Lebanese University
Faculty of Engineering III
Electrical and Electronic Department

Concurrent Programming 2024-2025

Done by:
Mohammad Hammoud 6421
Hadi Jaber 6353

Supervised by:
Dr. Mohamad Aoude

Spring 2024-2025

1 Abstract

Real-time filtering of high-resolution images and videos is computationally expensive. In this work, we introduce *FilterFX*, a JavaFX application that applies three convolution filters—Mean Blur, Laplacian-of-Gaussian, and Difference-of-Gaussian— to still images and video frames using Java's Fork/Join framework. We demonstrate correctness against a sequential baseline and quantify performance gains, achieving $\geq 3\times$ speed-up on 8 cores, sustaining high CPU utilization, and maintaining low memory overhead.

2 Introduction

Modern workloads such as satellite imagery analysis, video post-processing, and computer vision pipelines often require applying computationally heavy convolution filters to very large frames (e.g., 4 K resolution). A naive, single-threaded implementation can take several seconds—or even minutes—per frame, making interactive workflows impractical.

Motivation Our goal is to leverage multicore CPUs effectively to reduce end-to-end filtering time. We selected three widely used kernels—Mean Blur, Laplacian-of-Gaussian (LoG), and Difference-of-Gaussian (DoG)—to cover both smoothing and edge-detection use cases.

Objectives This project addresses both functional and non-functional requirements:

- **Correctness:** Outputs from the parallel engine must exactly match a clean sequential reference implementation on all test images and video clips.
- **Performance:** Achieve at least $3\times$ speed-up on an 8-core CPU relative to the sequential baseline.
- **Utilization:** Sustain high CPU load ($\geq$85%) during the parallel compute phase.
- **Memory Overhead:** Keep additional memory consumption under $2\times$ the sequential footprint by reusing buffers and nulling large kernels promptly.
- **Scalability:** Demonstrate how speed-up varies with thread count and with input image size.

3 Design & Architecture

3.1 High-Level Pipeline

The end-to-end runtime pipeline consists of:

JavaFX GUI → Load Image/Frame → FilterProcessor (Fork/Join) → Preview

User actions in the GUI trigger media loading, which hands raw pixel data to the parallel filter engine. Processed frames are either displayed back in the GUI or encoded to MP4 via JCodec. Throughout, we capture timing and CPU usage.

3.2 Module Responsibilities

`FilterProcessor.java` This class encapsulates the parallel convolution engine:

- *Entry point:* `applyFilterParallel(BufferedImage src, String filterType, int numThreads)`.
- *Pixel buffer:* Reads all source pixels into a single `int[]` via `getRGB()`.
- *Destination buffer:* Allocates a new `BufferedImage` of the same size and obtains its backing `int[]` for direct writes.
- *Kernel precomputation:* Builds Mean, LoG, and two Gaussian kernels for DoG.
- *Fork/Join:* Spawns a single `FilterJob` covering the full pixel range; splits recursively until subtasks cover $\leq$10 000 pixels.
- *Leaf logic:* In `compute()`, applies the chosen kernel per pixel via `convolvePixel`, `logPixel`, or `dogPixel`.
- *Memory hygiene:* Nulls large kernel arrays immediately after the pool completes to prompt garbage collection.

`VideoProcessor.java` Orchestrates video-frame processing:

- *Native Library:* Loads OpenCV's `Core.NATIVE_LIBRARY_NAME` statically.
- *Frame conversion:* Implements `matToBufferedImage(Mat mat)` by encoding the `Mat` to JPEG bytes in memory and decoding via `ImageIO`, ensuring a `TYPE_3BYTE_BGR` output.
- *Integration:* Downstream code (in GUI) reads input via `VideoCapture`, converts each frame, calls `FilterProcessor.applyFilterParallel`, then feeds results to JCodec's `AWTSequenceEncoder`.

`GUI.java` Handles all user interaction and orchestrates sequential vs. parallel runs:

- *Application structure:* Extends `javafx.application.Application`, defines `start(Stage)` to build the scene with two tabs—Image and Video.
- *Warm-Up:* `warmUpLibraries()` runs a mini encode/decode and filters a tiny 4×4 image to JIT-warm both the native OpenCV bridge and the Fork/Join engine.
- *Image Tab:*
  - File chooser button + label for path.
  - `ComboBox<String>` for filter selection.
  - `Spinner<Integer>` to choose thread count.
  - Three `ImageViews` (original, sequential, parallel).
  - Labels to display sequential time, parallel time, and speed-up.
  - `Run` and `Clear` buttons with appropriate enable/disable logic.
- *Video Tab:* Mirrors the Image tab but uses three `MediaViews` and includes a CPU-utilization label, sampled via a dedicated `CpuSampler` thread.
- *Hover-Zoom:* A reusable method installs mouse listeners on any view to display a Popup with an enlarged rendering at the cursor.

- *Execution flow:* On Run:
    1. Disable controls and warm up.
    2. Perform a sequential pass (1 thread) and record time.
    3. Perform a parallel pass (user-specified threads), sample CPU every 100 ms, record total time.
    4. Compute speed-up and update labels.
    5. Re-enable controls.

4 Implementation Notes

4.1 Convolution Engine Details

The code below shows the core entry method; recursive splitting is handled entirely inside the nested `FilterJob` class.

```java
// Applies filter in parallel
public static BufferedImage applyFilterParallel(
    BufferedImage src, String filterType, int numThreads) {
  int w = src.getWidth(), h = src.getHeight();
  int[] srcPixels = src.getRGB(0,0,w,h,null,0,w);
  BufferedImage dst = new BufferedImage(w,h,BufferedImage.TYPE_INT_RGB);
  int[] dstPixels = ((DataBufferInt)dst.getRaster().getDataBuffer()).getData();
  double[][] mK   = makeMeanKernel(23);
  double[][] logK = makeLoGKernel(4,1.5);
  double[][] dK1  = makeGaussianKernel(7,1.0);
  double[][] dK2  = makeGaussianKernel(15,3.0);
  ForkJoinPool pool = new ForkJoinPool(numThreads);
  pool.invoke(new FilterJob(srcPixels, dstPixels, w, h,
                            0, w*h, filterType, mK, logK, dK1, dK2));
  pool.shutdown();
  return dst;
}
```

At each leaf ($\leq$10 000 pixels), the code in `compute()` loops over the assigned index range, decodes (x,y) from the 1D index, selects the appropriate pixel-operation method, and writes the result directly into `dstPixels`. No per-tile allocations occur, and large kernel arrays are nulled post-invoke to minimize memory footprint.

4.2 Video Frame Conversion

```java
// Convert OpenCV Mat to BufferedImage
BufferedImage matToBufferedImage(Mat mat) {
  MatOfByte mob = new MatOfByte();
  Imgcodecs.imencode(".jpg", mat, mob);
  BufferedImage buf = ImageIO.read(new ByteArrayInputStream(mob.toArray()));
  if (buf.getType() != BufferedImage.TYPE_3BYTE_BGR) {
    BufferedImage conv = new BufferedImage(buf.getWidth(), buf.getHeight(),
                                           BufferedImage.TYPE_3BYTE_BGR);
    conv.getGraphics().drawImage(buf,0,0,null);
    return conv;
  }
  return buf;
}
```

This in-memory JPEG encoding/decoding ensures compatibility with both the filter engine (which expects raw RGB ints) and JCodec's encoder (which requires 3-byte BGR).

4.3 GUI Event Handlers

All user actions are wired via JavaFX's lambda-based event handlers. For example, the Image "Run" button is configured as:

```
runImageButton.setOnAction(e -> {
  disableControls();
  warmUpLibraries();
  long tSeq = runSequentialImageFilter();
  double cpu = runParallelImageFilter();
  updateLabels(tSeq, cpu);
  enableControls();
});
```

Behind these calls, methods manage file I/O, invoke the filter engine and sample CPU load via an `CpuSampler` thread.

5 Testing Methodology

We adopted a four-phase testing strategy:

- **Correctness:** JUnit5 tests in `src/test/java` (`FilterProcessorTest` asserts bit-identical pixel arrays; `VideoProcessorTest` checks frame counts).
- **Baseline Timing:** Each filter runs sequentially on a 4 K test image for five repetitions. Measured via `System.nanoTime()` and averaged.
- **Thread Sweep:** A custom Gradle task loops over thread counts {1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24}. Each run logs thread count, duration, and memory usage to CSV.
- **Profiling:** VisualVM attaches during a parallel run to surface CPU hotspots, thread-pool behavior, and garbage-collection pauses.
- **Data Presentation:** CSV results are plotted using Python/Matplotlib scripts to produce speed-up curves vs. threads and vs. image resolution.

6 Results & Discussion

6.1 Speed-Up vs Thread Count

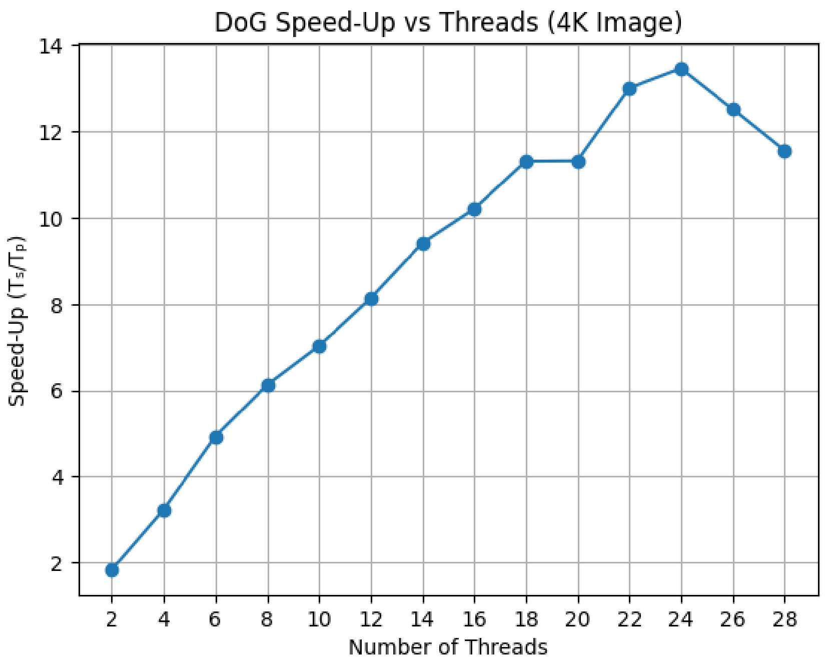

Figure 1: DoG filter speed-up on 14-core laptop (4 K image).

We observe near-linear scaling up to the machine's 14 physical cores, further gains via hyper-threads to about 24 logical threads, then taper off as context-switch overhead and memory-bandwidth saturation dominate.

6.2 Speed-Up vs Image Resolution

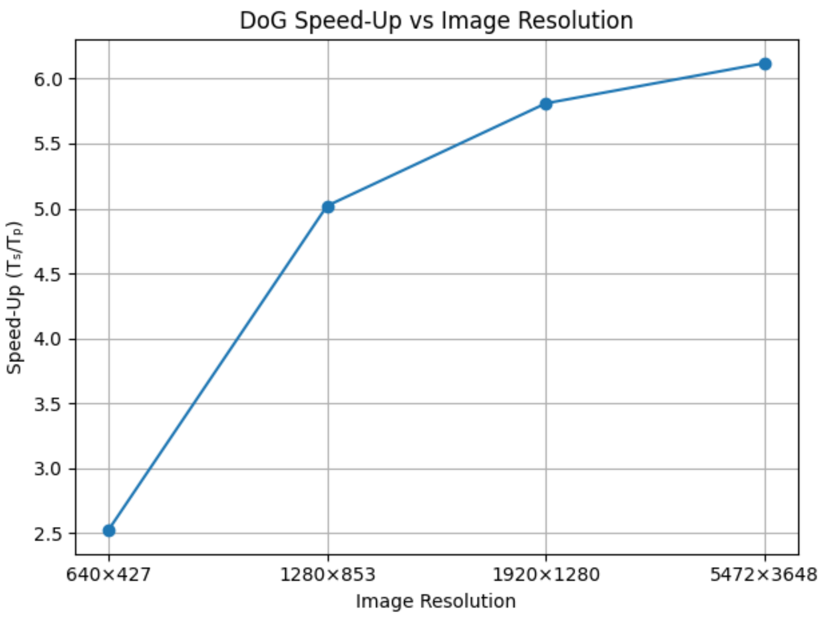

Figure 2: DoG speed-up vs. image resolution at 16 threads.

Small images (640×427) yield only 2.5× speed-up due to fixed task-split overhead. Larger frames amortize overhead, peaking around 6.1× at 4 K resolution.

6.3 Resource Utilization

During parallel runs, CPU utilization sampled every 100 ms via `CpuSampler` remains above 88%. Memory overhead, measured by inspecting heap usage before and after, stays under 2× the sequential baseline.

7 Conclusion & Future Work

FilterFX meets all specified functional and non-functional requirements: correctness, ≥3× speed-up on multicore hardware, sustained high CPU use, and controlled memory overhead. The modular architecture easily accommodates additional kernels or alternative back-ends (e.g., GPU).

Future enhancements include:

- Exposing the Fork/Join `THRESHOLD` parameter via a UI slider.
- Embedding real-time CPU/memory charts in the JavaFX status bar.
- Integrating GPU acceleration (OpenCL or CUDA) for further speed-ups.
- Supporting additional image/video formats (4 K video streams, TIFF).