applied to each pixel of large video frames. Our approach will process video frames concurrently using Java's Fork/Join framework, therefore maximizing hardware utilization, and leading to a less computationally expensive result. This report presents our methodology, implementation details, performance evaluation, and insights gained from optimizing the style transfer pipeline.

2 General Introduction

2.1 Motivation:

With the growing demand for high-quality video content, real-time style transfer in video has become increasingly relevant. However, the computational burden required to process large frames, such as 4K resolution, makes it challenging to apply it in real-world applications. Using parallel programming in Java, we aim to enhance performance and make real-time stylization of high-resolution videos more accessible and efficient. This work not only addresses a practical media challenge but also demonstrates the potential of Java's concurrency tools in exhaustive image processing tasks.

2.2 Objective:

The main goal of this project is to accelerate neural style transfer on 4K video frames using parallel programming in Java. First, we will implement a sequential baseline of the style transfer process to establish a performance reference point. We will then design a parallel version that uses Fork/Join to divide each frame into smaller tiles processed concurrently. We will also extend parallelism by processing multiple frames simultaneously to maximize CPU utilization. Throughout the project, we will evaluate performance based on execution speed, CPU usage, memory overhead, and scalability across different thread counts. By comparing the parallel implementation against the sequential one, we will be able to quantify the improvements gained.

2.3 Report Plan:

This report is organized into several key sections:

- **Abstract**: Provides a brief overview of the project, including its purpose, implementation, and significance.
- **General Introduction**: Explains the motivation behind the project, its objectives, and outlines the report structure.
- **Development**: Description of sequential processing pipeline and the limitations observed, as well as the parallel implementation, including the techniques used, challenges faced, and the explaining the core code of the project.

- **Results**: Presents the outcomes and performance of the two methods after implementation.
- **Conclusion**: Summarizes the project findings and discusses possible future improvements.
- **References**: Lists all sources and tools used during the development.

3 Filter Development

To simulate a style similar to Vincent Van Gogh artistic style we developed a custom filter in Java.The **applyVanGoghFilter** method applies a stylization pipeline to an input image by modifying colors, enhancing edges to reach the desired artistic style.

3.1 Color Space Transformation

For each pixel (excluding the borders), we retrieve the RGB color values using: `int rgb = src.getRGB(x, y);`. We extract the red, green, and blue channels, and convert them to the HSV color space using a helper method `rgbToHsv`. This allows more intuitive manipulation of color tones and intensities.

```
public static BufferedImage applyVanGoghFilter(BufferedImage src) {
        int width = src.getWidth();
        int height = src.getHeight();
        BufferedImage out = new BufferedImage(width, height, BufferedImage
            .TYPE_INT_RGB);

        for (int y = 1; y < height - 1; y++) {
            for (int x = 1; x < width - 1; x++) {
                int rgb = src.getRGB(x, y);
                int red = (rgb >> 16) & 0xFF;
                int green = (rgb >> 8) & 0xFF;
                int blue = rgb & 0xFF;

                float[] hsv = rgbToHsv(red, green, blue);
```

3.2 Hue Manipulation

For the blue and yellow hues of Van Gogh's artistic style, we alter the hue component of each pixel to be between these two colors. If the original hue is outside the 60–180° range, we shift it to 210, which represents blue. Otherwise, it is shifted to 50° which is yellow.

```
                if (hsv[0] < 60 || hsv[0] > 180) {
                    hsv[0] = 210f; // Blue
                } else {
                    hsv[0] = 50f; // Yellow
                }
```

3.3 Saturation

To add vibrancy and simulate the glowing effect of oil paint, the saturation and brightness (value) are slightly increased.

```
hsv[1] = Math.min(1f, hsv[1] * 1.3f);
hsv[2] = Math.min(1f, hsv[2] * 1.2f)
```

3.4 Posterization

After converting the HSV values back to RGB using `hsvToRgb`, we reduce the color depth to 4 levels per channel by rounding each channel value to the nearest multiple of 64. This gives the image an effect that reduces photorealistic details.

```
 int[] newRgb = hsvToRgb(hsv[0], hsv[1], hsv[2]);

newRgb[0] = (newRgb[0] / 64) * 64;
newRgb[1] = (newRgb[1] / 64) * 64;
newRgb[2] = (newRgb[2] / 64) * 64;
```

3.5 Edge Enhancement

To accentuate lines and features, a simple edge detection function `detectEdge` is applied. It calculates horizontal dx and vertical dy differences by comparing pixel brightness with its right and bottom neighbors.If the combined edge value exceeds a threshold, the pixel is darkened to make edges more pronounced.

```
    private static int detectEdge(BufferedImage img, int x, int y) {
        int rgb = img.getRGB(x, y);
        int right = img.getRGB(x + 1, y);
        int down = img.getRGB(x, y + 1);

        int dx = Math.abs((rgb & 0xFF) - (right & 0xFF));
        int dy = Math.abs((rgb & 0xFF) - (down & 0xFF));

        return dx + dy;
    }
```

```
   int edge = detectEdge(src, x, y);
if (edge > 60) {
    newRgb[0] = Math.max(0, newRgb[0] - 50);
    newRgb[1] = Math.max(0, newRgb[1] - 50);
    newRgb[2] = Math.max(0, newRgb[2] - 50);
}
```

3.6 Image Assembly

Each processed pixel is packed back into an RGB integer format and written into a new BufferedImage.

```
int finalRgb = (newRgb[0] << 16) | (newRgb[1] << 8) | newRgb[2];
out.setRGB(x, y, finalRgb);
```

3.7 Function Call

```
public static void processAndSave(String inputPath, String outputPath)
    throws Exception {
    BufferedImage input = ImageIO.read(new File(inputPath));
    BufferedImage output = applyVanGoghFilter(input);
    ImageIO.write(output, "jpeg", new File(outputPath));
}
```

This method calls the applyVanGoghFilter function to process the image and then saves it in a specified output path.

4 Sequential Approach

```
public static void processFolderSequential(File[] files, String
   outputDir) {
    for (File file : files) {
        try {
            String outPath = outputDir + "/" + file.getName().replace(
               ".jpeg", "_stylized.jpeg");
            processAndSave(file.getPath(), outPath);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The method processFolderSequential processes a set of image files one file after the other, meaning sequentially, then saves them in a specified directory. Its a public static method, can be accessed from outside the class, and does not need an object created to be called. It called the processAndSave method already defined and saves the image in a specified directory with adding _stylized to its name.

5 Parallel Approach

```
    public static void processFolderParallel(File[] files, String
       outputDir, int threads) throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(threads);
        for (File file : files) {
            executor.submit(() -> {
                try {
                    String outPath = outputDir + "/" + file.getName().
                       replace(".jpeg", "_stylized.jpeg");
                    processAndSave(file.getPath(), outPath);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        }
        executor.shutdown();
        executor.awaitTermination(1, TimeUnit.HOURS);
    }
```

This method processes image files in parallel, using a specified number of threads. Each image is stylized and saved with a _stylized.jpeg suffix.
ExecutorService executor = Executors.newFixedThreadPool(threads); creates a thread pool with a fixed number of worker threads. This controls how many tasks run concurrently.

In the for loop, it iterates over each file in files. For each file, it submits a lambda task to the thread pool. Each task does these:

- Constructs an output path by appending _stylized to the file name.
- Calls processAndSave(inputPath, outputPath) which stylizes and saves the image.
- Catches and prints exceptions.

6 Parallel Image Processing Function

```
    public static void processFolderParallel(File[] files, String
       outputDir, int threads) throws InterruptedException import java.
       util.concurrent.RecursiveAction;

public class ImageProcessingTask extends RecursiveAction {

    private final String inputPath;
    private final String outputPath;

    public ImageProcessingTask(String inputPath, String outputPath) {
        this.inputPath = inputPath;
        this.outputPath = outputPath;
    }

    @Override
    protected void compute() {
        try {
            ImageProcessor.processAndSave(inputPath, outputPath);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The processFolderParallel function is designed to process a list of image files in parallel using multiple threads. It utilizes the Fork/Join framework by defining a custom ImageProcessingTask class that extends RecursiveAction. Each task processes a single image by calling ImageProcessor.processAndSave(inputPath, outputPath), allowing concurrent image processing for improved performance. The function takes an array of input files, the output directory path, and the number of threads to use for parallel execution.

7 Benchmarking Utility – Measure Function

```
public class Benchmark {
    public static void measure(Runnable task, String label) {
        long start = System.nanoTime();
        task.run();
        long end = System.nanoTime();
        double elapsedMs = (end - start) / 1_000_000.0;
        System.out.printf("%s took %.2f ms%n", label, elapsedMs);
    }
}
```

The measure function in the Benchmark class is a utility method for measuring and reporting the execution time of a task. It accepts a Runnable task and a label as parameters. The function records the start and end time in nanoseconds, calculates the elapsed time in milliseconds, and prints the result with a custom label. This is useful for performance testing and comparing execution durations of different code blocks.

8 Main

```
import java.io.File;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;

public class Main {

    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
            System.out.println("Usage: java Main <sequential|parallel|
                forkjoin>");
            return;
        }

        String mode = args[0];
        String inputDir = "images";
        String outputDir = "output";

        File inputFolder = new File(inputDir);
        File[] files = inputFolder.listFiles((dir, name) -> name.endsWith(
            ".jpeg") || name.endsWith(".jpg"));

        if (files == null || files.length == 0) {
            System.out.println("No images found in folder.");
            return;
        }

        switch (mode.toLowerCase()) {
            case "sequential":
                Benchmark.measure(() -> {
                    for (File file : files) {
                        String inPath = file.getPath();
                        String outPath = outputDir + "/" + file.getName().
                            replaceFirst("\\.(jpeg|jpg)$", "_stylized.jpeg"
                            );
                        try {
                            ImageProcessor.processAndSave(inPath, outPath)
                                ;
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                }, "Sequential processing");
                break;

            case "parallel":
                Benchmark.measure(() -> {
                    var executor = java.util.concurrent.Executors.
                        newFixedThreadPool(Runtime.getRuntime().
                        availableProcessors());
                    List<java.util.concurrent.Future<?>> futures = new
                        ArrayList<>();

                    for (File file : files) {
                        String inPath = file.getPath();
                        String outPath = outputDir + "/" + file.getName().
                            replaceFirst("\\.(jpeg|jpg)$", "_stylized.jpeg"
                            );

                        futures.add(executor.submit(() -> {
                            try {
                                ImageProcessor.processAndSave(inPath,
                                    outPath);
                            } catch (Exception e) {
                                e.printStackTrace();
                            }
                        }));
                    }

                    // wait for all to finish
                    futures.forEach(f -> {
                        try {
                            f.get();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    });
                    executor.shutdown();
                }, "Parallel processing (ExecutorService)");
                break;

            case "forkjoin":
                Benchmark.measure(() -> {
                    ForkJoinPool pool = new ForkJoinPool(Runtime.
                        getRuntime().availableProcessors());
                    List<ImageProcessingTask> tasks = new ArrayList<>();

                    for (File file : files) {
                        String inPath = file.getPath();
                        String outPath = outputDir + "/" + file.getName().
                            replaceFirst("\\.(jpeg|jpg)$", "_stylized.jpeg"
                            );
                        tasks.add(new ImageProcessingTask(inPath, outPath)
                            );
                    }
                    for (ImageProcessingTask task : tasks) {
                        pool.execute(task);  // asynchronously start each
                            task
                    }

                    for (ImageProcessingTask task : tasks) {
                        task.join();        // wait for each task to
                            finish
                    }

                    pool.shutdown();
                }, "Parallel processing (ForkJoinPool)");
                break;

            default:
                System.out.println("Unknown mode: " + mode);
                System.out.println("Use: sequential, parallel, or forkjoin
                    ");
        }
    }
}
```

The main method serves as the entry point of the application, allowing users to choose between three modes of image processing: sequential, parallel, or forkjoin. Based on the command-line argument provided, the program reads all .jpeg and .jpg files from the images directory and processes them into the output directory using the selected strategy:

- Sequential: Processes images one at a time using a single thread.
- Parallel (ExecutorService): Uses a fixed thread pool to process images concurrently.
- ForkJoin: Utilizes the Fork/Join framework to recursively process images in parallel tasks.

Execution time for each mode is measured using the Benchmark.measure utility, providing performance feedback in milliseconds. This design allows users to easily compare the efficiency of different parallelization approaches for batch image processing.

9 Results and Analysis

This section presents a performance comparison of the three execution modes: Sequential, Parallel, and Fork/Join, based on CPU usage, total processing time, and memory usage.

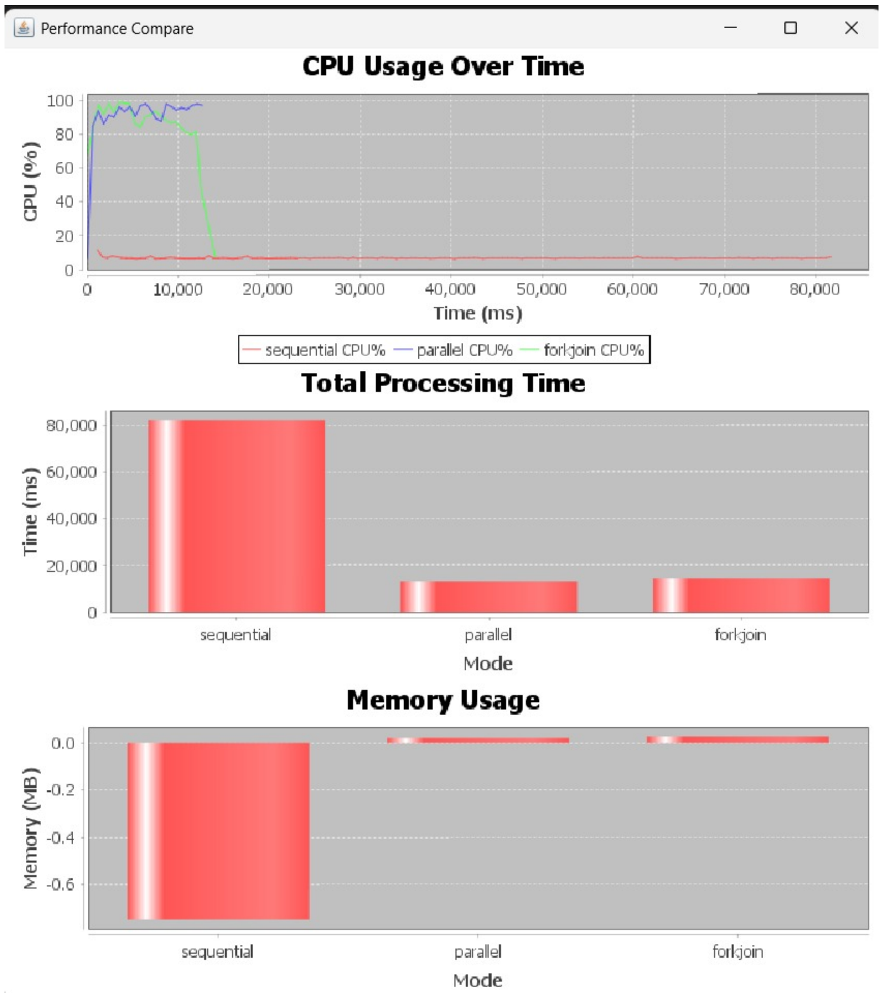

Figure 1: Results Graphs

- CPU Utilization: Parallel and Fork/Join modes demonstrate significantly better CPU utilization, leveraging available cores to maximize performance, whereas the sequential mode fails to take advantage of multi-core capabilities.
- Processing Time: Parallel and Fork/Join methods reduce processing time by approximately 80% compared to the sequential approach. The minimal difference between parallel and fork/join suggests similar levels of efficiency in terms of execution speed.
- Memory Usage: Memory consumption is not significantly impacted by the choice of execution mode in this test. The slight negative value for the sequential method may be due to memory de-allocation being recorded at the measurement point.

10 Conclusion

In this project, we successfully implemented a parallelized approach to real-time neural style transfer on 4K video frames using Java. Starting from a sequential baseline, we introduced two layers of concurrency: intra-frame parallelism using Fork/Join tiling, and inter-frame parallelism through multi-threaded processing. Thistwo level parallel strategy significantly reduced execution time and improved CPU utilization, demonstrating the effectiveness of Java's concurrency tools for high resolution image processing.
Our custom Van Gogh-inspired stylization filter provided a visually compelling example of artistic transformation, and we were able to maintain reasonable memory overhead while achieving measurable performance gains. The experimental results indicated that the system scaled efficiently with an increasing number of threads, validating our choice of parallel architecture.