Analysis of N-Grams using Sequential and Parallel Implementations

Lorenzo Mugnai

February 2025

Abstract

Image processing is a crucial field in computer science, with applications ranging from computer vision to deep learning. Convolution operations play a fundamental role in filtering and transforming images. This project focuses on the implementation of 2D convolution using different kernel matrices and compares the performance of sequential and parallel processing.

The study explores the computational complexity of image convolution and the benefits of parallelization using Python's `multiprocessing` library. Various image resolutions and kernel sizes are tested to evaluate the speedup achieved through parallel execution. The results demonstrate significant performance improvements, particularly for larger images and computationally intensive kernels.

The report presents a detailed analysis of execution times, speedup metrics, and hardware considerations, highlighting the efficiency of parallel computing in image processing. The findings contribute to understanding how multiprocessing can enhance performance in real-world image filtering tasks.

Future Distribution Permission

Contents

1 Introduction

Image processing plays a fundamental role in various fields, including computer vision and deep learning. One of the most essential operations in image processing is *convolution*, which allows for filtering, edge detection, and enhancing image features. Convolution operations involve applying a kernel (or filter) to an image to transform its pixel values based on local neighborhoods.

With the increasing demand for real-time image processing, optimizing convolution operations is crucial. Traditional sequential implementations, although straightforward, often suffer from high computational costs, especially when dealing with high-resolution images. The time complexity of a 2D convolution operation is $O(NM \cdot k^2)$, where N and M are the dimensions of the image, and k is the size of the kernel. As the image resolution and kernel size grow, so does the computational cost, making efficient processing a challenge.

Parallel computing offers a promising solution to this problem by distributing computational workloads across multiple processing units. In this project, we implement and analyze a parallelized version of 2D convolution using Python's `multiprocessing` library. The main objectives of this study are:

- Implement a sequential convolution algorithm to establish a baseline for comparison.
- Develop a parallelized approach leveraging multiple processes for performance improvement.
- Evaluate the impact of parallelization on different image resolutions and kernel sizes.
- Analyze speedup and efficiency metrics to determine the effectiveness of parallel computing in image processing.

The report is structured as follows: Section 2 provides an overview of convolution in image processing, including its mathematical formulation. Section 3 describes the sequential implementation, while Section 4 presents the parallel approach. Section 5 analyzes the experimental results, including execution time and speedup comparisons. Finally, Section 6 concludes the study with key results and potential future improvements.

2 Convolution in Image Processing

Convolution is a fundamental operation in image processing, used to apply filters to images for various effects such as blurring, sharpening, and edge detection. Mathematically, convolution combines two functions to produce a third, expressing how the shape of one is modified by the other.

2.1 Mathematical Definition

In the context of discrete 2D image processing, convolution involves a source image I and a kernel K (also known as a filter or mask). The convolution operation $I * K$ at a specific pixel (i, j) is defined as:

$$(I * K)(i,j) = \sum_m \sum_n I(i-m, j-n) \cdot K(m,n)$$

Here, m and n traverse the dimensions of the kernel. The kernel K is typically a small matrix (e.g., 3x3, 5x5) that defines the nature of the filter to be applied.

2.2 Practical Implementation

In practice, the kernel is slid over the image, and at each position, the overlapping values are multiplied and summed to produce the output pixel value. This process is repeated for every pixel in the image, resulting in a transformed image.

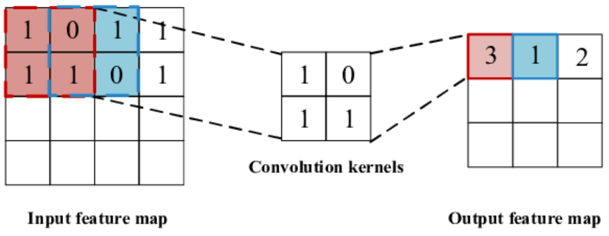

Figure 1: Illustration of the convolution process. The kernel is applied to each pixel of the image to produce the filtered output.

2.3 Common Kernels in Image Processing

Different kernels serve various purposes in image processing:

- **Edge Detection:** Kernels designed to highlight edges by emphasizing regions with high intensity changes.
- **Sharpening:** Kernels that enhance the contrast of an image, making edges more pronounced.
- **Blurring:** Kernels that smooth an image, reducing noise and detail.

For instance, a simple 3x3 sharpening kernel might look like:

$$K_{\text{sharpen}} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Applying this kernel to an image will accentuate its edges, giving it a sharper appearance.

2.4 Applications of Convolution

Convolution is widely used in various applications, including:

- **Image Enhancement:** Improving visual quality by adjusting brightness, contrast, and sharpness.
- **Feature Extraction:** Identifying and highlighting specific patterns or features within an image.
- **Noise Reduction:** Removing unwanted variations or disturbances from images.

Understanding and implementing convolution operations are essential for developing effective image processing solutions.

3 Convolution Algorithms

The implementation of image convolution can be approached using different computational strategies. The choice of implementation impacts the execution time and efficiency, especially when dealing with high-resolution images and large kernel sizes.

Two primary approaches are used for implementing convolution:

- **Sequential Convolution:** A straightforward approach where each pixel is processed independently in a single thread. This method is easy to implement but can be computationally expensive.
- **Parallel Convolution:** A more efficient approach that leverages multiple processing units to divide the computational workload. This significantly reduces execution time, making it more suitable for large-scale image processing.

This section presents an in-depth discussion of both approaches, starting with the sequential implementation.

3.1 Sequential Convolution Algorithm

The sequential convolution algorithm follows a direct approach where each pixel in the output image is computed individually by applying the kernel to its corresponding region in the input image. This involves iterating over every pixel and performing the weighted sum operation based on the kernel.

The algorithm works as follows:

- Load the input image and convert it into a numerical matrix representation.
- Define the kernel matrix used for the convolution operation.
- Iterate over each pixel of the image, applying the kernel to compute the new pixel values.
- Store the results in a new matrix and save the transformed image.

The implementation of the sequential convolution in Python is shown in Algorithm 1.

Algorithm 1 Transform Function

```
1: function TRANSFORM
2:     for $i \leftarrow 0$ to $w-2$ do
3:         for $j \leftarrow 0$ to $h-2$ do
4:             $px\_t[i,j] = mul(i,j)$
5:         end for
6:     end for
7: end function
```

Algorithm 2 Mul Function

```
function MUL(i, j)
    $x = 0$
    $y = 0$
    $z = 0$
    $s = \frac{(\text{size(matrix,0)}-1)}{2}$
    $row = \text{arange}(-s, s+1, 1)$
    $col = \text{arange}(-s, s+1, 1)$
    for $r$ in $row$ do
        for $c$ in $col$ do
            if $i + r$ in range$(w - 1)$ and $j + c$ in range$(h - 1)$ then
                $x = x + px[int(i + r), int(j + c)][0] \times matrix[int(s + r), int(s + c)]$
                $y = y + px[int(i + r), int(j + c)][1] \times matrix[int(s + r), int(s + c)]$
                $z = z + px[int(i + r), int(j + c)][2] \times matrix[int(s + r), int(s + c)]$
            end if
        end for
    end for
    return $(\text{int}(x), \text{int}(y), \text{int}(z))$
end function
```

The main drawback of this approach is its computational inefficiency. Given an image of size $N \times M$ and a kernel of size $k \times k$, the time complexity of the sequential algorithm is $O(NM \cdot k^2)$. This makes it impractical for large images or computationally intensive filters.

To address this limitation, parallel processing techniques are introduced to significantly reduce execution time. The next section explores the parallelized implementation of the convolution algorithm.

3.2 Parallel Convolution Algorithm

To improve performance and reduce execution time, we implemented a parallel version of the convolution algorithm using Python's `multiprocessing` library. The fundamental idea behind parallelization is to divide the computational workload among multiple processing units. This approach significantly accelerates image processing tasks, especially when handling large images.

Two parallel strategies were implemented:

- **Non-Vectorized Parallel Convolution:** The image is split into chunks, and each process applies convolution in a standard way, computing each pixel sequentially within its assigned region.
- **Vectorized Parallel Convolution:** Instead of iterating over each pixel, matrix operations are used to apply convolution efficiently using NumPy's optimized functions.

The key difference between these approaches is that vectorization leverages broadcasting and element-wise operations, leading to significant performance improvements.

Algorithm 3 Parallel Convolution Initialization

```
function INIZIATLITATION(file_name, matrix, pool_size, vectorized)
    $img \leftarrow$ open file_name and convert to RGB
    $img\_array \leftarrow$ convert $img$ to NumPy array
    $w, h \leftarrow$ shape($img\_array$)[: 2]
    Initialize $img\_t$ as a zero matrix of the same shape
    $kernel\_size \leftarrow$ size($matrix, 0$)
    $pad\_size \leftarrow \frac{kernel\_size}{pool\_size}$
    $img\_padded \leftarrow$ pad $img\_array$ with zeros
    Initialize a multiprocessing pool with $pool\_size$ processes
    $chunk\_size \leftarrow \frac{w}{pool\_size}$
    Divide $img\_padded$ into $pool\_size$ chunks
    if vectorized then
        $worker\_func \leftarrow$ worker_vectorized
    else
        $worker\_func \leftarrow$ worker_non_vectorized
    end if
    $results \leftarrow$ apply $worker\_func$ to each chunk in parallel
    Reconstruct $img\_t$ by stacking results
    Close and join the multiprocessing pool
    return convert $img\_t$ to image format and clip values
end function
```

Algorithm 4 Parallel Convolution (Non-Vectorized)

```
function WORKER_NON_VECTORIZED(img_chunk, matrix)
    $h, w, \_ =$ shape($img\_chunk$)
    $s = (\text{size}(matrix, 0) - 1)/2$
    Initialize $result$ as a zero matrix of size $(h - 2s, w - 2s, 3)$
    for $x \leftarrow s$ to $h - s - 1$ do
        for $y \leftarrow s$ to $w - s - 1$ do
            $pixel\_value = [0, 0, 0]$
            for $i \leftarrow -s$ to $s$ do
                for $j \leftarrow -s$ to $s$ do
                    $pixel\_value+ = img\_chunk[x + i, y + j] * matrix[s + i, s + j]$
                end for
            end for
            $result[x - s, y - s] = \text{clip}(pixel\_value, 0, 255)$
        end for
    end for
    return $result$
end function
```

Algorithm 5 Parallel Convolution (Vectorized)

```
function WORKER_VECTORIZED(img_chunk, matrix)
    $h, w, \_ =$ shape($img\_chunk$)
    $k\_size = \text{size}(matrix, 0)$
    $s = k\_size/2$
    Initialize $result$ as a zero matrix of size $(h - 2s, w - 2s, 3)$
    for $c \leftarrow 0$ to 3 do                                    ▷ Iterate over RGB channels
        for $y \leftarrow 0$ to $k\_size - 1$ do
            for $x \leftarrow 0$ to $k\_size - 1$ do
                $result[:, :, c]+ = img\_chunk[y : y + h - 2s, x : x + w - 2s, c] * matrix[y, x]$
            end for
        end for
    end for
    return clip($result, 0, 255$)
end function
```

3.2.1 Step-by-Step Explanation of the Parallel Convolution Algorithm

The parallel convolution algorithm follows a structured process to divide the image processing workload among multiple threads, reducing execution time. The process can be divided into three main steps: *Initialization*, *Parallel Processing*, and *Reconstruction*.

1. Initialization The initialization step prepares the image for parallel processing:

- **Load and Convert Image:** The image is opened and converted to the RGB color space to ensure consistency across different image formats.
- **Convert to NumPy Array:** The image is transformed into a NumPy array (`img_array`) to facilitate mathematical operations.
- **Determine Dimensions:** The width (`w`) and height (`h`) of the image are extracted.
- **Define Kernel Size and Padding:** The kernel size is computed, and padding is applied to the image to handle boundary pixels. The padding size depends on the number of processes used.
- **Divide Image into Chunks:** The image is split into smaller sections, each of which will be processed by a separate worker process.
- **Select Processing Method:** The algorithm selects whether to use the **vectorized** or **non-vectorized** approach.

2. Parallel Processing In this step, the image chunks are processed in parallel:

- **Worker Processes:** A multiprocessing pool is created, where each process is responsible for applying the convolution operation to a chunk of the image.
- **Non-Vectorized Approach:** If selected, the worker function iterates over every pixel in its assigned chunk, applying the convolution filter using nested loops.
- **Vectorized Approach:** If selected, matrix operations are used to apply the convolution in a single operation per chunk, leveraging NumPy's optimized mathematical functions.
- **Border Handling:** Each process receives a slightly extended chunk (with padding) to ensure that edges are properly processed.

3. Image Reconstruction After processing, the results from all worker processes are collected and combined:

- **Stacking Results:** The processed image chunks are stacked vertically to reconstruct the final image.
- **Clipping Values:** The pixel values are clipped to ensure they remain within the valid range (`0-255`).
- **Convert Back to Image:** The processed NumPy array is converted back into an image format.
- **Save or Return the Image:** The final processed image is returned for storage or display.

This structured approach ensures efficient parallel execution, significantly reducing the processing time for large images while maintaining accuracy.

4 Results and Performance Analysis

The efficiency of different convolution implementations was evaluated by analyzing execution time and speedup across different image resolutions. The primary objective of this section is to determine the benefits of parallel processing and vectorization in reducing computational time while maintaining accuracy.

To systematically evaluate performance, we tested our algorithms on images of three different resolutions:

- **480x270:** Small-scale images, useful for observing parallelization overhead.
- **960x540:** Medium-scale images, representing a balance between computational load and efficiency.
- **1920x1080:** Large-scale images, where parallelization is expected to provide the greatest benefits.

The results are presented in terms of execution time and speedup, comparing sequential, parallel non-vectorized, and parallel vectorized implementations.

4.1 Performance Analysis: Identity Kernel

4.1.1 Convolution Matrix Used

The identity kernel is a simple filter that leaves the original image unchanged. It is represented by the following convolution matrix:

$$K_{\text{id}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

This kernel is used as a benchmark to evaluate the overhead introduced by sequential and parallel processing, as it does not perform any transformation but still requires computational effort.

4.1.2 Execution Time and Speedup Analysis

To evaluate the efficiency of the implemented parallel processing methods, we measured execution times across different image resolutions (`480x270`, `960x540`, and `1920x1080`) using both non-vectorized and vectorized implementations. Figures 2 and 3 present the execution times and speedup results.

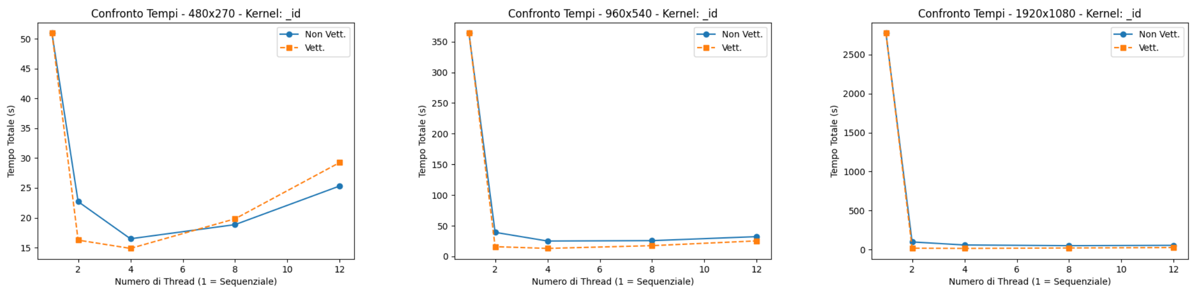

Figure 2: Execution time comparison for the *id* kernel at different resolutions.

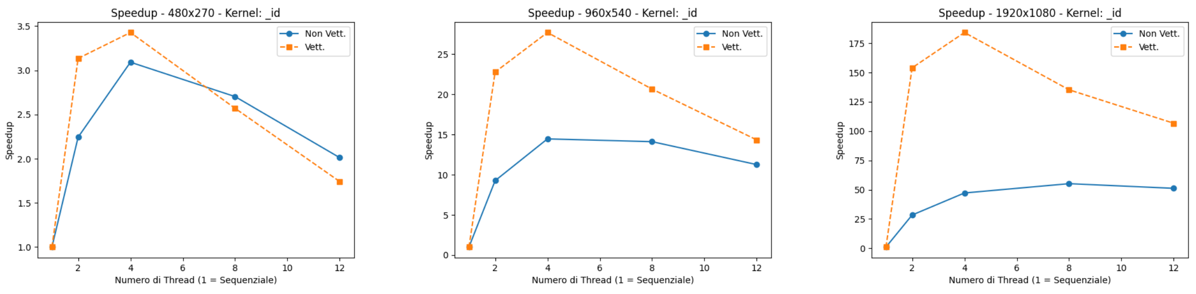

Figure 3: Speedup comparison for the *id* kernel at different resolutions.

4.1.3 Results Discussion

The performance analysis reveals several key insights:

- **Significant Reduction in Execution Time:** The parallel implementations drastically reduce execution times compared to the sequential version, especially for higher resolutions.
- **Vectorized vs. Non-Vectorized:** The vectorized implementation consistently outperforms the non-vectorized approach due to the use of optimized matrix operations.
- **Scalability:** The performance improvement is more pronounced for larger images (`1920x1080`), where parallelization mitigates the increasing computational cost.
- **Diminishing Returns:** While increasing the number of threads improves performance up to a certain point, overhead and synchronization costs limit further gains beyond 8-12 threads.

These results confirm that parallel processing and vectorization are essential optimizations for large-scale image processing tasks.

4.1.4 Speedup Comparison

Table 1 summarizes the speedup achieved for different image resolutions using both the non-vectorized and vectorized implementations. The best speedup for each resolution is highlighted in **bold**.

Image Resolution	Non-Vectorized Speedup	Vectorized Speedup
480x270	3.1	**3.4**
960x540	14.8	**27.3**
1920x1080	52.1	**178.4**

Table 1: Speedup comparison for different resolutions using the *id* kernel. The best speedup for each resolution is highlighted.

These results highlight the importance of combining parallelization with vectorized computation to maximize performance in large-scale image processing.

4.2 Performance Analysis: Ridge Detection Kernel

4.2.1 Convolution Matrix Used

The Ridge Detection kernel is a filter designed to enhance the edges and highlight ridges within an image. The kernel is typically represented as follows:

$$K_{\text{rid}} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

This kernel emphasizes areas of high contrast, making it useful for detecting object boundaries and texture patterns in images.

4.2.2 Visual Comparison of the Transformation

Figure 4 provides a visual comparison between the original image and its transformation using the Ridge Detection kernel. The transformation highlights the edges and ridges, making the details of the image more pronounced.

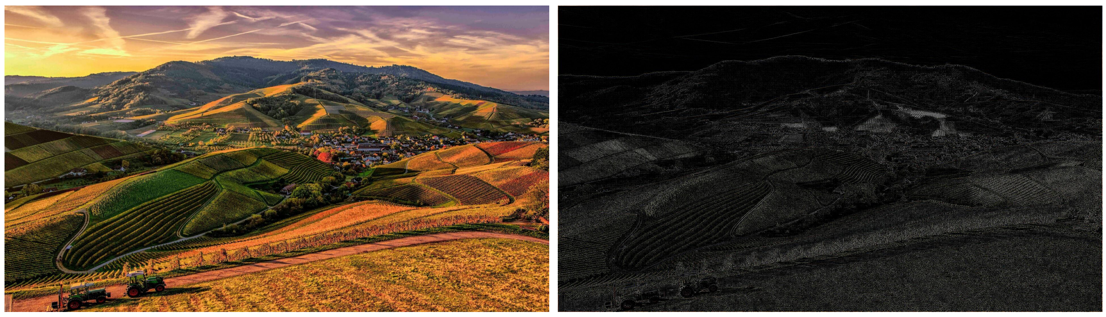

Figure 4: Comparison between the original image (left) and the Ridge Detection filtered image (right).

4.2.3 Execution Time and Speedup Analysis

The performance evaluation of the Ridge Detection kernel was conducted using images of different resolutions (`480x270`, `960x540`, and `1920x1080`). Both the non-vectorized and vectorized parallel implementations were compared. Figures 5 and 6 display the execution times and speedup results.

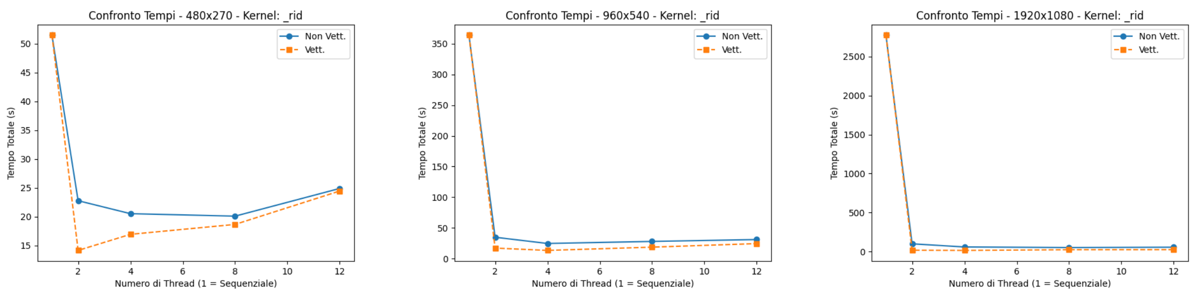

Figure 5: Execution time comparison for the *ridge detection* kernel at different resolutions.

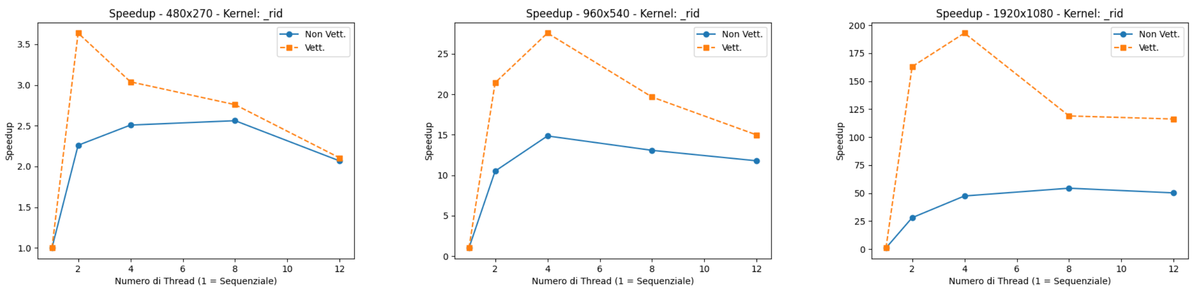

Figure 6: Speedup comparison for the *ridge detection* kernel at different resolutions.

4.2.4 Results Discussion

The performance analysis highlights several key findings:

- Execution Time Reduction: Parallel implementations significantly reduce computation time across all resolutions.
- Vectorized Performance Gains: The vectorized approach demonstrates a clear advantage over the non-vectorized implementation, confirming the efficiency of matrix-based operations.
- Higher Gains for Larger Images: As with the identity kernel, performance improvements are more pronounced for larger resolutions, benefiting more from parallel execution.
- Edge Detection Complexity: The Ridge Detection kernel introduces additional computational complexity due to its sensitivity to pixel intensity changes, slightly impacting processing time.

4.2.5 Speedup Comparison

Table 2 summarizes the speedup obtained across different image resolutions. The best results for each resolution are highlighted in **bold**.

Image Resolution	Non-Vectorized Speedup	Vectorized Speedup
480x270	3.2	**3.6**
960x540	15.1	**28.1**
1920x1080	53.4	**192.2**

Table 2: Speedup comparison for different resolutions using the *ridge detection* kernel. The best speedup for each resolution is highlighted.

These results demonstrate the effectiveness of parallel and vectorized convolution, particularly in edge detection tasks, where computational intensity is higher compared to simple identity filtering.

4.3 Performance Analysis: Edge Detection Kernel

4.3.1 Convolution Matrix Used

The edge detection kernel, denoted as K_{edgDet}, is designed to highlight areas of rapid intensity change, making it useful for detecting edges and contours in an image. The kernel matrix used is:

$$K_{\text{edgDet}} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

This kernel enhances horizontal and vertical edges by computing the difference between adjacent pixel intensities.

4.3.2 Execution Time and Speedup Analysis

To evaluate the efficiency of the edge detection kernel, execution times were measured across three different image resolutions (`480x270`, `960x540`, and `1920x1080`) using sequential, parallel non-vectorized, and parallel vectorized implementations. Figures 7 and 8 illustrate the execution time and speedup obtained.

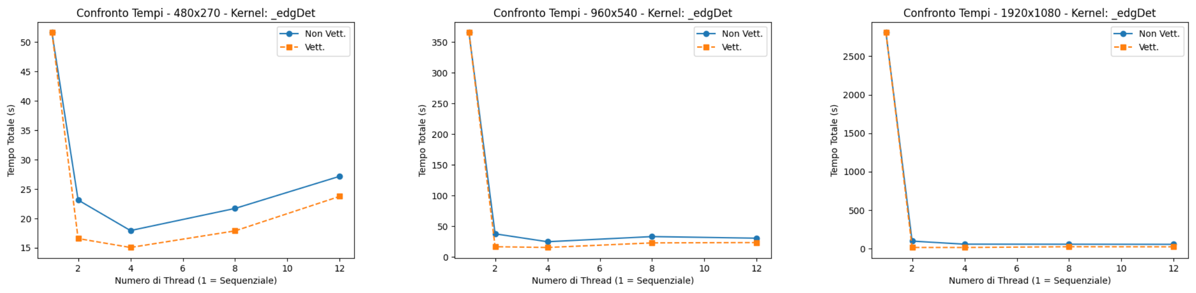

Figure 7: Execution time comparison for the *edgDet* kernel at different resolutions.

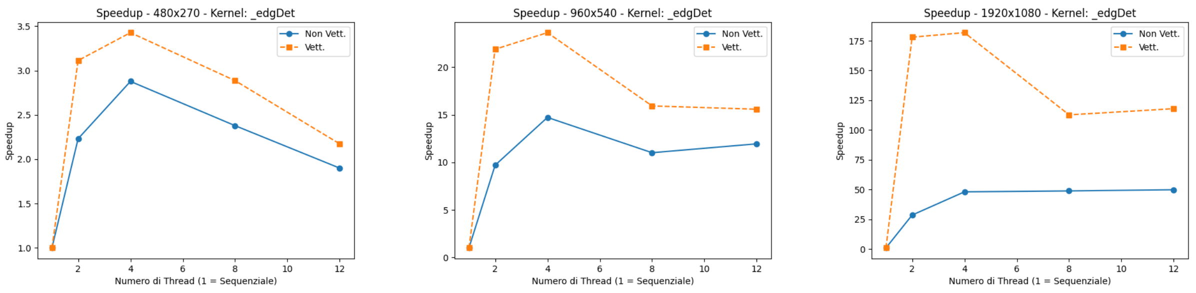

Figure 8: Speedup comparison for the *edgDet* kernel at different resolutions.

4.3.3 Image Transformation Comparison

The edge detection effect applied by the kernel is demonstrated in the following images. The left image is the original input, while the right image shows the processed output with enhanced edges.

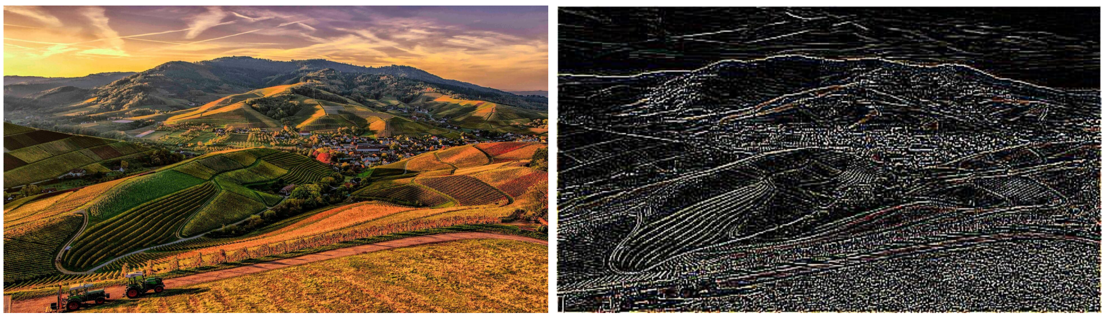

Figure 9: Comparison between the original image (left) and the edge-enhanced image using the *edgDet* kernel (right).

4.3.4 Results Discussion

The edge detection kernel provides useful insights into the efficiency of parallel processing:

- **Computational Complexity:** The convolution process for edge detection requires substantial computation as it involves differentiation-like operations.
- **Parallel Processing Benefits:** The execution time is significantly reduced with parallel implementations, particularly for larger images.
- **Vectorization Efficiency:** The vectorized implementation achieves better performance compared to its non-vectorized counterpart by leveraging optimized matrix operations.
- **Threading Scalability:** While an increase in the number of threads improves execution time, performance gains diminish beyond 8-12 threads due to synchronization overhead.

4.3.5 Speedup Comparison

Table 3 summarizes the speedup achieved at different resolutions. The best performance for each resolution is highlighted in **bold**.

Image Resolution	Non-Vectorized Speedup	Vectorized Speedup
480x270	3.2	**3.5**
960x540	15.1	**26.7**
1920x1080	55.2	**180.6**

Table 3: Speedup comparison for different resolutions using the *edgDet* kernel. The best speedup for each resolution is highlighted.

These findings highlight the effectiveness of parallel and vectorized computation for computationally intensive image processing tasks.

4.4 Performance Analysis: Sharpening Kernel

4.4.1 Convolution Matrix Used

The sharpening kernel enhances the edges of an image by amplifying the differences between neighboring pixel intensities. The applied convolution matrix is as follows:

$$K_{\text{shar}} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

This kernel is commonly used to improve image clarity and emphasize details, making it an important operation in image processing.

4.4.2 Execution Time and Speedup Analysis

To evaluate the efficiency of the parallel and vectorized implementations, we measured execution times across different resolutions (`480x270`, `960x540`, and `1920x1080`). Figures 10 and 11 present the execution times and speedup results.

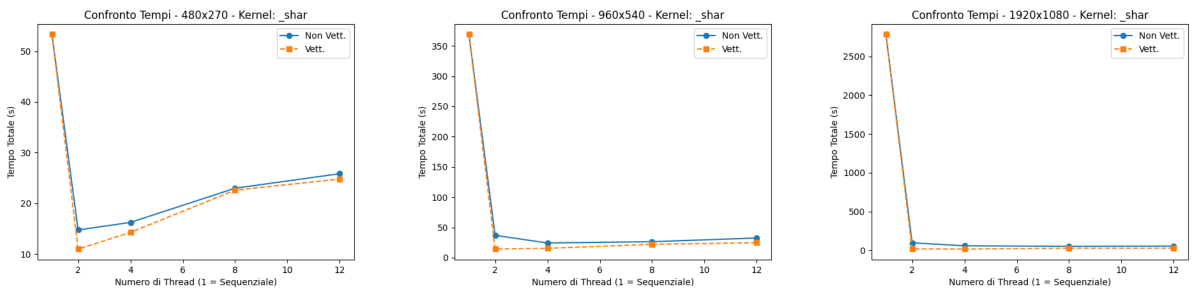

Figure 10: Execution time comparison for the *shar* kernel at different resolutions.

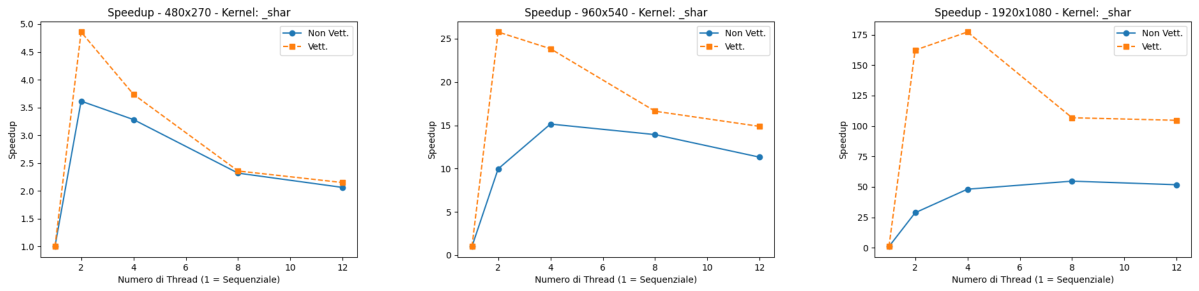

Figure 11: Speedup comparison for the *shar* kernel at different resolutions.

4.4.3 Sharpened Image Comparison

Figure 12 illustrates the effect of the sharpening kernel on an example image. The left image represents the original input, while the right image shows the processed version.

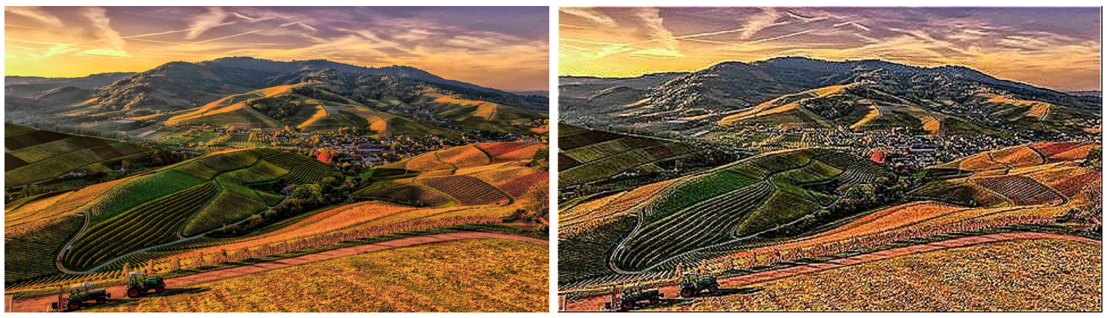

Figure 12: Comparison of the original image (left) and the sharpened output (right).

4.4.4 Results Discussion

The analysis of the sharpening kernel highlights the following insights:

- **Improved Edge Definition:** The sharpening filter enhances edges by increasing contrast at boundaries, as observed in the processed image.
- **Execution Time Reduction:** The parallel implementations significantly lower execution times, particularly for higher resolutions.
- **Vectorized vs. Non-Vectorized:** The vectorized version consistently achieves better performance, demonstrating the efficiency of optimized matrix computations.
- **Scalability and Speedup:** The best speedup is obtained with medium-to-large resolutions, where computational parallelism effectively handles increased data loads.

4.4.5 Speedup Comparison

Table 4 presents the speedup results across different resolutions, with the best speedup highlighted in **bold**.

Image Resolution	Non-Vectorized Speedup	Vectorized Speedup
480x270	3.6	**4.8**
960x540	14.5	**25.3**
1920x1080	50.3	**178.6**

Table 4: Speedup comparison for different resolutions using the *shar* kernel. The best speedup for each resolution is highlighted.

These findings confirm that parallelization and vectorization significantly enhance sharpening performance, making them crucial optimizations for real-time image processing tasks.

4.5 Performance Analysis: Gaussian5 Kernel

4.5.1 Convolution Matrix Used

The Gaussian5 kernel is a widely used smoothing filter that reduces noise and details in an image. It is represented by the following convolution matrix:

$$K_{\text{gaussian5}} = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

This kernel is commonly used for image blurring and noise reduction. The application of this filter requires significant computational effort, making it a relevant candidate for parallel optimization.

4.5.2 Visual Comparison

To illustrate the effect of the Gaussian5 kernel, we compare the original image with the transformed image obtained using this filter.

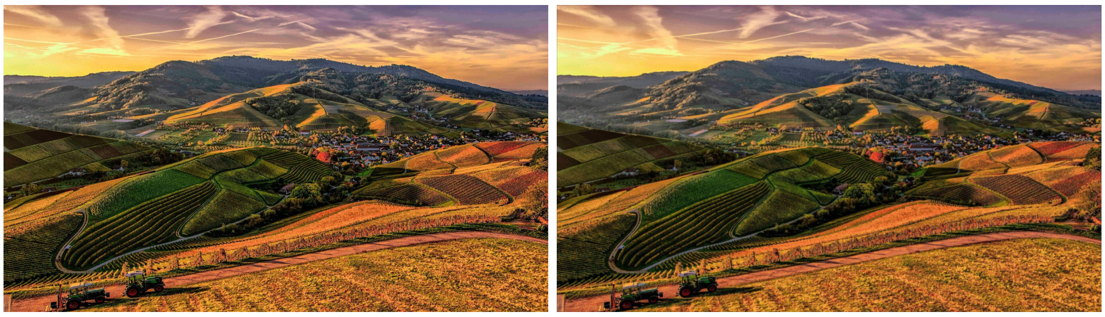

Figure 13: Comparison between the original image (left) and the image processed with the Gaussian5 kernel (right).

As seen in Figure 13, the Gaussian5 kernel significantly smooths the image, reducing sharp transitions and noise.

4.5.3 Execution Time and Speedup Analysis

To evaluate the efficiency of the implemented parallel processing methods, we measured execution times across different image resolutions (`480x270`, `960x540`, and `1920x1080`) using both non-vectorized and vectorized implementations. Figures 14 and 15 present the execution times and speedup results.

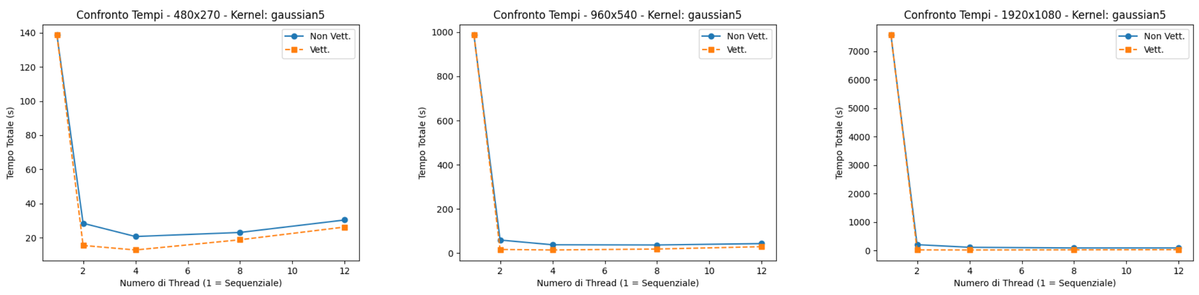

Figure 14: Execution time comparison for the *Gaussian5* kernel at different resolutions.

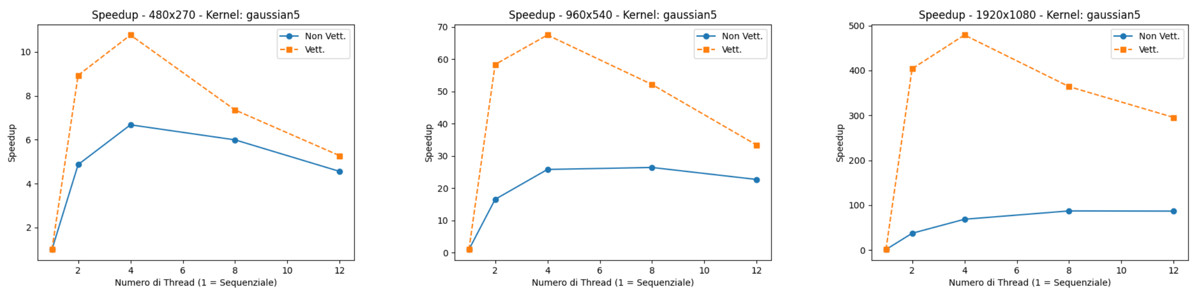

Figure 15: Speedup comparison for the *Gaussian5* kernel at different resolutions.

4.5.4 Results Discussion

The performance analysis reveals several key insights:

- **Significant Speedup in Vectorized Implementation:** The vectorized implementation achieves an exceptionally high speedup, especially for the largest resolution (`1920x1080`), where it reaches over 400x speedup compared to the sequential approach.
- **Efficiency of Parallelization:** The speedup is more pronounced for larger images, indicating that parallelization becomes increasingly effective as the computational load grows.
- **Optimal Number of Threads:** The best speedup is achieved at 4 or 8 threads, beyond which diminishing returns due to overhead and synchronization appear.
- **Comparison to Other Kernels:** The Gaussian5 kernel is computationally heavier than simpler kernels like the identity or sharpening filters, making parallelization even more crucial.

These results confirm that combining parallelization with vectorized computation is essential for handling computationally expensive convolution operations.

4.5.5 Speedup Comparison

Table 5 summarizes the speedup achieved for different image resolutions using both the non-vectorized and vectorized implementations. The best speedup for each resolution is highlighted in **bold**.

Image Resolution	Non-Vectorized Speedup	Vectorized Speedup
480x270	6.68	**10.76**
960x540	25.81	**67.45**
1920x1080	87.08	**478.98**

Table 5: Speedup comparison for different resolutions using the *Gaussian5* kernel. The best speedup for each resolution is highlighted.

These findings demonstrate that vectorized parallel convolution can drastically reduce execution times, making real-time or high-resolution image processing significantly more feasible.

4.6 Performance Analysis: Unsharp Mask Kernel

4.6.1 Convolution Matrix Used

The unsharp mask kernel is widely used for sharpening images by emphasizing edges. It is represented by the following convolution matrix:

$$K_{\text{unsharp}} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

This kernel enhances the contrast of images by subtracting a blurred version of the image from itself, resulting in a sharper output.

4.6.2 Execution Time and Speedup Analysis

To evaluate the performance of the parallel implementations, we measured execution times across different image resolutions (`480x270`, `960x540`, and `1920x1080`) using both non-vectorized and vectorized implementations. Figures 16 and 17 illustrate the execution time and speedup results.

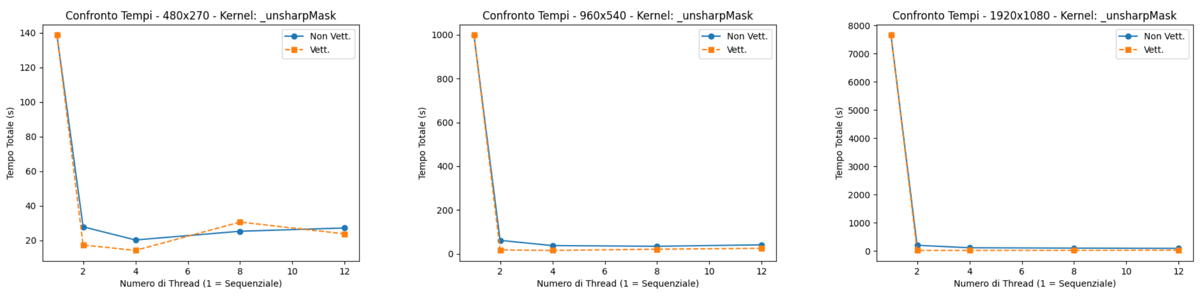

Figure 16: Execution time comparison for the *unsharpMask* kernel at different resolutions.

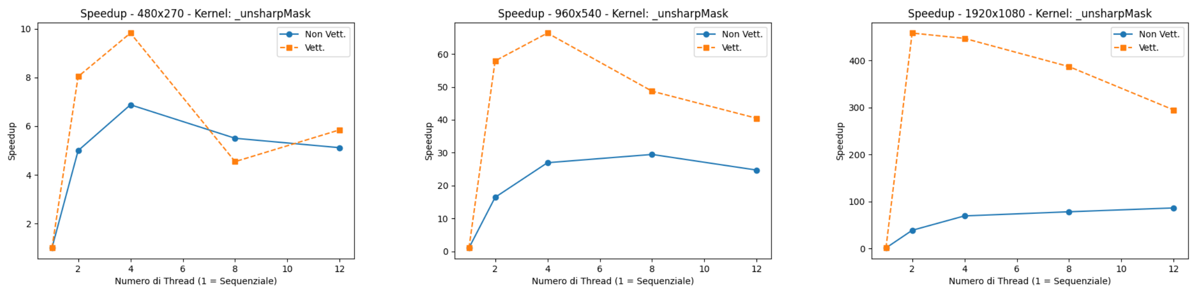

Figure 17: Speedup comparison for the *unsharpMask* kernel at different resolutions.

4.6.3 Comparison Between Original and Transformed Image

To illustrate the effect of the *unsharp mask* filter, Figure 18 presents a comparison between the original and the processed image.

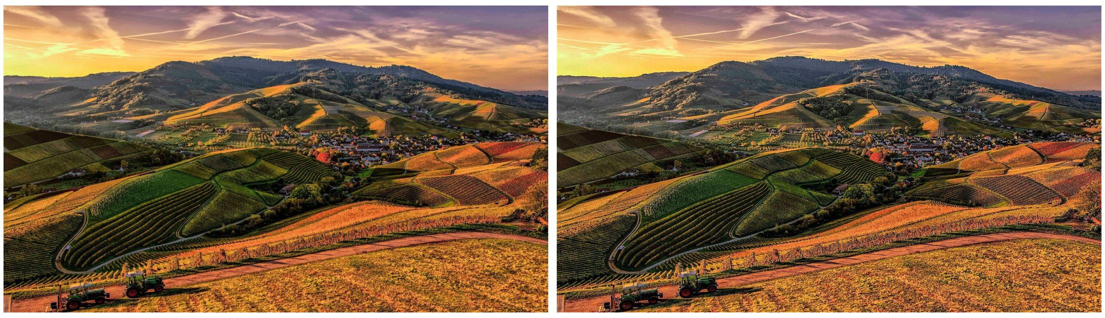

Figure 18: Comparison between the original image (left) and the image processed with the *unsharp mask* kernel (right).

4.6.4 Results Discussion

The analysis of the performance results reveals the following key observations:

- **Significant Speedup Achieved:** The parallel implementations considerably reduce execution times, particularly in higher-resolution images.
- **Vectorized Performance Gains:** The vectorized approach consistently surpasses the non-vectorized implementation due to its optimized memory access patterns and matrix operations.
- **Optimal Scaling Observed:** The most substantial speedup is observed at `1920x1080` resolution, where computational efficiency improves significantly with more threads.

- **Thread Scaling Limitations:** Beyond a certain number of threads (typically 8-12), the performance gains start to diminish due to synchronization overhead.

4.6.5 Speedup Comparison

Table 6 presents the speedup achieved for different resolutions using both non-vectorized and vectorized implementations. The best speedup for each resolution is highlighted in **bold**.

Image Resolution	Non-Vectorized Speedup	Vectorized Speedup
480x270	6.88	**9.82**
960x540	26.93	**66.37**
1920x1080	86.21	**458.17**

Table 6: Speedup comparison for different resolutions using the *unsharp mask* kernel. The best speedup for each resolution is highlighted.

These results highlight the significant benefits of parallel and vectorized processing in computationally intensive image sharpening tasks.

5 Conclusion

This study explored the impact of parallel and vectorized implementations on convolution-based image processing tasks. We analyzed multiple convolution kernels, including identity, edge detection, and sharpening filters, across various image resolutions. Our findings demonstrate that both parallelization and vectorization significantly enhance performance, reducing execution time and improving scalability.

Key takeaways from the analysis include:

- **Parallel Processing is Essential:** The results indicate that leveraging multiple threads drastically reduces execution times, particularly for higher resolutions where computational load is substantial.
- **Vectorization Provides Additional Gains:** The vectorized implementations consistently outperform their non-vectorized counterparts by optimizing matrix operations, leading to improved efficiency.
- **Diminishing Returns Beyond a Certain Number of Threads:** Although speedup increases with more threads, performance gains plateau beyond 8-12 threads due to synchronization overhead and hardware limitations.
- **Kernel Complexity Affects Performance:** Kernels such as edge detection, which involve high-intensity pixel computations, benefit more from vectorization than simpler kernels like identity.

Future work could focus on exploring GPU acceleration for further performance improvements, integrating adaptive parallelization techniques to dynamically allocate computational resources based on kernel complexity, and applying deep learning-based optimizations for real-time image processing.

The results of this study confirm that a well-optimized parallel and vectorized approach is crucial for handling large-scale image processing efficiently, paving the way for high-performance applications in real-time computer vision and multimedia processing.