Kernel Image Processing

Luka Uršič
89221145
UP Famnit
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 number 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". [1]

2 Convolution

2.1 Definition

Here is the mathematical definition of the convolution of original image $f(x, y)$, kernel ω, and resulting image $g(x, y)$:

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

[2]

2.2 Explanation

To perform a convolution, the program uses a kernel 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 pixels are multiplied by the corresponding values in the kernel and then summed up. The result is then used as the new value of the pixel.

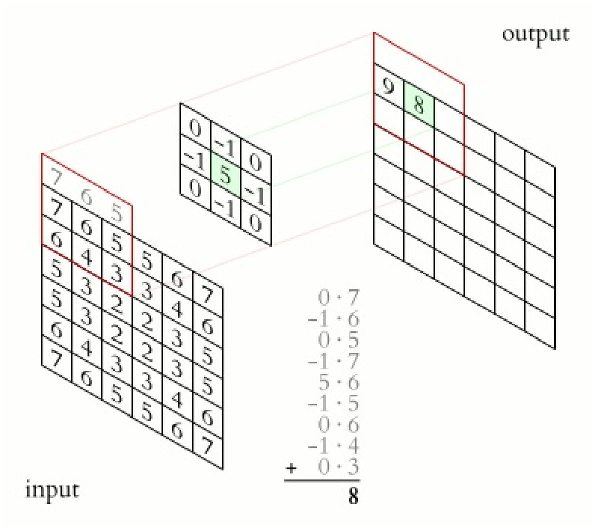


Figure 1: Convolution of an image with a kernel, illustrating how the output pixel value is calculated

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.

Processing an image sequentially, pixel by pixel, is inefficient, especially for large images and can be time-consuming. To overcome this, parallel computing offers a compelling solution, by partitioning the image into smaller, independent sub-images. These sections can be processed concurrently across multiple threads. Data used by different threads is not depending on one another, which makes it a trivially parallel problem. This approach can drastically accelerate image processing tasks compared to sequential methods.

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, applyKernelSequential, and applyKernelParallel. The applyKernelToPixel method applies the kernel to a single pixel, 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

```
Class ImageProcessor

  Function applyKernelToPixel(origImg, kernel, newImg, x, y)
    Initialize r, g, b to 0
    for each value in kernel do
      Add (origImg pixel $*$ kernel value)
    end for
    Clamp r, g, b to [0, 255]
    Set newImg at (x, y) to (r, g, b) to r, g, b
  End Function

  Function applyKernelSequential(origImg, kernel)
    Initialize newImg
    for each pixel (x, y) in origImg do
      Call applyKernelToPixel(origImg, kernel, newImg, x, y)
    end for
    Return newImg
  End Function

  Function applyKernelParallel(origImg, kernel)
    Initialize newImg
    Use ForkJoinPool to process pixels in parallel
    Return newImg
  End Function

End Class
```

3.1 Sequential implementation

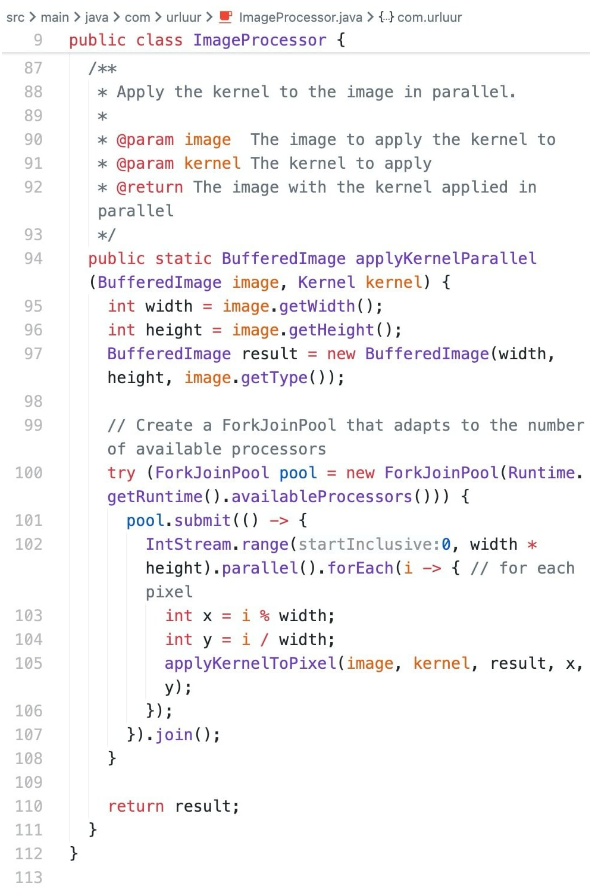
The applyKernelSequential method processes each pixel in the image one by one, row by row. This method is simple and easy to implement, but it can be slow for large images.

3.2 Parallel implementation

The applyKernelParallel method applies the kernel to the image in parallel using a ForkJoinPool that has threads corresponding to the number of computer processors. Then pixels are submitted to the pool and automatically asigned to any thread. I used ForkJoinPool because of these advantages:

- **Work-Stealing Algorithm**: It distributes tasks to idle threads for balanced workload.
- **Lower Overhead**: It reuses threads instead of creating new ones, reducing overhead.
- **Built-in Java Framework**: It is part of the Java standard library, making it easy to use.

Here is the code for the parallel processing:


src > main > java > com > urluur > ImageProcessor.java > {} com.urluur

```java
public class ImageProcessor {
  /**
   * Apply the kernel to the image in parallel.
   *
   * @param image  The image to apply the kernel to
   * @param kernel The kernel to apply
   * @return The image with the kernel applied in parallel
   */
  public static BufferedImage applyKernelParallel(BufferedImage image, Kernel kernel) {
    int width = image.getWidth();
    int height = image.getHeight();
    BufferedImage result = new BufferedImage(width, height, image.getType());

    // Create a ForkJoinPool that adapts to the number of available processors
    try (ForkJoinPool pool = new ForkJoinPool(Runtime.getRuntime().availableProcessors())) {
      pool.submit(() -> {
        IntStream.range(startInclusive:0, width * height).parallel().forEach(i -> { // for each pixel
          int x = i % width;
          int y = i / width;
          applyKernelToPixel(image, kernel, result, x, y);
        });
      }).join();
    }

    return result;
  }
}
```

Figure 2: Code for parallel processing that submits each pixel to the ForkJoinPool

Later I implemented parallel mode that splits the image into smaller chunks and processes them in parallel. This way, the processing time is further reduced due to less overhead. Its implementation is a bit more complicated, but both are similar. I decided to keep both implementations, so the user can choose which mode to use.

3.3 Distributed implementation

For distributed computing I used MPI (Message Passing Interface) in MPJ Express. I split the image into smaller parts and sent them to other machines for processing. The results were then sent back to the main machine and combined to form the final image.

3.3.1 Key Methods

- `masterDistributed`: This method sends the image and kernel to the workers, then processes one chunk of the image. It then receives the results from the workers and combines them to form the final image.
- `workerDistributed`: This method receives the chunk of an image and kernel from the master, processes the chunk, and sends the results back to the master.
- `applyKernelDistributed`: This method is used by master and workers to process a chunk of the image using the kernel.

3.3.2 Performance Considerations

While distributed computing can theoretically provide significant performance improvements, the actual performance gains depend on several factors:

- **Network Latency**: The time taken to send and receive chunks over the network can impact performance.
- **Data Transfer Overhead**: Large images have significant overhead because of their size.
- **Synchronization**: Ensuring that all chunks are combined correctly can introduce additional delays.

4 Testing environment

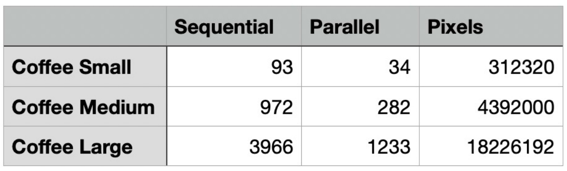
I tested the program on my laptop computer with the following specifications:

- **Processor**: Apple M1 chip
- **RAM**: 8 GB unified memory
- **Operating System**: macOS Sonoma
- **Java Version**: OpenJDK 22
- **MPI Version**: MPJ Express 0.44

5 Results

I tried running the program in sequential mode and in parallel mode with the demo picture of coffee of three different sizes. Results in sequential mode were following the number of pixels in the image. The results in parallel mode were much faster than in sequential mode.

The difference was significant. It noted speedups of nearly up to 4x. I assume that the speedup number is connected to the number of processor cores. Such speedup was present when processing small and large images.



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

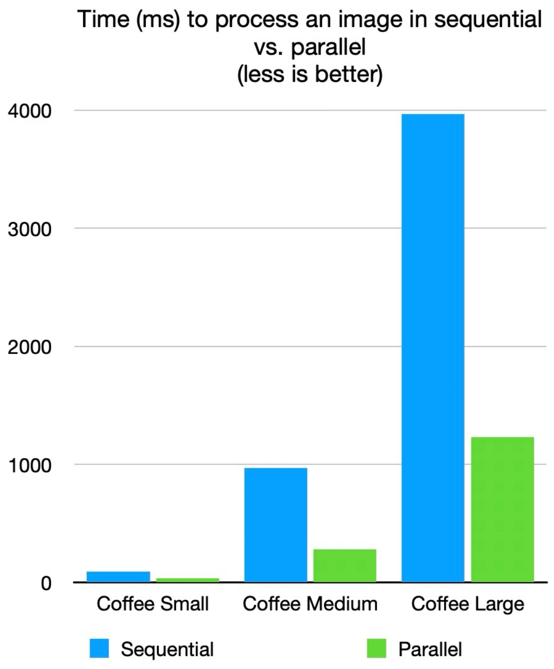


Figure 3: Initial speedup comparison between sequential and parallel processing

I was satisfied with the results of the parallel processing, compared to the very slow sequential approach. This proves the problem is really trivially parallel problem.

However, operating system, processor temperature and other programs running in the background can affect the results.

5.1 Average measurements

Because of the slightly inconsistent measurements, I decided to run the it 10 times and take the average. The results were consistent, and the speedup was still present. The average speedup was around 3.5x.

In the same way, I measured the average times of all implemented modes. The data is shown in the following table.

Coffee Small Time Measurement

	Parallel (Blocks)	Parallel (Pixels)	Sequential	Distributed
1	33	33	89	89
2	36	36	89	107
3	37	33	90	110
4	35	33	90	108
5	36	32	88	103
6	36	34	89	109
7	33	30	91	108
8	29	34	90	107
9	33	35	90	109
10	36	35	89	106

Coffee Medium Time Measurement

	Parallel (Blocks)	Parallel (Pixels)	Sequential	Distributed
1	307	335	911	1089
2	302	304	886	1027
3	312	298	899	1046
4	297	300	902	1055
5	328	309	916	1106
6	321	328	917	1041
7	324	355	913	1070
8	323	331	908	1078
9	320	324	910	1057
10	325	318	904	1059

Coffee Large Time Measurement

	Parallel (Blocks)	Parallel (Pixels)	Sequential	Distributed
1	1222	1233	3972	4161
2	1256	1239	4016	4170
3	1252	1246	4008	4184
4	1269	1202	4022	4240
5	1234	1203	4030	4231
6	1273	1281	4033	4181
7	1195	1228	3986	4140
8	1269	1219	4016	4175
9	1200	1232	4004	4155
10	1254	1243	4031	4222

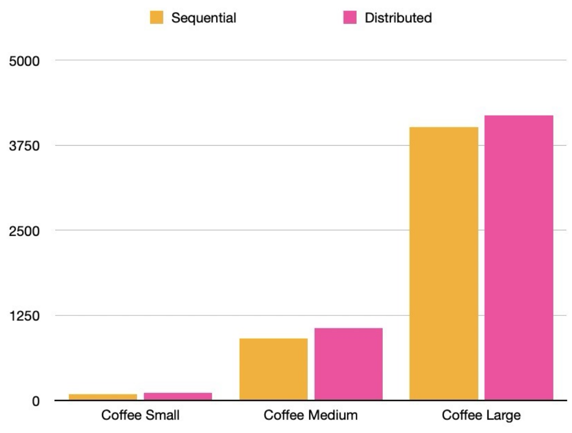
Figure 4: Measurements of processing times

I noticed that the time it took to process the image in distributed mode was slower than in sequential. This was surprising to me, as I expected distributed computing to be much faster, evem faster than parallel, but it turns out it was the slowest.

The distributed mode was actually very close to sequential mode, so I compared the two in the graph. Both parallel versios were way ahead of the other two modes, but the tested images were not large enough to see any significant difference between the two parallel modes.

Average Processing Time

	Coffee Small	Coffee Medium	Coffee Large
Parallel (Blocks)	34,4	315,9	1242,4
Parallel (Pixels)	33,5	320,2	1232,6
Sequential	89,5	906,6	4011,8
Distributed	105,6	1062,8	4185,9



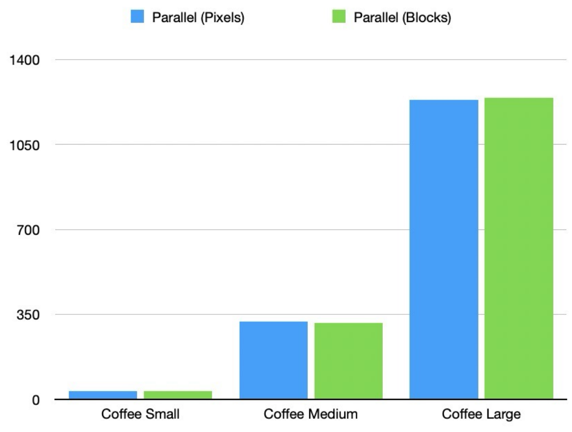


Figure 5: All Average Measurements

The reason for distributed mode not reaching my expectations is that the overhead of sending the chunks of an image was too high.

6 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

[1] Setosa. "Image kernels explained visually." Accessed: 2024-05-16. (2015), [Online]. Available: `https://setosa.io/ev/image-kernels/`.

[2] Wikipedia. "Kernel (image processing)." Accessed: 2024-07-29. (2019), [Online]. Available: `https://en.wikipedia.org/wiki/Kernel_(image_processing)`.