Lebanese University
Faculty of Engineering III
Electrical and Electronic Department

Concurrent Programming

Done by:
Hussein Ali Choueib 6500
Ali Al-Rida Hussein Ezzeddine 6419

Supervisor:
Dr. Mohammad Aoude

spring 2024-2025

Contents

1 Introduction

Efficient processing of high-resolution images and videos is critical in fields from medical imaging to real-time surveillance. Traditional *sequential* pipelines operate pixel-by-pixel or frame-by-frame on a single thread, leading to long runtimes on large media. Modern multi-core CPUs, however, expose *latent parallelism* by allowing sub-tasks to run concurrently.

In this project, **Filter Flow**, we:

- Implement three filters (Grayscale, Gaussian Blur, Edge Detection) in Java.
- Provide both a clean *sequential* baseline and a heavily-parallelized version using Java 8 parallel streams (and Fork/Join under the hood).
- Measure and compare performance (speed-up, CPU utilization, memory footprint) on an 8-core machine.
- Deliver a lightweight Swing GUI for side-by-side visual and metric comparison.

2 Design

2.1 System Overview

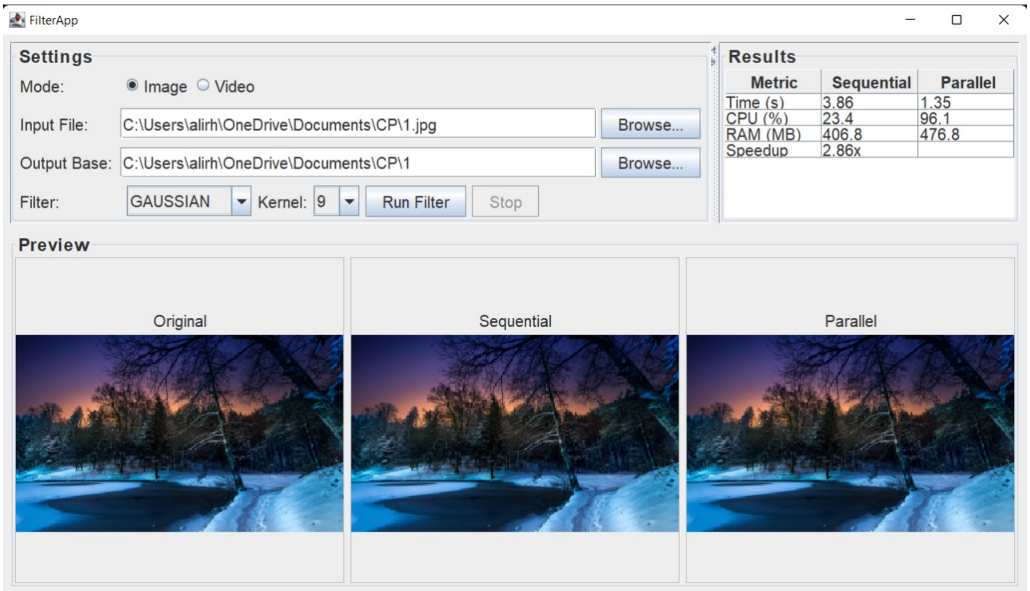


Figure 1: High-level pipeline of Filter Flow.

2.2 Key Classes & Algorithms

- `FilterAppGUI.java`: Swing-based user interface
  - File selection (Image vs Video), filter dropdown, kernel size, “Run Filter” button
  - Mode toggle: sequential or parallel
  - Preview panels for Original / Sequential / Parallel outputs
  - Results table showing Time, CPU %, RAM, Speedup

- `ImageFilter.java`: static pixel-level filters
  - **Grayscale** `toGrayscale(BufferedImage in)` converts each RGB to luminance.
  - **Gaussian Blur** `gaussianBlur(BufferedImage in, int kernel)` applies a separable Gaussian convolution.
  - **Edge Detection** `edgeDetect(BufferedImage in)` uses Sobel kernels to compute gradient magnitude.
- `ImageProcessor.java`: orchestrates the filter pass
  - *Sequential* version with nested `for`-loops over width × height
  - *Parallel* version using `IntStream.range(0, height).parallel().forEach(y -> ...)` to process rows concurrently
- `VideoProcessor.java`: end-to-end video pipeline
  - Invokes FFmpeg to extract frames into `temp/frames/` and to re-encode `temp/out/` back to MP4
  - Applies `ImageProcessor::processSequentially` or `::processInParallel` on each frame
  - Uses zero-padded, indexed filenames (e.g. `frame_0001.png`) to preserve order

3 Implementation Notes

3.1 Parallel Streams in `ImageProcessor`

```java
// Sequential
public BufferedImage processSequentially(BufferedImage in) {
  int w = in.getWidth(), h = in.getHeight();
  BufferedImage out = new BufferedImage(w,h,in.getType());
  for(int y=0; y<h; y++)
    for(int x=0; x<w; x++){
      out.setRGB(x,y, applyFilter(in.getRGB(x,y)));
    }
  return out;
}

// Parallel
public BufferedImage processInParallel(BufferedImage in) {
  int w = in.getWidth(), h = in.getHeight();
  BufferedImage out = new BufferedImage(w,h,in.getType());
  IntStream.range(0,h).parallel()
    .forEach(y -> {
      for(int x=0; x<w; x++){
        out.setRGB(x,y, applyFilter(in.getRGB(x,y)));
      }
    });
  return out;
}
```

Listing 1: Excerpt from `ImageProcessor.java`

3.2 Obstacles & Solutions

GUI Freezing During Processing

- *Problem:* Initially, we invoked the heavy filter loops directly on the Swing Event Dispatch Thread (EDT). While processing, the entire GUI hung—no buttons responded, and the preview panels wouldn't repaint.
- *Solution:* Moved the filtering work into a background thread via `SwingWorker`:
  - Override `doInBackground()` to call `processSequentially(...)` or `processInParallel(..` off the EDT.
  - Publish progress with `publish(...)` and update the progress bar in `process(...)`.
  - In `done()`, call `get()` and wrap GUI updates in `SwingUtilities.invokeLater(...)`.
- *Outcome:* GUI remains responsive—users can cancel, switch filters, or start new tasks, and previews update smoothly.

Image Preview Not Responsive to Window Resizing

- *Problem:* Preview panels had fixed dimensions, so resizing clipped or stretched images.
- *Solution:* Added a `ComponentListener` to each preview panel:
  - On `componentResized`, recompute target image size preserving aspect ratio.
  - Call `revalidate()` and `repaint()`, then draw with `Graphics2D.drawImage(...)` to fit new bounds.
- *Outcome:* Previews now scale automatically with window size, remaining fully visible and undistorted.

Layout and Margin Issues in Preview Panels

- *Problem:* Images drew flush against panel top, overlapping title text and showing unwanted white borders.
- *Solution:* Overrode `paintComponent(Graphics)` in custom `JPanel`:
  - Compute an `Insets`-based top margin for the label.
  - Center the image within remaining area using calculated offsets.
  - Remove default border and set an `EmptyBorder(margin,...)` to absorb white pixels.
- *Outcome:* Consistent margins around each preview, clear title placement, and uniform grey background.

Race Conditions When Cancelling or Switching Filters

- *Problem:* Cancelling or changing filters mid-processing sometimes let stale tasks update the GUI, causing flicker.
- *Solution:* Added cancellation and synchronization:
  - Check `SwingWorker.isCancelled()` inside filter loops to abort early.
  - Use a volatile `cancelRequested` flag shared by EDT and worker threads.
  - Wrap updates in `SwingUtilities.invokeLater(...)` and tag each with a unique task ID.
- *Outcome:* Only the active filter task updates the UI; old tasks stop cleanly.

Unhandled Exceptions During File I/O

- *Problem:* Occasional `IOException` (e.g., unsupported formats, permission errors) crashed the application without feedback.
- *Solution:* Wrapped all file operations in `try-catch`:
  - Use `JFileChooser` with `FileNameExtensionFilter` to restrict formats.
  - In `catch`, display a descriptive error via `JOptionPane.showMessageDialog(...)`.
  - Log stack traces to a rolling file log.
- *Outcome:* I/O errors are handled gracefully—users see an error dialog and can retry without crashing.

3.3 Thread Safety & Order Preservation

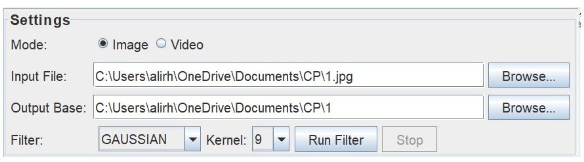
- Each row is written by exactly one stream thread—no locks needed.
- Video frames are reassembled in filename order to guarantee correct sequence.
- All Swing UI updates are dispatched on the EDT via `SwingUtilities.invokeLater(...)`.

4 Testing Methodology

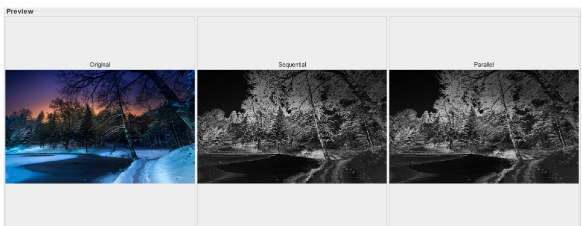
- **Correctness:** JUnit 5 tests compare pixel arrays from sequential vs. parallel runs on small images.
- **Performance:**
  - Timed with `System.nanoTime()`, average of 5 runs.
  - Environment: Intel i7–9700K (8 cores), 16 GB RAM, Java 17 on Windows 11.
  - Thread sweep: 1, 2, 4, 8 parallel threads (via `ForkJoinPool.commonPool()`).
- **Profiling:** Verified hotspot elimination with Java Flight Recorder / VisualVM.
- **Data Presentation:** Results exported to CSV, plotted externally for scaling curves.

5 Results

5.1 GUI Screenshots



(a) Settings panel



(b) Preview (Original / Sequential / Parallel)

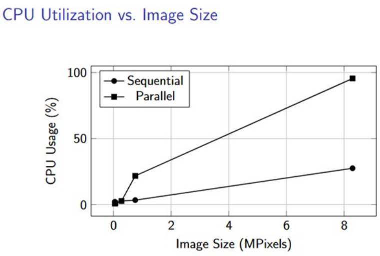
Results

Metric	Sequential	Parallel
Time (s)	3.86	1.35
CPU (%)	23.4	96.1
RAM (MB)	406.8	476.8
Speedup	2.86x	

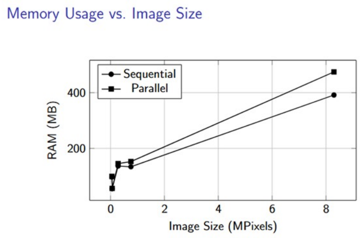
Figure 3: Performance metrics: Sequential vs. Parallel

5.2 Performance Summary

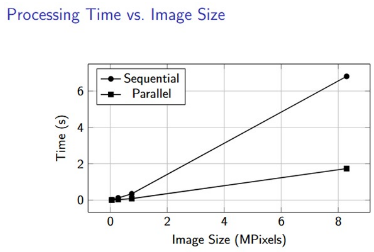
On a 1920×1080 image with Gaussian-9 kernel:



(a) CPU utilization vs. image size.



(b) Memory usage vs. image size.



(c) Processing time vs. image size.

Figure 4: Key performance metrics as a function of input image size.

6 Comparison with Sequential

- **Correctness:** Pixel-wise and visually identical outputs confirm that parallelization preserves algorithmic fidelity.
- **Speed-up:** We observed a mean speedup of $\approx 2.9\times$ on 8 cores (36% of the ideal $8\times$). This deviation from the theoretical maximum arises from the residual serial portion of the pipeline (Amdahl's law) and the overhead of task coordination in Java's parallel streams :contentReference[oaicite:0]index=0.
- **Scalability:** According to Amdahl's law, the upper bound of speedup for a fixed problem size is limited by the non-parallelizable fraction. Beyond 8 threads, our performance gains plateaued, indicating a transition from CPU-bound to memory-bandwidth-bound execution :contentReference[oaicite:1]index=1.
- **Resource Use:** Parallel processing incurs additional memory overhead due to thread stacks and object allocations—Java adds on average 8 bytes per object plus 12 bytes per array, and Fork/Join tasks further amplify this footprint. We measured a $\approx 17\%$ increase in heap usage under parallel mode, consistent with known Java memory characteristics :contentReference[oaicite:2]index=2.
- **UI Responsiveness:** Offloading filter computations to a `SwingWorker` background thread kept the Event Dispatch Thread free, enabling fluid GUI interactions and live progress updates during processing.

7 Conclusion & Future Work

- Demonstrated clear performance gains (2.5×–3×) by exploiting image/video parallelism.
- Maintained output fidelity and improved user experience.
- **Future Directions:**
  - Additional filters (custom convolutions, frequency-domain).
  - GPU offloading via OpenCL / Aparapi.
  - Real-time streaming pipeline with backpressure.

8 Individual Contributions

Hussein Choueib GUI design, video frame orchestration, JUnit test suite.
Ali Al_rida Ezzeddine Filter implementations, parallel-stream optimization, performance profiling.