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".

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 ω, and resulting image g:

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

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

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.

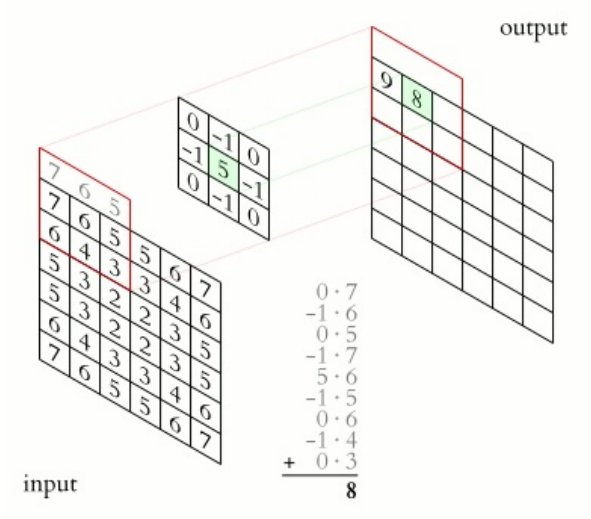

Figure 1: Convolution of an image with a kernel, illustrating how the output pixel value is calculated

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 can be 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

Class ImageProcessor
Function applyKernelToPixel(image, kernel, result, x, y)
for each value in the kernel **do**
    Multiply the corresponding pixel color by the kernel value
    Add the result to r, g, b
end for
Clamp r, g, b between 0 and 255
Set the pixel in the result image to the new color
End Function

Function applyKernel(image, kernel)
for each pixel in the image **do**
    Apply the kernel to the pixel
end for
End Function

Function applyKernelSequential(image, kernel)
Apply the kernel to the image
Return the result image
End Function

Function applyKernelParallel(image, kernel)
Use a ForkJoinPool to apply the kernel to each pixel in parallel
Return the result image
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. This approach can be much faster than the sequential method, especially for large images.

Here is the code snippet for the parallel processing:

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
        (BufferedImage image, Kernel kernel) {
95          int width = image.getWidth();
96          int height = image.getHeight();
97          BufferedImage result = new BufferedImage(width,
            height, image.getType());
98
99          // Create a ForkJoinPool that adapts to the number
            of available processors
100         try (ForkJoinPool pool = new ForkJoinPool(Runtime.
            getRuntime().availableProcessors())) {
101             pool.submit(() -> {
102                 IntStream.range(startInclusive:0, width *
                    height).parallel().forEach(i -> { // for each
                    pixel
103                     int x = i % width;
104                     int y = i / width;
105                     applyKernelToPixel(image, kernel, result, x,
                        y);
106                 });
107             }).join();
108         }
109
110         return result;
111     }
112 }
113
```

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

I also 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 tested both versions of parallel processing and decided to keep both implementations, so the user can choose which mode to use, because sometimes submitting each pixel to the pool can be faster than sequentially processing each chunk.

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 itself. 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 may incur significant overhead due to data transfer between machines.
- **Synchronization**: Ensuring that all chunks are processed and combined correctly can introduce additional complexity and potential delays.

4 Testing environment

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

- **Processor**: Apple M1 chip (8 cores; 4 performance cores, 4 efficiency cores)
- **RAM**: 8 GB unified memory
- **Operating System**: macOS Sonoma
- **Java Version**: OpenJDK 22
- **MPI Version**: MPJ Express 0.44

5 Results

I tested the program in sequential mode and in parallel mode with the demo picture of coffee 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

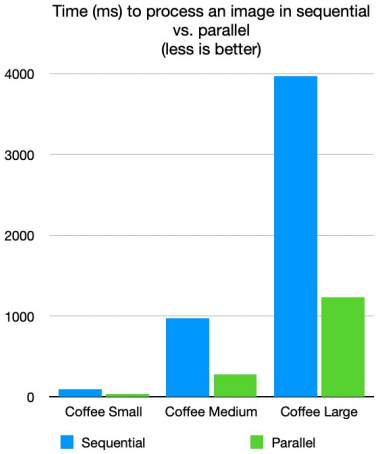

Figure 3: Speedup comparison between sequential and parallel processing

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.

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

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).