# Kernel image processing

**Gustas Linkus**

E-mail: 76230055@student.upr.si

*In this paper I will summarize process of image processing using small matrix – Kernel. I will present what is Kernel image processing, how it works, benefits of it, implementation of algorithm, design of the system. Moreover, I will provide my implementation of algorithm, results of program computing time vs image size.*

## 1 Introduction

Kernel image processing is a technique in digital image manipulation and analysis. Its primary purpose is to modify images in a way that either enhances certain features or extracts useful information that may not be immediately apparent in the raw image itself. This manipulation is achieved through a process called convolution, where a matrix, known as a kernel or filter, is applied over an image to produce a transformed version of the original.

Convolution is a mathematical operation that involves a kernel matrix sweeping over an image, applying a series of calculations to each pixel and its neighbors. This operation is akin to looking at the image through a "lens" that emphasizes or de-emphasizes certain patterns in the pixel values. The kernel's design—which can vary widely in shape, size, and the values it contains—determines the nature of the transformation. These transformations can range from sharpening, blurring, edge detection, to more complex operations that can isolate features, remove noise, or perform artistic effects.

The applications of kernel image processing are vast and diverse. In computer vision, kernels are used to preprocess images for better feature extraction, which is critical for tasks such as facial recognition, object detection, and autonomous vehicle navigation. In the realm of photography and graphic design, convolution filters are used to enhance image quality or create visual effects.

In essence, kernel image processing is an indispensable tool in the digital world, providing the means to enhance, analyze, and transform images in ways that extend the capabilities of human vision and machine-based image analysis. Through the iterative refinement of convolution techniques, image processing continues to advance, pushing the boundaries of what we can visualize and comprehend from the world captured in pixels.

## 2 Design

Designing the program in the sequential mode, the algorithm processes the image in a single-threaded manner, operating on one pixel at a time without parallelization. This method is fundamental to understanding the efficiency and behavior of the algorithm without the complexity of concurrent or distributed system interactions.

S**equential implementation** algorithm performs the following steps:

1. It loads the input image into a buffer as a two-dimensional array of pixels.
2. The algorithm then iterates over each pixel in the image, applying the kernel convolution operation. This is done by overlaying the kernel matrix centered on the current pixel and multiplying the surrounding pixels by the corresponding values in the kernel matrix.
3. The results of these multiplications are then summed up to compute the new value for the current pixel. This value is a weighted sum of the original pixel values in the neighborhood defined by the kernel.
4. The new pixel values are written to an output buffer, creating the transformed image.

This sequential approach is essential in the context of image processing, as it represents the most straightforward method to apply convolution filters. It is also a prerequisite to understanding and optimizing more advanced methods such as parallel or distributed processing. The results from the sequential processing tests provide a benchmark against which the performance improvements of parallel and distributed methods can be measured.

In sum, the sequential part of the implementation showcases the fundamental operations of kernel convolution and sets the stage for evaluating the algorithm's performance in the simplest computational setting.

**Parallel approach** divides the image into segments, each handled by a separate thread(number of threads equal to the number of CPU cores). Each processing of segment is submitted to the executor service. Then processed segments are combined to single final image.

**Distributed approach** uses Message Passing Interface implementation in Java – MPJ. The image is divided into horizontal segments with each node processing a different segment. Helper method applies a convolution kernel to its assigned segment, convolution involves iterating over each pixel in the segment, applying the kernel to compute new RGB values, and handling boundary conditions. Each node converts its processed segment into an integer array then master node gathers all the processed segments and reconstructs the full image from the gathered segments.

All three implementations use the same processing logic from sequential implementation (read above). Distributed method should be the least time consuming but the most difficult to implement and sequential should be overwise. The computation time will be tested for different

## 3 Results

I have tested my project with 10 different size images. Here is the tables of computational time vs image size(image is 3:2)

| Width, px | Sequential, ms |
|---|---|
| 235 | 302 |
| 470 | 466 |
| 846 | 801 |
| 1222 | 1256 |
| 1598 | 1771 |
| 1974 | 2199 |
| 2352 | 3299 |
| 3000 | 3940 |
| 4000 | 5826 |
| 8000 | 29592 |

*Table 1 Sequential part computational time*

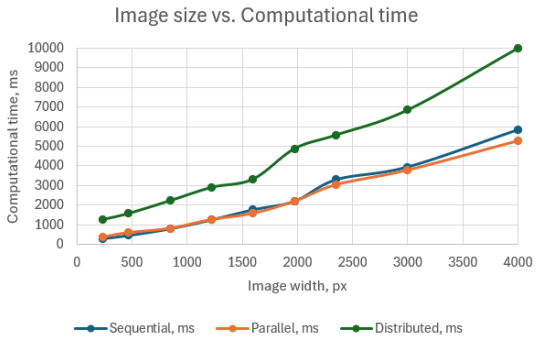
| Width, px | Parallel, ms |
|---|---|
| 235 | 362 |
| 470 | 590 |
| 846 | 807 |
| 1222 | 1262 |
| 1598 | 1576 |
| 1974 | 2197 |
| 2352 | 3048 |
| 3000 | 3797 |
| 4000 | 5300 |
| 8000 | 27512 |

*Table 2 Parallel part computational time*

| Width, px | Distributed, ms |
|---|---|
| 235 | 1260 |
| 470 | 1573 |
| 846 | 2223 |
| 1222 | 2902 |
| 1598 | 3319 |
| 1974 | 4873 |
| 2352 | 5573 |
| 3000 | 6843 |
| 4000 | 9990 |
| 8000 | „Java heap space“ |

*Table 3 Distributed part computational time*

In graph below I have excluded the width = 8000px.




As we could have guessed parallel part takes the least time to compute with bigger pictures. With small images sequential is the best choice. Distributed part gave results not as expected but stable – at every iteration with different size pictures it had worse results than parallel or sequential implementation. There are couple of possible explanations.

First, network latency. In a distributed system, data must be transferred between nodes over a network. This can introduce significant latency and reduce performance.
Secondly, data serialization. Data must be serialized (converted to a byte stream) before being sent over the network and deserialized (reconstructed from the byte stream) upon receipt. This process can add significant computational overhead.

Now I will provide some images how convulsion is applied. I used kernel matrix like this:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

This matrix is used to sharpen images. The image before:

*Figure 1 Before sharpening*

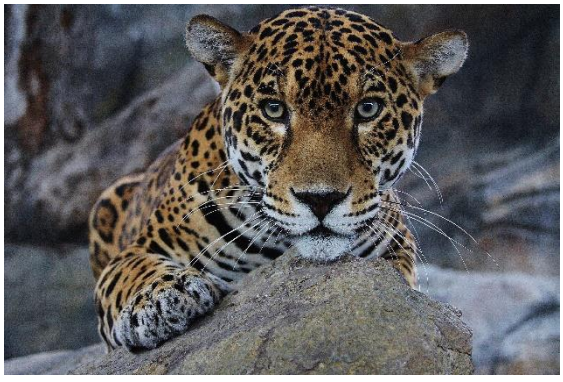


*Figure 2 After sharpening*

You will have to look closely at the pictures to see a difference, but it is visible. The size of the second image is more than visible – 2031 KB vs 9919 KB.

At the second test I will use edge detection matrix:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Then result photo will look like this:

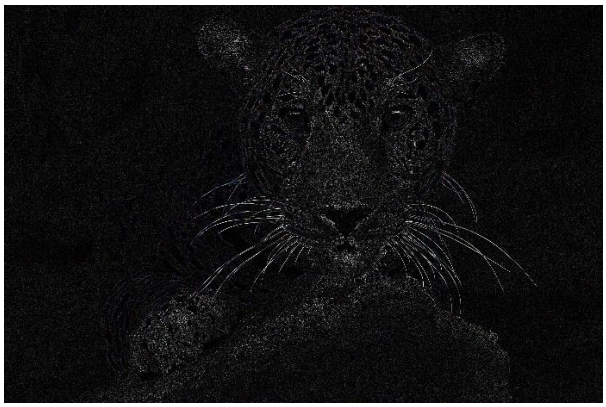


*Figure 3 After edge detection*

You can compare figures 1 and 3 and you will see a difference between.

## 4 Conclusion

In conclusion, this paper provides an in-depth exploration of kernel image processing, demonstrating the versatility and power of convolution operations in digital image manipulation. Kernel image processing, through the application of small matrices known as kernels, enables various image transformations such as sharpening, blurring, and edge detection.

The implementation of the kernel convolution algorithm in different computational settings—sequential, parallel, and distributed—highlights the trade-offs between ease of implementation, computational complexity, and performance.

Sequential approach: the simplest to implement and understand, sequential processing provides a baseline for performance measurement. It is efficient for smaller images but becomes time consuming as image size increases due to its single-threaded nature.

Parallel approach: by leveraging multi-core processors, parallel processing significantly reduces computation time for larger images. The image is divided into segments, each processed concurrently, resulting in faster processing times compared to the sequential approach.

Distributed approach: although theoretically the most scalable, distributed processing introduces significant overhead due to network latency and data serialization. The results showed that, contrary to expectations, distributed processing was less efficient than parallel and sequential processing for the tested image sizes. This inefficiency is due to the communication overhead between distributed nodes and the complexity of managing distributed resources.