Kernel Image Processing (2D Convolution) in Python

Mirko Bicchierai
E-mail address
mirko.bicchierai@edu.unifi.it

Abstract

Convolution is a fundamental operation in signal processing, image analysis, and deep learning. In the realm of image processing, discrete convolution is applied to filter images or preprocess them before machine learning tasks. In deep learning, convolution is a critical operation within Convolutional Neural Networks (CNNs), playing a vital role during both training and inference phases. With the increasing demand for real-time processing of large datasets, the efficient parallelization of convolution algorithms has become essential. The goal of this project is to parallelize the execution of image convolution with a kernel in Python, effectively applying a filter to the image. All project-related materials can be found in the GitHub repository: `https://github.com/MirkoBicchierai/Parallel_Programming_Kernel_2DConvolution`.

1. Introduction

This report presents an exploration of the 2D convolution operation, focusing on its fundamentals and the strategies employed to parallelize its computations. The parallelization was implemented in Python, primarily using the *multiprocessing* library. There are many methods to perform image convolutions. In this report, we will focus on transforming the image into a matrix across the three channels—red, green, and blue. We will then perform matrix convolution with the kernel on all three channels, and finally, convert the resulting matrix back into an image.

2. Convolution

The convolution of $f()$ and $g()$ is written $f * g$, it is defined as the integral of the product of the two functions after one is reflected about the y-axis and shifted. As such, it is a particular kind of integral transform:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)\, d\tau$$

At each t, the convolution formula can be described as the area under the function $f(\tau)$ weighted by the function $g(-\tau)$ shifted by the amount t.

We are however focusing on 2-dimensional discrete convolution operation in which both function has finite support, in this case the convolution has the following form:

$$(f * g)[m, n] = \sum_{i=-a}^{a} \sum_{j=-b}^{b} f[m - i, n - j] \cdot g[i, j]$$

In the context of image processing, the function f represents the input image, while the function g denotes the kernel matrix. Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel.

For example, if we have two three-by-three matrices, the first a kernel, and the second an image piece, convolution is the process of flipping both the rows and columns of the kernel and multiplying locally similar entries and summing. The element at coordinates [2, 2] (that is, the central element) of the resulting image would be a weighted combination of all the entries of the image matrix, with weights given by the kernel:

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right)[2,2]$$

$$= i + 2h + 3g + 4f + 5e + 6d + 7c + 8b + 9a$$

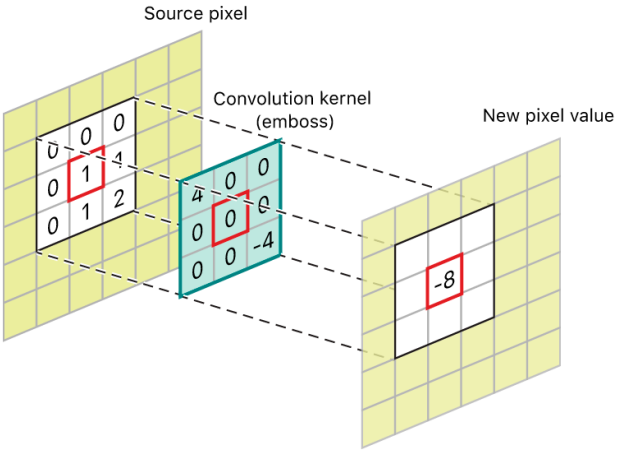

Figure 1. Example of image processing convolution.

When performing a 2D convolution operation on an input image of dimensions $N \times M$ with a kernel of size $k \times k$, the computational complexity is $O(NM \cdot k^2)$. However, if separable kernels are used, this complexity can be reduced to $O(2NM \cdot k)$. Despite this potential optimization, the focus of this study is to evaluate the speed-up achieved by the parallel version of the algorithm compared to the naive sequential approach. Therefore, the optimized algorithm using separable kernels will not be employed in this evaluation.

2.1. The naive serial algorithm

For the sequential version of the convolution algorithm, a vanilla implementation was developed in Python without any convolution optimizations, such as leveraging kernel separability. An important point to note is that the convolution of an image at the matrix level does not account for the borders of the image matrix. There are two main solutions to this issue: the first is to avoid the borders, resulting in an output image that is smaller and cropped; the second is to apply zero padding to the borders of the image matrix based on the size of the kernel used. The second solution was applied for the tests conducted, ensuring a fair comparison of execution times between the sequential and parallel versions.

Algorithm 1 Sequential algorithm of 2D convolution

```
def apply_convolution(img, kernel, height,
    width, pad_y, pad_x):
    result = np.zeros((height, width, 3))
    for i in range(height):
        for j in range(width):
            convolved_value = np.zeros(3)
            for dy in range(-pad_y, pad_y +
                1):
                for dx in range(-pad_x,
                    pad_x + 1):
                    convolved_value+=img[i+
                        pad_y+dy,j+pad_x+dx
                        ]*kernel[pad_y+dy,
                        pad_x+dx]
            result[i, j] = convolved_value
    result = np.clip(result, 0, 255)
    return result.astype(np.uint8)
```

The input variable `img` to the function represents the image matrix to which padding has already been applied to the borders. The variable `kernel` is the matrix of the kernel to be applied.

In order to solve the 2D convolution problem, the simplest approach is to loop over all the image pixels and all the kernel elements in one go. This algorithm is the most naive and slow one. It uses 4 nested loops: the 2 outer loops on the rows and columns of the image, and the 2 inner loops on the rows and columns of the kernel.

3. Algorithm correctness

To verify the correctness of the convolution algorithm, two methods can be used: the first is to manually compute all the results and check for consistency, while the second involves a graphical visualization of the output compared to the input. Additionally, to validate the correctness of the implemented parallel algorithm (see the next section), the output matrix of the sequential algorithm was compared with the output matrix of the parallel algorithm to ensure they were identical.

Figure 2. Example of input image (HD Resolution).

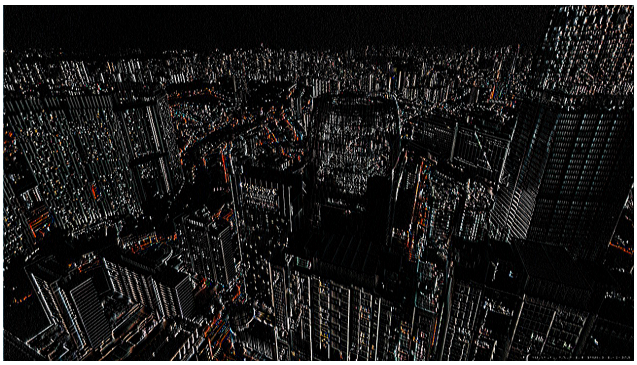

Figure 3. An example of the output where a Prewitt horizontal kernel of size 3×3 has been applied.

4. Parallelization

It is worth noting that the convolution operation is an example of an embarrassingly parallel problem. Each pixel's value can be computed independently, relying solely on the kernel values and the surrounding pixel values. This independence means that there are no dependencies or prerequisite computations between pixels, making the problem highly suitable for parallel processing.

In this project, the convolution code was parallelized using Python, primarily utilizing the `multiprocessing.Pool` library.

4.1. Parallelization strategy

As the primary parallelization strategy, the outermost `for` loop, which iterates over the height of the image, was parallelized. This approach involves creating multiple processes in Python, each handling a specific portion or strip of the image. Each process computes the convolution for its assigned strip. Once all processes have completed their tasks, the various outputs are collected and combined into a single matrix.

Algorithm 2 Parallel algorithm of 2D convolution

```
def parallel_apply_convolution_normal(img,
    kernel, num_workers, height, width,
    pad_y, pad_x):
    chunk_size = height // num_workers
    chunks = [(img, kernel, i * chunk_size,
         (i + 1) * chunk_size if i <
        num_workers - 1 else height, height,
         width,
               pad_y, pad_x) for i in
              range(num_workers)]
    with Pool(processes=num_workers) as
        pool:
        result_chunks = pool.map(
            apply_convolution_chunk_normal,
            chunks)
    result = np.vstack(result_chunks)
    result = np.clip(result, 0, 255).astype
        (np.uint8)
    return result

def apply_convolution_chunk_normal(args):
    img, kernel, start_row, end_row, height
        , width, pad_y, pad_x = args
    result = np.zeros((end_row - start_row,
         width, 3))
    for i in range(start_row, end_row):
        for j in range(width):
            convolved_value = np.zeros(3)
            for dy in range(-pad_y, pad_y +
                 1):
                for dx in range(-pad_x,
                    pad_x + 1):
                    convolved_value+=mg[i+
                        pad_y+dy,j+pad_x+dx
                        ]*kernel[pad_y+dy,
                        pad_x+dx]
            result[i - start_row, j] =
                convolved_value
    return result
```

The first function is responsible for generating the various processes and recombining the results. Meanwhile, the second is the function called by the various processes, which processes and returns a single strip.

4.2. Vectorization

Another aspect considered to improve performance was vectorizing the computation of each

output element in the matrix, specifically the convolution computation itself, pixel by pixel (or matrix element by matrix element). To achieve this, certain tricks were employed using the `numpy` library, which allowed the elimination of the two inner `for` loops related to the kernel. The vectorization is thus managed autonomously by the underlying directives of the `numpy` library, which are written in C.

Algorithm 3 Parallel algorithm of 2D convolution with vectorization

```
def parallel_apply_convolution(img, kernel,
     num_workers, height, width, pad_y,
    pad_x):
    chunk_size = height // num_workers
    chunks = [(img, kernel, i * chunk_size,
         (i + 1) * chunk_size if i <
        num_workers - 1 else height, height,
         width,
               pad_y, pad_x) for i in
              range(num_workers)]
    with Pool(processes=num_workers) as
        pool:
        result_chunks = pool.map(
            apply_convolution_chunk, chunks)
    result = np.vstack(result_chunks)
    result = np.clip(result, 0, 255).astype
        (np.uint8)
    return result

def apply_convolution_chunk(args):
    img, kernel, start_row, end_row, height
        , width, pad_y, pad_x = args
    result = np.zeros((end_row - start_row,
         width, 3))
    for i in range(start_row, end_row):
        for j in range(width):
            region = img[i:i + 2 * pad_y +
                1, j:j + 2 * pad_x + 1, :]
            result[i - start_row, j, :] =
                np.sum(region * kernel[:, :,
                 np.newaxis], axis=(0, 1))
    return result
```

Another important aspect to note is that each process is passed the entire image rather than just the necessary strip. It was tested with only the required strip of the image, but this did not improve performance. This is because passing only the single strip is insufficient; additional rows are needed to define the convolution at the borders. Therefore, each process requires the strip plus the necessary additional rows, both above and below, depending on the kernel size. For kernels of larger dimensions, passing only the strips actually resulted in a decrease in execution speed.

As demonstrated in the results section, this type of implementation has allowed us to achieve a significant improvement in terms of speedup, especially as the kernel size increases.

5. Hardware used

Experiments were performed on 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz (Mobile) (8 CPU cores and 16 threads) and NVIDIA GeForce RTX 3050 Ti (Mobile).

6. Results

This section presents the results in terms of the SpeedUp of the implemented convolution algorithm, defined as the ratio between the execution time of the sequential version and that of the parallel version. The analysis focuses on the SpeedUp achieved with varying image sizes and kernel dimensions.

6.1. Resolution tested

In this project, various image resolutions were tested to observe how the speedup improvement scaled between the sequential and parallel versions of the code as the image size varied. Below is a list of the tested resolutions along with their respective dimensions:

- **4K** (3840x2160)
- **2K** (2560x1440)
- **Full-HD** (1920x1080)
- **HD** (1280x720)
- **SD** (720x480)

For each resolution, a set of 5 different images was used, and the results were averaged to stabilize the findings. This approach was taken be-

cause minor variations in execution time were observed across different images of the same resolution. These differences are likely due to the influence of the image matrix values on the computational time of the matrix multiplication. Additionally, the execution time recorded for each individual image is averaged over 100 runs of the convolution function on the same image.

6.2. Kernel tested

There are many types of kernels for image convolutions, each serving a specific purpose, such as applying a blur filter to an image or performing edge detection. For the tests, numerous types of kernels were used, all of which can be found in the `kernel.py` file within the GitHub repository. However, for the final tests concerning speedup analysis, only three kernels of different sizes were considered: one 3×3, one 5×5, and one 7×7. This was done to demonstrate how a small variation in kernel size could have a significant impact on execution time.

Prewitt horizontal kernel of size 3×3:

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

Gaussian blur kernel of size 5×5:

$$\frac{1}{256} \cdot \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Gaussian blur kernel of size 7×7:

$$\frac{1}{140} \cdot \begin{bmatrix} 1 & 1 & 2 & 2 & 2 & 1 & 1 \\ 1 & 2 & 2 & 4 & 2 & 2 & 1 \\ 2 & 2 & 4 & 8 & 4 & 2 & 2 \\ 2 & 4 & 8 & 16 & 8 & 4 & 2 \\ 2 & 2 & 4 & 8 & 4 & 2 & 2 \\ 1 & 2 & 2 & 4 & 2 & 2 & 1 \\ 1 & 1 & 2 & 2 & 2 & 1 & 1 \end{bmatrix}$$

It is important to note that the Gaussian kernels are presented in a normalized form, where the scalar multiplying the matrix is the inverse of the sum of the matrix elements. This normalization allows for a cleaner visualization.

6.3. SpeedUp analysis

This section presents the results in terms of speedup achieved between the parallel version without vectorization and the sequential version. The kernel size and the image size will be varied to observe the speedup benefits as the number of processes generated with Python changes.

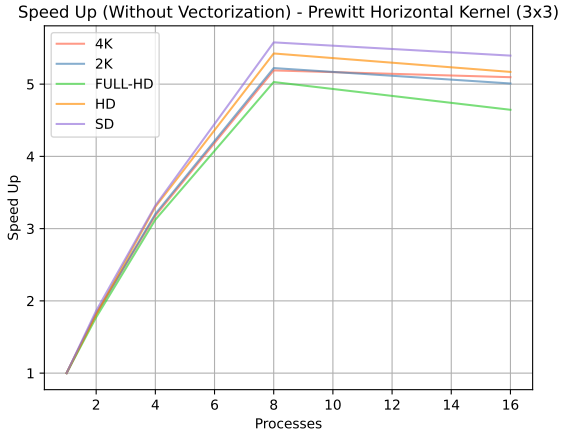

Figure 4. Speed Up graph using Prewitt Horizontal Kernel (3x3).

Resolution / processes	**2**	**4**	**8**	**16**
SD	1.86	3.31	**5.57**	5.40
HD	1.81	3.30	5.42	5.16
Full-HD	1.77	3.12	5.02	4.65
2K	1.81	3.20	5.22	5.00
4K	1.82	3.17	5.18	5.01

Table 1. Speed Up table using Prewitt Horizontal Kernel (3x3).

As we can see from the graph 4 and the corresponding table with numerical values (Table 1), the speedup analysis for processing an image with a Prewitt horizontal kernel of size 3×3 shows maximum average values around 5. Another noticeable point from the graph is that increasing the number of processes from 8 to 16 results in a decrease in the speedup curve. Although this is not ideal, it is an expected outcome when parallelizing with Python, as the generated processes

lead to a form of pseudo-parallelization. Additionally, the machine used for the tests has 8 physical cores.

Another noticeable result is that images of lower quality, specifically SD, exhibit a better speedup. However, it is important to note three aspects that can justify the fact observed:

- First, the improvement is very slight and almost negligible, in the order of 0.1 - 0.2.
- Second, there is always an overhead associated with the creation and management of processes, the division of labor, and the subsequent recombination of results. This overhead has a relatively larger impact on larger images compared to smaller ones because the total computation time is shorter for smaller images, making the overhead a more significant fraction of the total time.
- Additionally, lower-resolution images can benefit more from CPU cache compared to higher-resolution ones. The data of a smaller image are more likely to fit entirely in the CPU cache, reducing memory access time and improving performance. With larger images, the cache is less effective, as data must be frequently exchanged with the main memory.

All this has nonetheless been partially verified by using Python code profiling techniques.

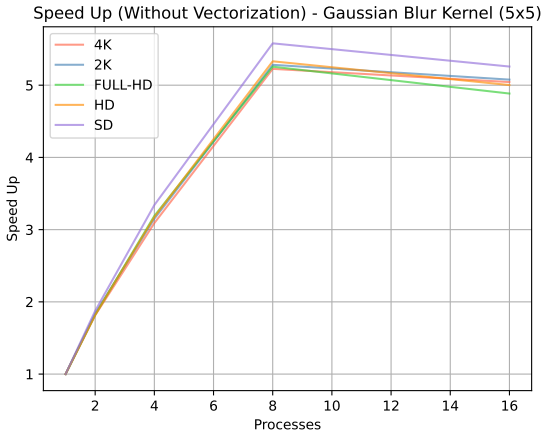

Figure 5. Speed Up graph using Gaussian Blur Kernel (5x5).

Resolution / processes	2	4	8	16
SD	1.87	3.34	**5.57**	5.26
HD	1.81	3.17	5.33	5.01
Full-HD	1.82	3.19	5.25	4.88
2K	1.84	3.15	5.28	5.08
4K	1.82	3.09	5.22	5.04

Table 2. Speed Up table using Gaussian Blur Kernel (5x5).

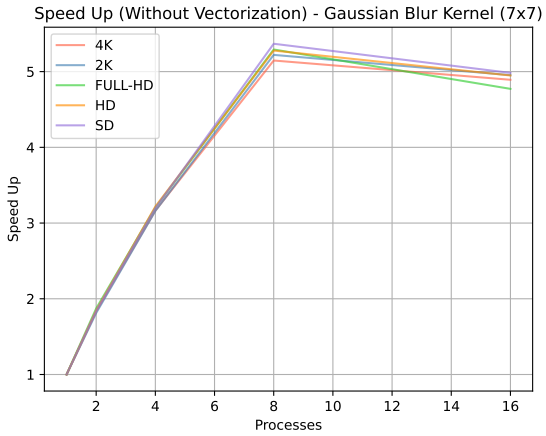

Figure 6. Speed Up graph using Gaussian Blur Kernel (7x7) .

Resolution / processes	2	4	8	16
SD	1.83	3.19	**5.36**	4.98
HD	1.84	3.21	5.27	4.95
Full-HD	1.87	3.20	5.29	4.78
2K	1.81	3.16	5.22	4.96
4K	1.83	3.16	5.13	4.90

Table 3. Speed Up table using Gaussian Blur Kernel (7x7).

These same results can also be observed as the kernel size increases, for the same reasons. As we can see from the graphs 5, 6 and the corresponding tables with numerical values (Tables 2 and 3).

Another notable result is that with this parallelism technique, the speedup achieved does not change significantly with varying kernel sizes. This contrasts with the situation when vectorization is used for the convolution computation, where the speedup does vary with kernel size, as we will see later. On average, the speedup remains around 5 regardless of the kernel size.

6.3.1 Vectorization analysis

In this section, we will analyze the results obtained in terms of speedup from the parallel version of the convolution code, which includes vectorization of the matrix operations for the convolution computation.

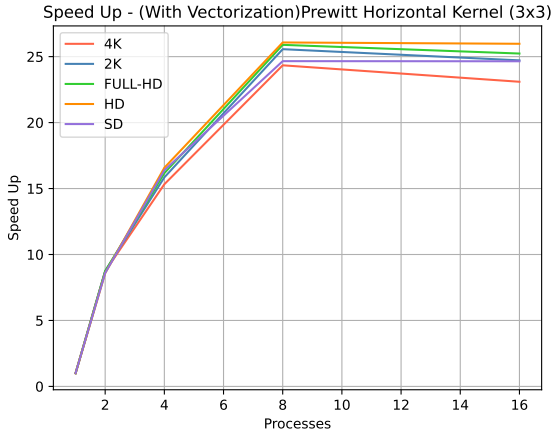

Figure 7. Speed Up graph using Prewitt Horizontal Kernel (3x3) with vectorization strategy.

Resolution / processes	**2**	**4**	**8**	**16**
SD	8.59	16.38	24.65	24.64
HD	8.58	16.57	**26.07**	25.97
Full-HD	8.72	16.16	25.89	25.24
2K	8.75	15.84	25.56	24.71
4K	8.72	15.32	24.34	23.09

Table 4. Speed Up table using Prewitt Horizontal Kernel (3x3) with vectorization strategy.

Looking at the results obtained with a 3×3 kernel, we can observe a significant improvement compared to the previous version of the code. This is evident from both the graph 7 and the associated numerical table 4. We achieve approximately double the speedup on average (between 24 and 26). However, it is worth noting that images of lower quality still experience slightly better speedup for the same reasons discussed in the previous section.

Another detail to note is a certain stabilization in speedup when transitioning from 8 to 16 processes, compared to the non-vectorized version. Although the impact is modest, this is an additional advantage of the vectorization methodology.

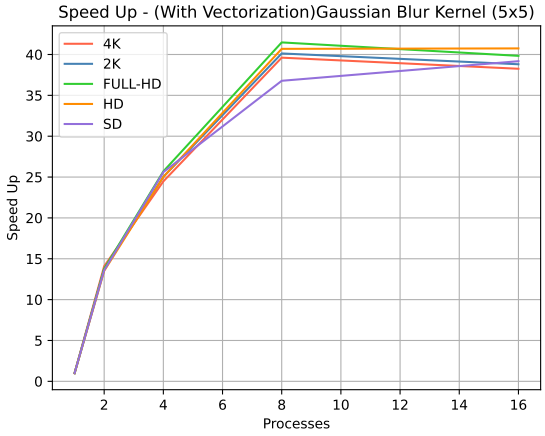

Figure 8. Speed Up graph using Gaussian Blur Kernel (5x5) with vectorization strategy.

Resolution / processes	**2**	**4**	**8**	**16**
SD	13.58	25.60	36.78	39.17
HD	13.51	25.02	40.69	40.74
Full-HD	13.77	25.69	**41.47**	39.83
2K	13.74	25.02	40.12	38.80
4K	14.08	24.47	39.61	38.25

Table 5. Speed Up table using Gaussian Blur Kernel (5x5) with vectorization strategy.

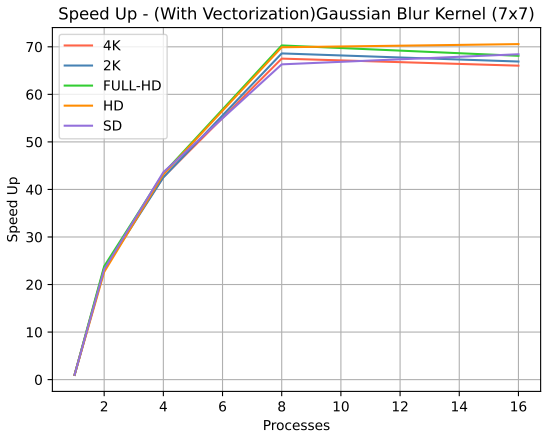

Figure 9. Speed Up graph using Gaussian Blur Kernel (7x7) with vectorization strategy.

Resolution / processes	2	4	8	16
SD	23.21	43.62	66.31	68.43
HD	22.63	43.19	69.90	**70.56**
Full-HD	23.83	43.36	70.29	68.10
2K	23.15	42.52	68.61	66.90
4K	23.50	42.59	67.51	66.03

Table 6. Speed Up table using Gaussian Blur Kernel (7x7) with vectorization strategy.

As can be seen from the results obtained with increasingly larger kernels, there is a significant improvement in speedup as the kernel size increases. This is a notable departure from the previous results, where speedup remained relatively constant regardless of kernel size. As observed from the graphs and their corresponding tables, there is a consistent increase in speedup with larger kernel sizes. Starting from a 3×3 kernel, which achieved the highest speedup score of 26.07, up to a 7×7 kernel which achieved a very high speedup of 70.56.

In general, increasing the size of square kernels by 2 units results in a linear increase in speedup, approximately doubling each time. For example, when examining the actual execution times for a 4K image, we observe that as the kernel size increases, the execution times for the sequential algorithm skyrocket, whereas the execution times for the parallel algorithm with vectorization increase only slightly compared to the sequential version.

6.3.2 Comparison between two strategies used

In this section, we compare the speedup achieved through vectorization against the non-vectorized approach.

As reported in the previous section, the use of vectorization techniques for convolution computation has led to significant improvements in execution times, particularly with respect to kernel size.

Below are some bar graphs illustrating the remarkable efficiency of the vectorization approach compared to the non-vectorized version. The graphs for 4K and 2K resolutions are presented here. For a comprehensive view of all the graphs, refer to the GitHub repository.

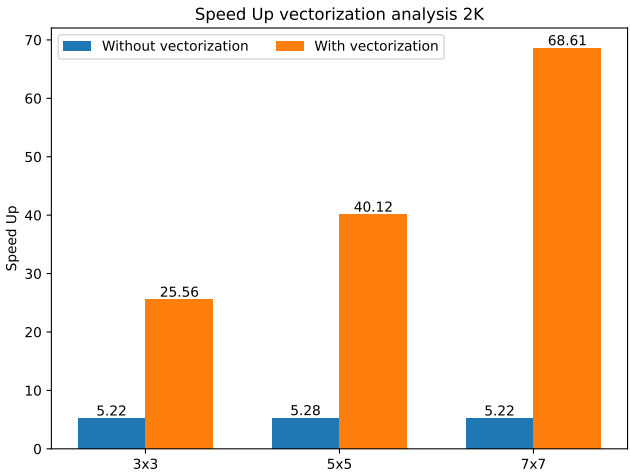

Figure 10. Comparison of speedup achieved using vectorization versus without vectorization, with a 2K image resolution.

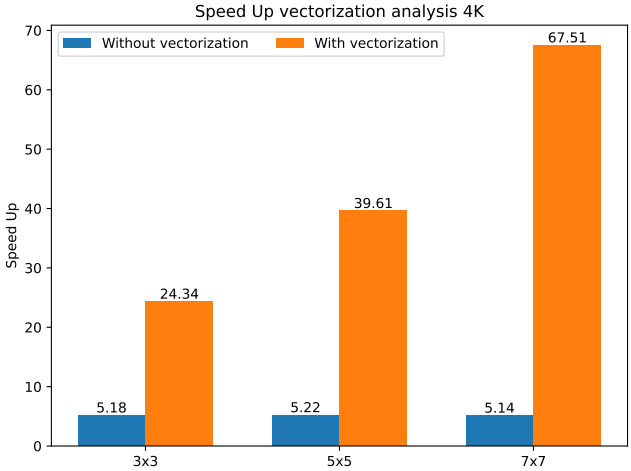

Figure 11. Comparison of speedup achieved using vectorization versus without vectorization, with a 4K image resolution.

As observed in the two graphs 10 and 11, the image size does not impact the speedup significantly. Additionally, the effect of vectorization is markedly evident, as it significantly outperforms the algorithm that does not use vectorization.

Another reason why a significant impact is not observed with the increasing image size could be that larger images cannot be retained in the cache. This results in numerous cache misses, causing frequent transfers of parts of the matrix

from the main memory to the cache. Additionally, an alternative version of the algorithm was implemented, where each process received only the relevant strip of the matrix instead of the entire image. However, this did not result in improvements and, in some cases, slightly worsened performance due to the substantial portion of code required to handle this type of operation. The code for this version can be found in the GitHub repository, and the results were not reported as they were deemed irrelevant.

7. Conclusions

In this project, a convolution algorithm designed for image processing with various kernels was successfully parallelized, effectively applying different types of filters to images. Although Python is not the most suitable tool for intensive image analysis and processing tasks, the adopted parallelization approach has shown promising results.

The implementation of parallelization, particularly through vectorization techniques, has significantly improved the performance of the convolution algorithm. Despite Python's limitations for such tasks, the use of advanced parallel programming techniques allowed for substantial speedups.

The results demonstrated that the parallelized version of the algorithm, with vectorized computations, achieved exceptional performance. This improvement was particularly noticeable with larger kernels, where the benefits of vectorization became increasingly evident.

In summary, while Python may not be the ideal choice for high-performance image processing, the approach taken successfully leveraged parallelism techniques to overcome these limitations. The significant speedup achieved confirms the effectiveness of these techniques and demonstrates that excellent results can be obtained even with less optimal tools.

The project illustrates that, with targeted application of parallelism and vectorization, significant performance improvements can be realized. It shows how effective parallel programming can overcome some of the inherent limitations of the chosen programming environment.