Image Kernel Processing Coding Project

Florian Daunay
Università degli studi Firenze
florian.daunay@gmail.com

Abstract

This project explores the performance of sequential and parallel image kernel processing in Python across various image sizes. Using square images from 50x50 to 2000x2000 pixels, we evaluate execution times for both methods. Surprisingly, the sequential approach outperforms for smaller images, while parallel execution, especially with 8 cores, excels as image dimensions increase. These findings offer valuable insights for optimizing image processing workflows, emphasizing the need for tailored strategies based on dataset characteristics.

1. Future Distribution Permission

The author(s) of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses. GitHub link: `https://github.com/Mr-Donot/kernel_image_processing`

2. Introduction

This report focuses on comparing the performance of sequential and parallel implementations of kernel processing specifically applied to images. Sequential processing involves linear execution, while parallel processing enables simultaneous task execution. With the increasing demand for computational efficiency in handling large image datasets, this report aims to analyze the trade-offs between these approaches. By providing insights into their advantages and limitations, we aim to guide developers and researchers in making informed decisions for optimizing image processing workflows in Python.

2.1. Definition

The general expression of a convolution is shown in figure 1.

Where g(x, y) is the filtered image, f(x, y) is the original image, w is the filter kernel. Every element of the filter kernel is considered by $-a \leq i \leq a$ and $-b \leq j \leq b$. Depending on the element values, a kernel can cause a wide range of effects, such as : ridge, edge detection, sharpen, blur, unsharp masking...

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

Figure 1. Convolution formula

Image kernel processing involves the application of a convolution operation to enhance or modify pixel values within an image. The core of this process lies in the systematic traversal of a convolution kernel across the entire image. As the kernel moves over each pixel, a weighted sum of the neighboring pixels is computed.

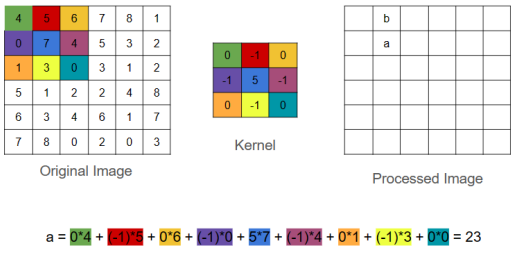

Figure 2. Kernel applied on a pixel

To ensure consistent processing across the entire image, extend edge handling is employed at the image edges : The nearest border pixels are conceptually extended as far as necessary to provide values for the convolution. Corner pixels are extended in 90° wedges. Other edge pixels are extended in lines.

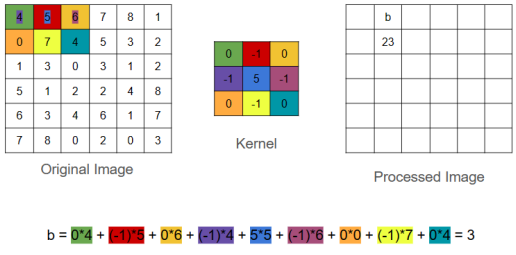

Figure 3. Edges case

The convolution operation is conducted iteratively for each pixel, resulting in a transformed image that reflects the influence of the chosen kernel. This approach enables the extraction of specific features or enhancements, depending on the characteristics encoded within the kernel matrix. The subsequent subsections will delve into the details of the convolution process, zero-padding strategy, and the impact of different kernels on image features.

3. Setup

For the experiments conducted in this report, the following hardware and software configurations were employed:
Python Version: Python 3.9.13
Processor: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
Cores: 4
Logical Processors: 8
RAM: 16 GB
These specifications provide a basis for evaluating the performance of sequential and parallel implementations of image kernel processing in Python. The 11th Gen Intel Core i7 processor, with its multiple cores and logical processors, enables us to explore the potential benefits of parallel processing. The substantial 16 GB RAM ensures sufficient memory availability for handling large image datasets.

4. Dataset

The experiments conducted in this report involve a diverse set of square images, each varying in size. The sizes of the images are as follows: 50x50, 100x100, 200x200, 300x300, 400x400, 500x500, 1000x1000, 1500x1500, and 2000x2000 pixels. This range of image sizes allows for a comprehensive assessment of the performance of sequential and parallel implementations of kernel processing across different dimensions.

These images serve as the input data for the image kernel processing tasks, and the variations in size aim to simulate real-world scenarios where datasets may differ significantly in scale.

5. Sequential Implementation

In the sequential part of our image kernel processing, the primary goal is to apply a given kernel to each pixel of the image. The kernel is a small matrix that defines the weights used to calculate the new value of a pixel based on its neighbors. The sequential implementation processes each pixel of the image one at a time, following a linear order.

5.1. Algorithm overview

1. **Input Parameters:** The function `process_image` takes an image and a kernel as input parameters.
2. **Image Initialization:** A new image (`newImg`) of the same size as the input image is created, initialized with black pixels.
3. **Pixel Iteration:** Two nested loops iterate over each pixel in the input image.
4. **Neighborhood Extraction:** For each pixel, a neighborhood is extracted based on the dimensions of the kernel. The values are obtained from the input image, ensuring proper handling of pixels at the edges.
5. **Pixel Value Calculation:** The `get_pixel_value` function calculates the new pixel value using the kernel and the extracted neighborhood.
6. **New Image Pixel Assignment:** The calculated pixel value is assigned to the corresponding position in the new image.

5.2. Coding explanation

```python
def get_pixel_value(initial_pixels, kernel) -> [int, int, int]:
    result = [0,0,0]
    for i in range(len(kernel)):
        for j in range(len(kernel[i])):
            for color in range(len(result)):
                result[color] += initial_pixels[i][j][color] * kernel[i][j]
    return result
```

Figure 4. Applying kernel on a pixel function

The get pixel value function calculates the new pixel value by applying the kernel to the given neighborhood.

```python
def process_image(img: Image, kernel) -> Image:
    (largeur,hauteur)=img.size
    newImg = Image.new('RGB', (largeur,hauteur), color = (0, 0, 0))
    for i in range (largeur):
        for j in range (hauteur):
            initial_pixels = []
            for k in range(len(kernel)):
                pixel_line = []
                for l in range(len(kernel[k])):
                    pos_i = i-1+l if 0 <= i-1+l < largeur else i
                    pos_j = j-1+k if 0 <= j-1+k < hauteur else j
                    pixel_line.append(img.getpixel((pos_i, pos_j)))

                initial_pixels.append(pixel_line)

            new_value = get_pixel_value(initial_pixels, kernel)
            newImg.putpixel((i,j), tuple(new_value))
    return newImg
```

Figure 5. Processing an image function

The process image function iterates through each pixel of the input image, extracts the neighborhood, calculates the new pixel value, and assigns it to the corresponding position in the new image.

In the subsequent sections, we will present the parallel implementation and provide a detailed performance analysis of both approaches.

6. Parallel implementation

In the parallel part of our image kernel processing, the objective is to leverage multiple processes to concurrently handle distinct strips of the image. This approach aims to distribute the computational load across multiple cores, potentially enhancing the overall processing speed.

6.1. Algorithm overview

1. **Input Parameters:** The variable The function process strip takes an image (img), a kernel, and the start and end rows as input parameters. It processes a specific strip of the image.

2. **Strip Initialization:** A new image strip (strip) is created for the specified rows, initialized with black pixels.

3. **Parallel Processing:** The process image parallel function divides the image into strips based on the number of specified processes. Each strip is processed independently by a separate process using the multiprocessing module.

4. **Results Aggregation:** The results from each parallel process are combined to reconstruct the final image.

6.2. Code explanation

```python
def process_strip(img, kernel, start_row, end_row):
    (width, height) = img.size
    strip = Image.new('RGB', (width, end_row - start_row), color=(0, 0, 0))
    for i in range(width):
        for j in range(start_row, end_row):
            initial_pixels = []
            for k in range(len(kernel)):
                pixel_line = []
                for l in range(len(kernel[k])):
                    pos_i = i - 1 + l if 0 <= i - 1 + l < width else i
                    pos_j = j - 1 + k if 0 <= j - 1 + k < height else j
                    pixel_line.append(img.getpixel((pos_i, pos_j)))
                initial_pixels.append(pixel_line)
            new_value = get_pixel_value(initial_pixels, kernel)
            strip.putpixel((i, j - start_row), tuple(new_value))
    return start_row, end_row, strip
```

Figure 6. Processing a strip of the image

The process strip function is responsible for processing a specific strip of the image. It follows a similar logic to the sequential implementation but operates on a subset of rows.

The process image parallel function divides the image into strips, processes them in parallel using the multiprocessing.Pool, and then combines the results to reconstruct the final image. This parallel approach aims to exploit the capabilities of multi-core processors for accelerated image kernel processing.

```python
def process_image_parallel(img: Image, kernel, num_processes) -> Image:
    (width, height) = img.size
    strip_height = height // num_processes

    with multiprocessing.Pool(processes=num_processes) as pool:
        results = []
        for i in range(0, height, strip_height):
            end_row = min(i + strip_height, height)
            results.append(
                pool.apply_async(process_strip, (img, kernel, i, end_row))
            )

        new_img = Image.new('RGB', (width, height), color=(0, 0, 0))
        for result in results:
            start_row, end_row, strip = result.get()
            new_img.paste(strip, (0, start_row))

    return new_img
```

Figure 7. Processing an image function

7. Comparaison

The comparison between the sequential and parallel implementations of image kernel processing is essential to evaluate the potential performance gains achieved through parallelization. In this experiment, we consider three scenarios: sequential execution, parallel execution with 2 processes, and parallel execution with 8 processes.

7.1. Sequential Execution

In the sequential execution, the image is processed pixel by pixel, following a linear order. Each pixel's new value is calculated based on its neighbors, and the process continues until the entire image is processed. This approach is straightforward but may limit the utilization of multi-core processors, especially when dealing with large datasets.

7.2. Parallel Execution (2 processes)

The parallel execution with 2 processes aims to exploit the capabilities of multi-core processors by dividing the image into two strips and processing them concurrently. Each process independently handles its assigned strip, potentially reducing the overall processing time. This approach is expected to show improvements over the sequential execution, especially for mid-sized images.

7.3. Parallel Execution (8 processes)

Expanding on the parallelization, the execution with 8 processes further subdivides the image into more strips, allowing for greater parallelism. This approach leverages the full potential of an 8-core processor, aiming for significant speedup, especially with larger images. However, the benefits might diminish for smaller images or due to potential overhead associated with managing multiple processes.

7.4. Code for the testing

The following code snippet conducts a comparative analysis of sequential and parallel implementations of image kernel processing across a range of image sizes:

1. The variable `imgs_path` contains a range of square image dimensions.

2. For each image size, the script measures the execution time of both the sequential and parallel implementations.

3. Parallel execution is tested with 2 and 8 processes.

4. Recorded execution times (`seq_time`, `par_time`, `par_time2`) for subsequent analysis and graph plotting.

```
seq_time = []
par_time = []
par_time2 = []
imgs_path = [50, 100, 200, 300, 400, 500, 1000, 1500, 2000]
for path in imgs_path:

    img = Image.open("img/" + str(path) + "x" + str(path)+ ".png")
    t0 = time()
    img2 = process_image(img, kernel_sharpen)
    t1 = time()
    img3 = process_image_parallel(img, kernel_sharpen, 2)
    t2 = time()
    img4 = process_image_parallel(img, kernel_sharpen, 8)
    t3 = time()
    seq_time.append(t1-t0)
    par_time.append(t2-t1)
    par_time2.append(t3-t2)
```

Figure 8. Testing function

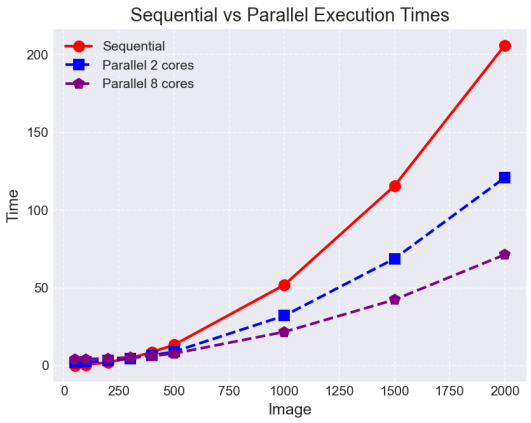

Figure 9. Execution time in regard to the size of images

8. Results

The experimental results reveal interesting insights into the comparative performance of sequential and parallel implementations of image kernel processing across varying image sizes. For smaller images, up to 200x200 pixels, the sequential approach exhibits superior performance, likely due to the lower computational overhead associated with managing parallel processes. However, as image dimensions increase, the advantage shifts towards parallelization. Notably, for 300x300 pixels, the execution times converge, showcasing the competitive nature of both approaches. Surprisingly, as image sizes further escalate, the parallel execution with 8 cores outperforms both the sequential and 2-core parallel implementations, indicating the scalability of parallel processing for larger datasets. The accompanying graph visually represents these trends, illustrating the nuanced relationship between image size and the efficiency of parallelization strategies. In conclusion, the optimal choice between sequential and parallel implementations depends on the specific characteristics of the image processing task, emphasizing the need for careful consideration when designing computational workflows for diverse datasets.