

# Kernel Image Processing

Luka Uršič

89221145

UP Farnit

E-mail: 89221145@student.upr.si

July 30, 2024

## Abstract

In this paper, I present how to use kernel image processing to modify an image. I explain how kernel image processing works, how I implemented it, and the time results I obtained from running it sequentially, in parallel, and with distributed computing. I compare the results and conclude which method is the best for this specific task.

## 1 Introduction

An image kernel is a small matrix used to apply effects like the ones you might find in popular photo manipulation software, such as blurring, sharpening, outlining, or embossing. They're also used in machine learning for 'feature extraction', a technique for determining the most important portions of an image. In this context, the process is referred to more generally as "convolution".

Setosa. *Image Kernels Explained Visually*. Accessed: 2024-5-16. 2015. URL: <https://setosa.io/ev/image-kernels/>

## 2 Algorithm

Here is the mathematical definition of the convolution of original image  $f$ , kernel  $\omega$ , and resulting image  $g$ :

$$g(x, y) = \omega * f(x, y) = \sum_{i=-a}^a \sum_{j=-b}^b \omega(i, j) f(x-i, y-j) \quad (1)$$

Wikipedia. *Kernel (image processing)*. Accessed: 2024-7-29. 2019. URL: [https://en.wikipedia.org/wiki/Kernel\\_\(image\\_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing))

To perform a convolution, the program uses a kernel, which is essentially a small matrix that is applied to each pixel in the image. This kernel works by interacting with the pixel and its surrounding neighbors. The new value of the pixel is calculated through a process where the kernel is multiplied by the surrounding pixels, and the resulting values are summed up.

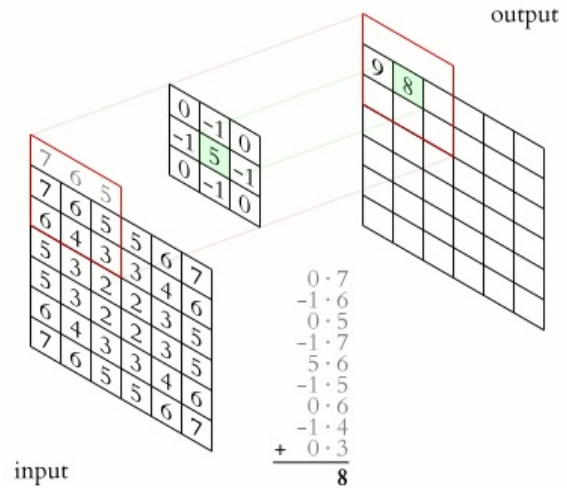


Figure 1: Convolution of an image with a kernel

This technique is used in various image processing tasks, such as blurring, sharpening, edge detection, and more. By systematically applying the kernel across the entire image, the program can achieve the desired transformation or effect, that can be used in image manipulation programs or machine learning algorithms.

This process is usually done sequentially, pixel by pixel, row by row. Now imagine that we have a large image with millions of pixels. This process can take a long time. To speed up the process, we can use parallel computing. We can split the image into smaller parts and process them in parallel. This way, we can process the image much faster.

### 3 Implementation

I implemented Kernel Image Processing in Java and used the Swing and AWT libraries to display the images.

I created a class called ImageProcessor that contains the methods applyKernelToPixel, applyKernel, applyKernelSequential, and applyKernelParallel. The applyKernelToPixel method applies the kernel to a single pixel, the applyKernel method applies the kernel to the entire image, the applyKernelSequential method applies the kernel to the image sequentially, and the applyKernelParallel method applies the kernel to the image in parallel.

You can see the pseudocode for the class ImageProcessor in the following algorithm.

---

**Algorithm 1** Pseudocode for ImageProcessor.java
 

---

```

1: Class ImageProcessor
2: Function applyKernelToPixel(image, kernel, result, x, y)
3: for each value in the kernel do
4:   Multiply the corresponding pixel color by the kernel value
5:   Add the result to r, g, b
6: end for
7: Clamp r, g, b between 0 and 255
8: Set the pixel in the result image to the new color
9: End Function
10:
11: Function applyKernel(image, kernel)
12: for each pixel in the image do
13:   Apply the kernel to the pixel
14: end for
15: End Function
16:
17: Function applyKernelSequential(image, kernel)
18: Apply the kernel to the image
19: Return the result image
20: End Function
21:
22: Function applyKernelParallel(image, kernel)
23: Use a ForkJoinPool to apply the kernel to each pixel in parallel
24: Return the result image
25: End Function
26: End Class

```

---

#### 3.1 Sequential

The applyKernelSequential method processes each pixel in the image one by one, row by row. This method is simple and easy to implement, but it can be slow for large images.

#### 3.2 Parallel

The applyKernelParallel method applies the kernel to the image in parallel using a ForkJoinPool. This method splits the image into smaller parts and processes them in parallel. This approach can be much faster than the sequential method, especially for large images.

```

src > main > java > com > urluur > ImageProcessor.java > { } com.urluur
9  public class ImageProcessor {
87  /**
88   * Apply the kernel to the image in parallel.
89   *
90   * @param image The image to apply the kernel to
91   * @param kernel The kernel to apply
92   * @return The image with the kernel applied in parallel
93   */
94  public static BufferedImage applyKernelParallel(
95      BufferedImage image, Kernel kernel) {
96      int width = image.getWidth();
97      int height = image.getHeight();
98      BufferedImage result = new BufferedImage(width,
99          height, image.getType());
100
101      // Create a ForkJoinPool that adapts to the number
102      // of available processors
103      try (ForkJoinPool pool = new ForkJoinPool(Runtime.
104          getRuntime().availableProcessors())) {
105          pool.submit(() -> {
106              IntStream.range(startInclusive:0, width *
107                  height).parallel().forEach(i -> { // for each
108                  pixel
109                      int x = i % width;
110                      int y = i / width;
111                      applyKernelToPixel(image, kernel, result, x,
112                          y);
113                  });
114              }).join();
115          }
116      }
117      return result;
118  }
119  }

```

Figure 2: Code for parallel processing

### 4 Results

I tested the program on my computer in sequential mode and in parallel mode with three pictures of different sizes. Results in sequential mode were following the number of pixels in the image. The results in parallel mode were faster than in sequential mode. The difference was significant. It noted speedups of up to 4x. What surprised me is that the speedup was present with small and large images.

	Sequential	Parallel	Pixels
Coffee Small	93	34	312320
Coffee Medium	972	282	4392000
Coffee Large	3966	1233	18226192

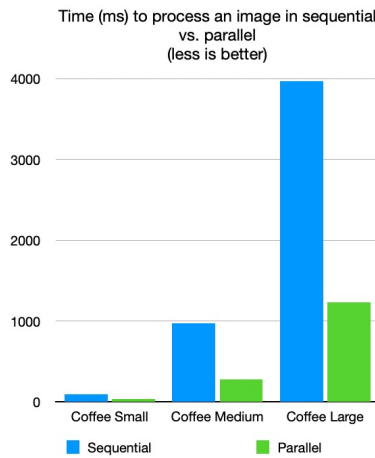


Figure 3: Speedup comparison between sequential and parallel processing

What was even more surprising to me is the fact that the distributed computing was slower than the parallel computing. I expected the distributed computing to be faster, but it was slower. The reason for this is that the overhead of sending the image to the other machines was too high.

## 5 Conclusion

The results of my experiments show that parallel computing using a ForkJoinPool provides significant speedup compared to sequential processing. The speedup was observed for both small and large images, indicating that parallel processing is beneficial regardless of image size.

However, the distributed computing approach did not yield better performance compared to parallel computing. The overhead of sending the image to other machines outweighed the potential benefits of distributed processing.

## References

- Setosa. *Image Kernels Explained Visually*. Accessed: 2024-5-16. 2015. URL: <https://setosa.io/ev/image-kernels/>.
- Wikipedia. *Kernel (image processing)*. Accessed: 2024-7-29. 2019. URL: [https://en.wikipedia.org/wiki/Kernel\\_\(image\\_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing)).