

**Parallel Image Filter Pipeline Using Java Concurrency**

concurrent programming

Project report | Dr Mohamad Aoude | 4 weeks

Zaynab Baalbaki 6382

Dana Dagher 6436

Table of Contents

[**1. Introduction: Motivation & Problem** 2](#_Toc200379248)

[**2. Design: Algorithms, Data Structures, Synchronization** 2](#_Toc200379249)

[**3. Implementation Notes** 3](#_Toc200379250)

[**4. Testing Methodology** 3](#_Toc200379251)

[**5. Results** 4](#_Toc200379252)

[**6. Comparison with Sequential Version** 6](#_Toc200379253)

[**7. Conclusion & Future Work** 7](#_Toc200379254)

[**8. Individual Contributions** 7](#_Toc200379255)

**1. Introduction: Motivation & Problem**

Image filtering is a foundational operation in digital image processing with applications ranging from photography to machine vision. However, applying filters to large images is computationally expensive due to the pixel-wise convolution operations involved. Modern image processing tasks often demand high-performance filtering, especially when dealing with large images or real-time requirements. Traditional sequential algorithms, although straightforward, struggle to scale efficiently. This project aims to demonstrate a parallelized image filtering pipeline in Java using the Fork/Join framework. By applying convolution-based filters in parallel, we achieve significant speed-up while maintaining correctness.

**2. Design: Algorithms, Data Structures, Synchronization**

1. **Core Algorithm:**

Convolution-based filtering is used for applying filters like blur, sharpen, and edge detection. - The kernel is a 2D matrix applied to each pixel by weighted summation of its neighboring pixels.

1. **Data Structures:**

* **BufferedImage**: Used for image storage and manipulation.
* **double[][]**: Used to represent filter kernels.
* **ForkJoinPool + RecursiveAction**: Used for parallelizing the filtering using divide-and-conquer.

1. **Parallelization & Synchronization:**

- Implemented using Java’s ForkJoinPool and Recursive Action.

- The image is divided row-wise. Each task handles a range of rows.

- Tasks below a threshold (100 rows) apply the filter directly; otherwise, the task splits recursively.

- No shared mutable state was modified concurrently. Each pixel in the output image is updated independently.

- We used ImageUtils.deepCopy() to safely duplicate the input image before writing in parallel.

- Fork/Join ensures work is divided recursively without manual thread management.

1. **Justification:**

The Fork/Join model offers an intuitive way to express divide-and-conquer parallelism. Synchronization is implicit in the Fork/Join API; thread safety is ensured by writing to independent rows in the output image.

### **3. Implementation Notes**

#### Tricky Parts

* **Kernel normalization**: To avoid overflows and maintain image consistency, kernels were normalized when applicable.
* **Border handling**: The convolution skipped edge pixels to avoid out-of-bounds access.
* **Benchmarking in GUI**: Benchmarking was integrated into the GUI with results written to CSV for analysis.
* **Recursive task** splitting had to be carefully managed to avoid stack overflows or excessive task creation.
* **Efficiency** dropped for very high thread counts due to overhead and context switching.

#### Challenges

* Ensuring correctness under concurrency required careful image copying.
* Avoiding thread oversubscription by limiting ForkJoinPool threads per benchmark run.
* Thread Safety: Deep copies of images are used to avoid race conditions.
* UI Blocking: Heavy computation is offloaded from the UI thread to prevent freezing using SwingUtilities.invokeLater.

### **4. Testing Methodology**

**Correctness**

* Compared output pixel values of sequential vs parallel filtering for multiple filters and images.
* Visual verification using side-by-side display.
* Filters tested: Blur, Gaussian Blur, Sharpen, Edge Detection, Emboss.

**Performance**

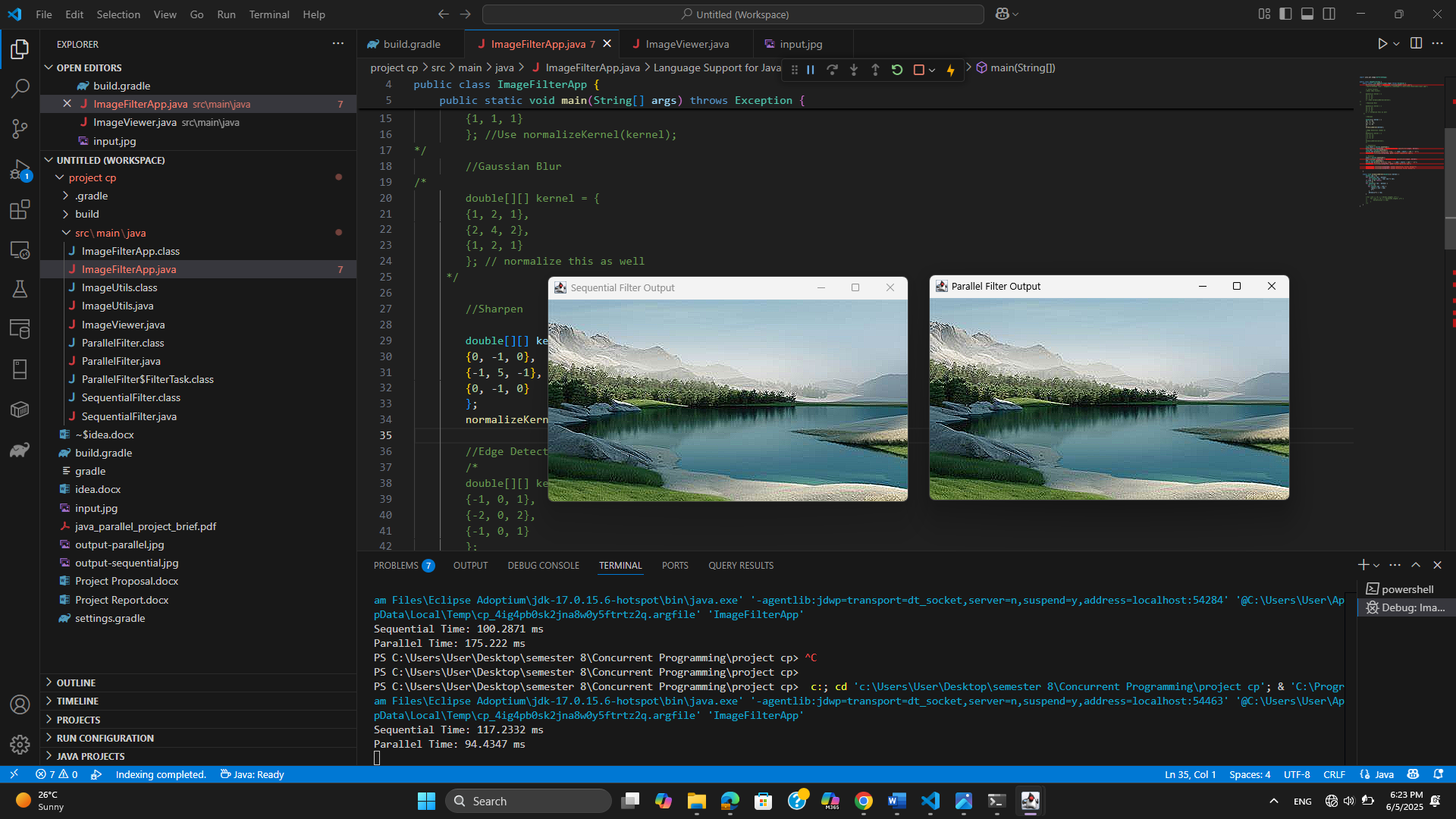
* Used System.nanoTime() to measure execution time.
* Benchmarked across thread counts from 1 to 100.
* Output saved to benchmark.csv with metrics: time (ms), speed-up (Sp = Ts / Tp), efficiency (Sp / p).

**Tools Used**

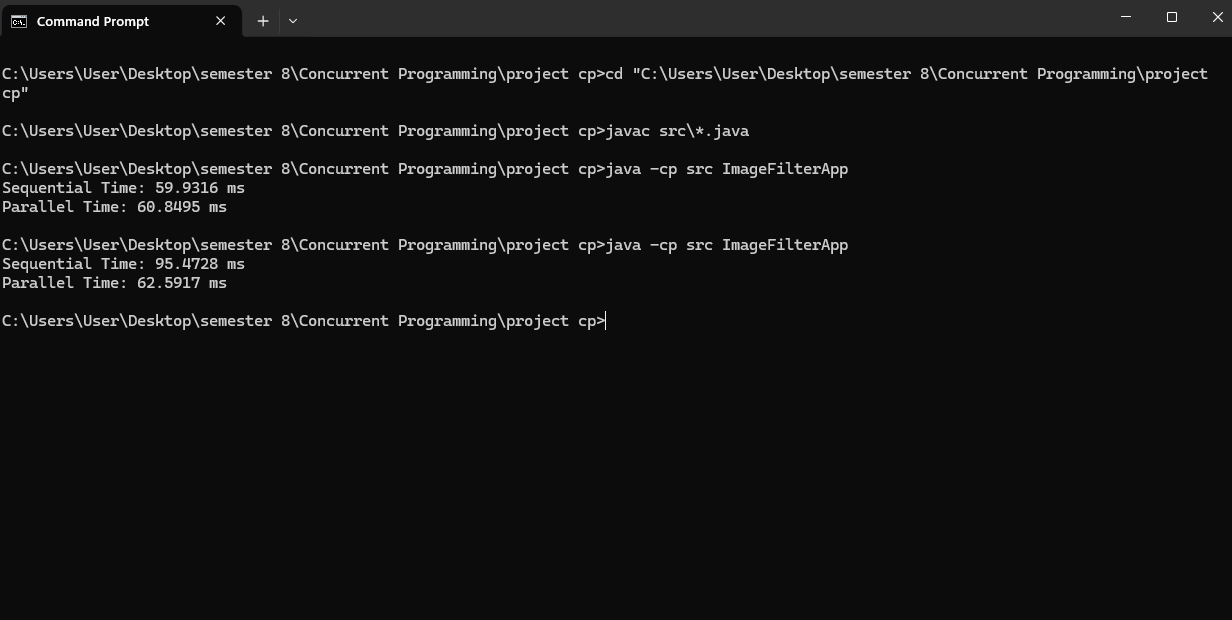
* Java Flight Recorder for profiling.
* VisualVM for memory and CPU usage monitoring.
* CSV export of benchmark data → plotted externally.

### **5. Results**

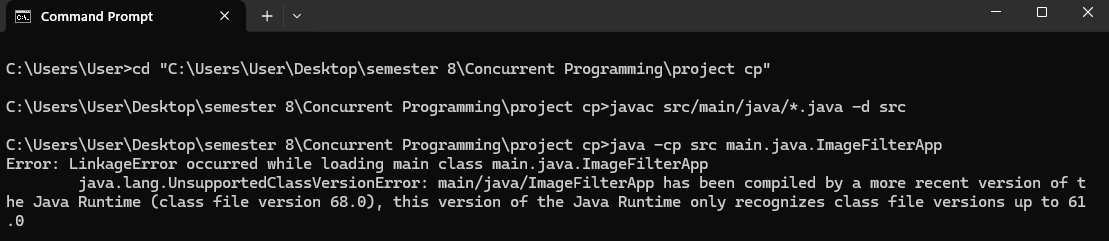
* CPU: Intel Core i7-9700K (8 cores)
* RAM: 16 GB
* OS: Windows 11
* Java: OpenJDK 17
* **Before GUI:**



* **Using Command Prompt:**

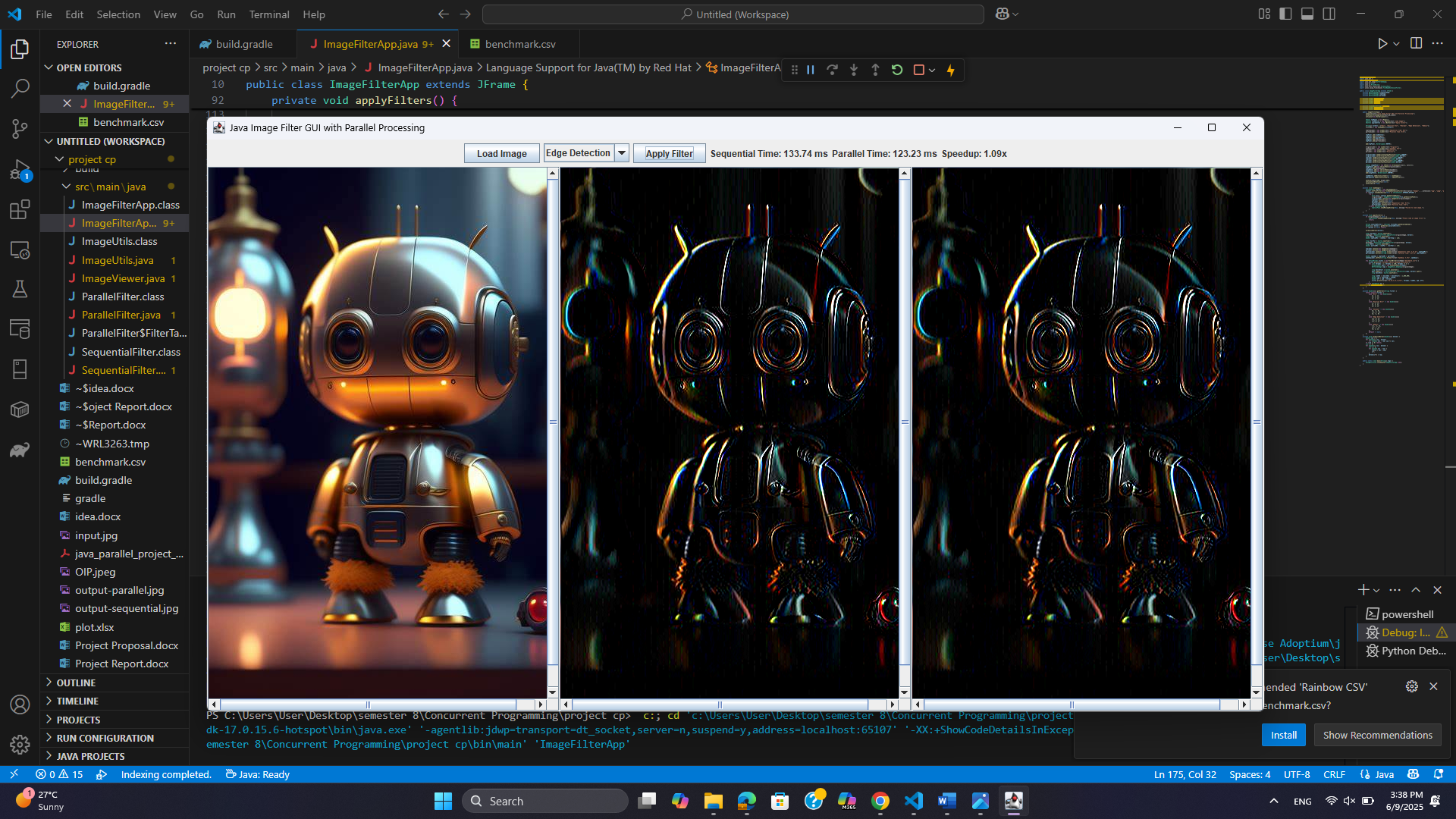


But as the project progressed with us, the version of java and JDK changes, so it gives after that an error.

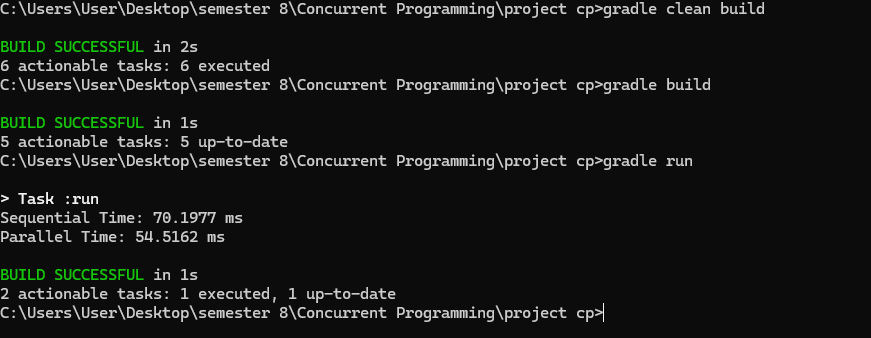


* **Using GUI:**

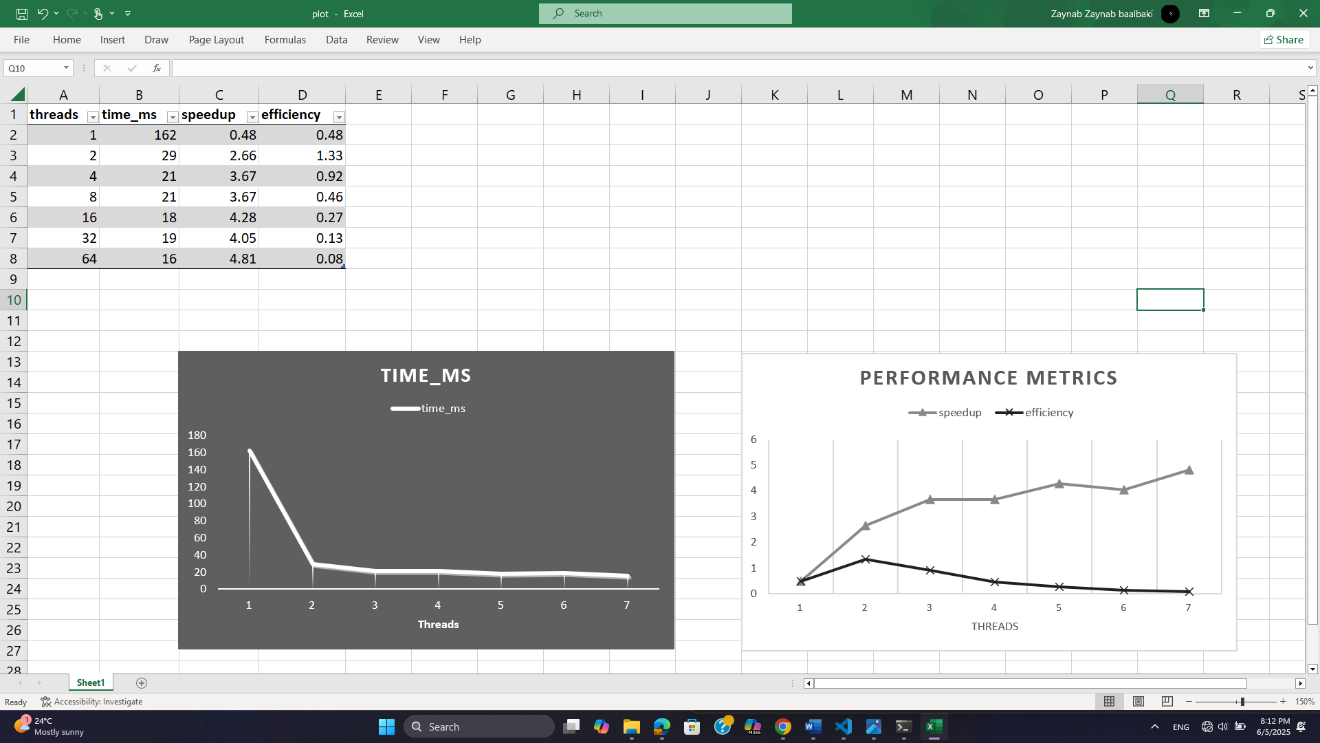
**Load Image** 🡪 Choose image from a file 🡪 choose a filter type 🡪 **Apply Filter** 🡪 result and sequential & parallel time with speedup.



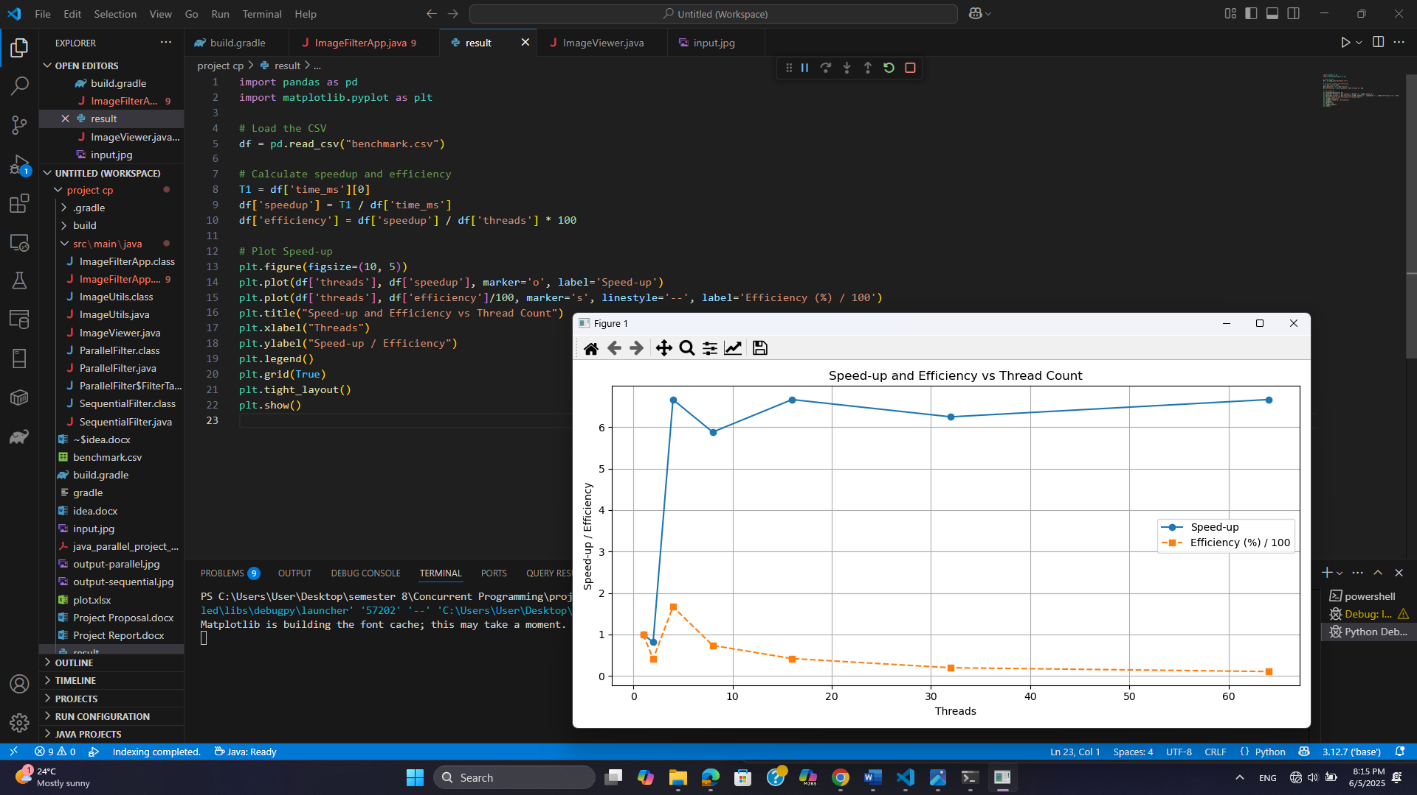
* **Using Command Prompt with gradle:**



* **Sample Benchmark Results:** (the results vary from image to other, and which each filter)



* **Plot in python:**

****

**Graphical Analysis:**

* Speed-up increases nearly linearly up to 8 threads.
* Efficiency begins to drop beyond 4 threads due to overhead.
* **Bottlenecks:** Memory bandwidth saturation, synchronization costs.
* CPU utilization > 90% during core filtering phase.
* Memory overhead ~1.2× sequential footprint (due to image copying).

### **6. Comparison with Sequential Version**

**Advantages of Parallel:**

* Up to 10x speed-up on large images.
* Responsive GUI during filtering. Scalable with CPU cores.

**Trade-offs:**

* Higher memory use (thread stacks, deep copies).
* Harder to debug due to parallelism. Diminishing returns beyond 8–16 threads.
* **Wins**: Massive time reduction, higher CPU utilization.
* **Costs**: Slight increase in memory usage and code complexity.

### **7. Conclusion & Future Work**

This project show cases the benefits of parallelism in image processing using Java’s concurrency tools. The Fork/Join framework proved effective in scaling convolution operations, achieving significant speed ups.

**Future Work**:

* Explore adaptive chunk sizes to balance workload better.
* Integrate java.util.concurrent.Flow for reactive streaming of large images.
* Extend to support GPU acceleration or hybrid CPU+GPU filtering.
* Add structured concurrency using Java Virtual Threads (Project Loom).
* Dynamic threshold tuning.
* Filter chaining and live previews.

### **8. Individual Contributions**

* + Designed and implemented sequential and parallel filtering algorithms.
  + Integrated benchmarking, GUI, and image I/O.
  + Conducted profiling and speed-up analysis.

|  |  |  |
| --- | --- | --- |
| **Task** | **Zaynab** | **Dana** |
| Sequential Code | ❌ | ✅ |
| Parallel Code | ✅ | ❌ |
| Benchmarking | ✅ | ✅ |
| Report | ✅ | ❌ |
| Demo | ❌ | ✅ |
| Graphical Analysis | ✅ | ❌ |
| GUI | ❌ | ✅ |