Kernel Image Processing

Parallelizazion in Python

Mirko Bicchierai

Firenze 05/07/2024

Convolutions

Introdutction

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

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

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

Example of Convolutions

Compute a convolution

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.

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

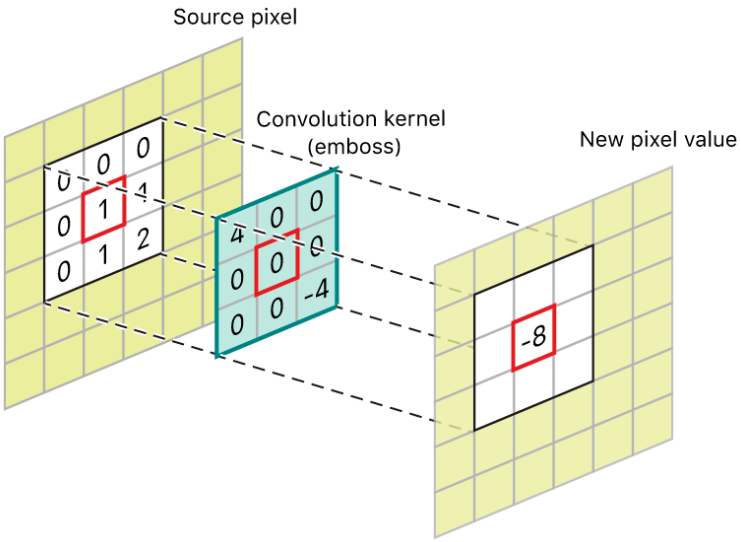


Image processing

Convolution Example

- It can be used both as preprocessing tool or a image manipulation.
- The effect of the convolution on the input image depend on the kernel used.

Image and Kernels used for testing

Quality of the images and dimension of kernels

•For the tests conducted regarding the obtained performance, images of different quality were used:

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

•All these image qualities were tested for convolutions with various kernels; however, in the end, three were chosen.

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

Prewitt horizontal kernel (3x3)

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

Gaussina Blur kernel (5x5)

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

Gaussian Blur kernel (7x7)

Hardware used and testing performance method

•All the 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).

•All the results reported below regarding the obtained speedup are calculated as the average of 100 executions of the sequential algorithm divided by the average of 100 executions of the parallel algorithm. This approach is used to ensure the most accurate results possible.

•For each image quality tested, 5 different images were used, and the results were averaged. This approach was taken to further improve the accuracy and reliability of the final results.

Implementation

Sequential algorithm (naive)

```python
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, a_min: 0, a_max: 255)
    return result.astype(np.uint8)
```

The input image is represented as a matrix across the three RGB channels. Additionally, zero padding was applied to the edges of the image beforehand; otherwise, the convolution would not be defined at the borders.

Convolutions

Parallelization strategy

In this way, each process handles and computes the convolution for a single row of the image. Subsequently, the output from each process needs to be reassembled.

Implementation

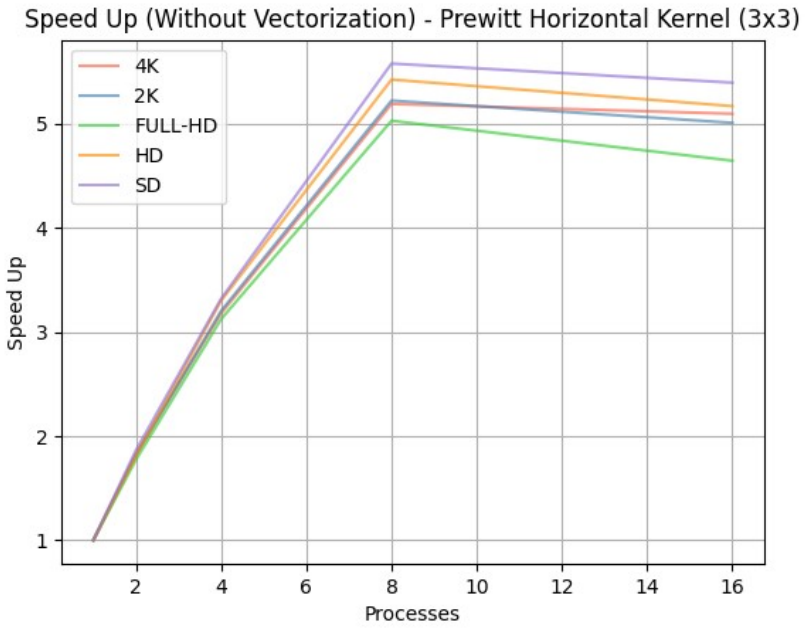
Parallel algorithm

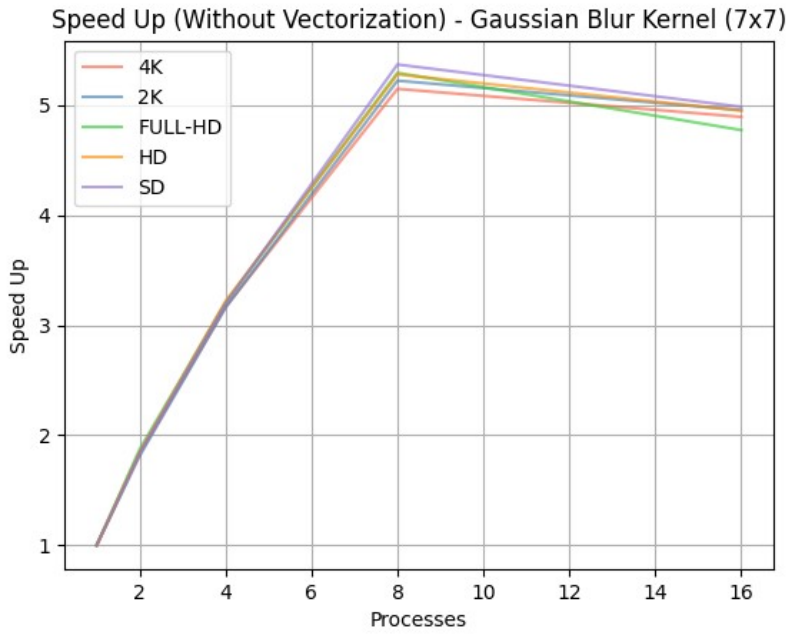
```
def parallel_apply_convolution_normal(img, kernel, num_workers, height, width, pad_y, pad_x):
    chunk_size = height // num_workers
    chunks = [(img, kernel, i * chunk_size, (i + 1) * chunk_size if i < num_workers - 1 else height, height, width,
               pad_y, pad_x) for i in
              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, a_min: 0, a_max: 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 += img[i + pad_y + dy, j + pad_x + dx] * kernel[pad_y + dy, pad_x + dx]
            result[i - start_row, j] = convolved_value
    return result
```

•This implementation reduces the cost from O(h*w*hK*wK) to O(h/np * w * hK * wK), where “np” represents the number of parallel processes. Padding is not taken into account because its size is minimal.

Convolutions Results

SpeedUp Obtained





•The speedup achieved is stable with respect to the kernel size (though results for the 5x5 kernel are not shown). This outcome was expected since the parallelization does not involve the kernel loops.
•However, as the kernel size increases, the speedup tends to stabilize across all tested image dimensions.

Vectorization

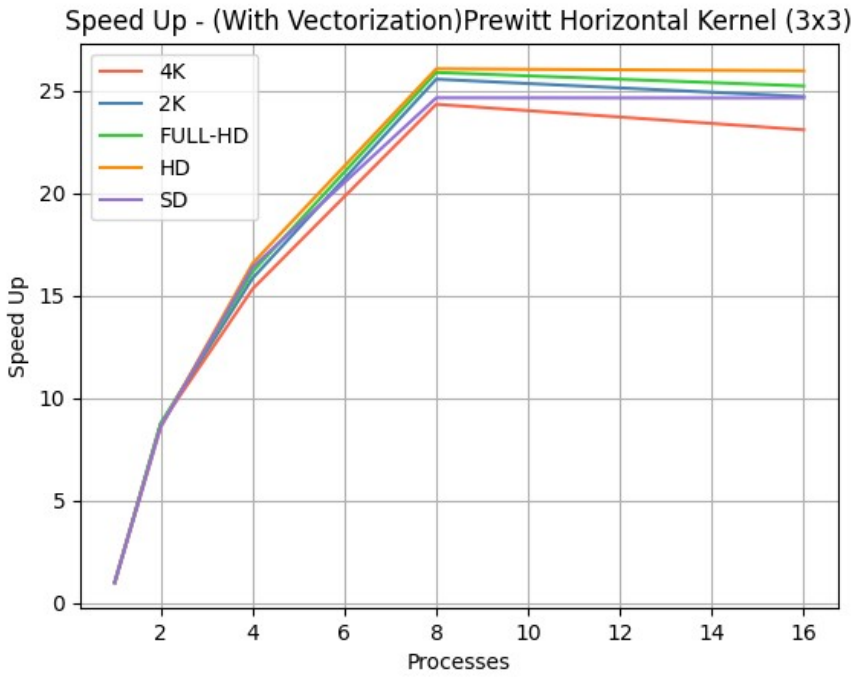
Parallel algorithm with vectorization for computing the convolution

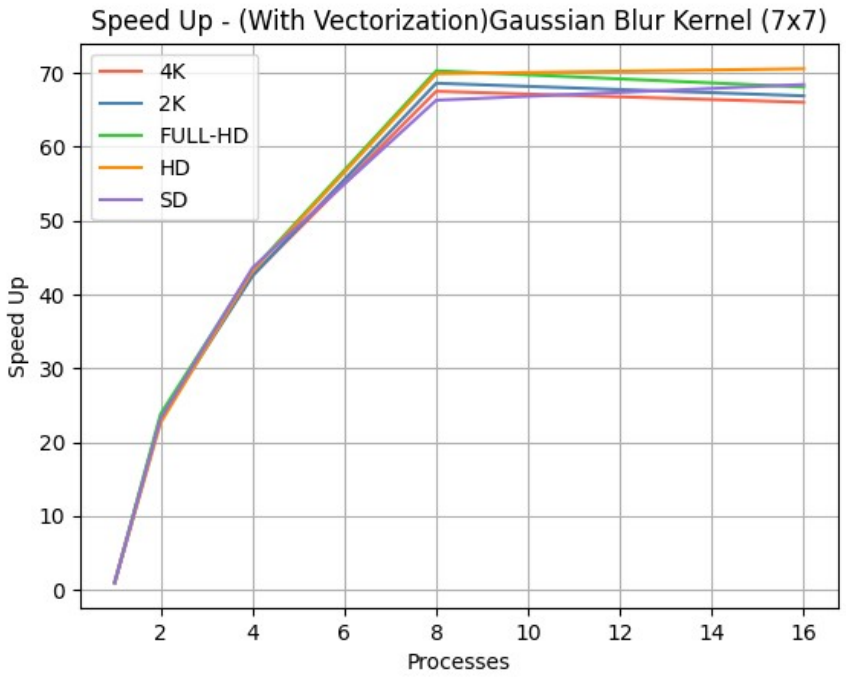
```
def parallel_apply_convolution(img, kernel, num_workers, height, width, pad_y, pad_x):
    chunk_size = height // num_workers
    chunks = [(img, kernel, i * chunk_size, (i + 1) * chunk_size if i < num_workers - 1 else height, height, width,
               pad_y, pad_x) for i in
              range(num_workers)]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

- This implementation leverages the NumPy library, which operates under the hood with optimized C code. As we will see, this results in significant performance advantages.

Vectorization Results

SpeedUp Obtained

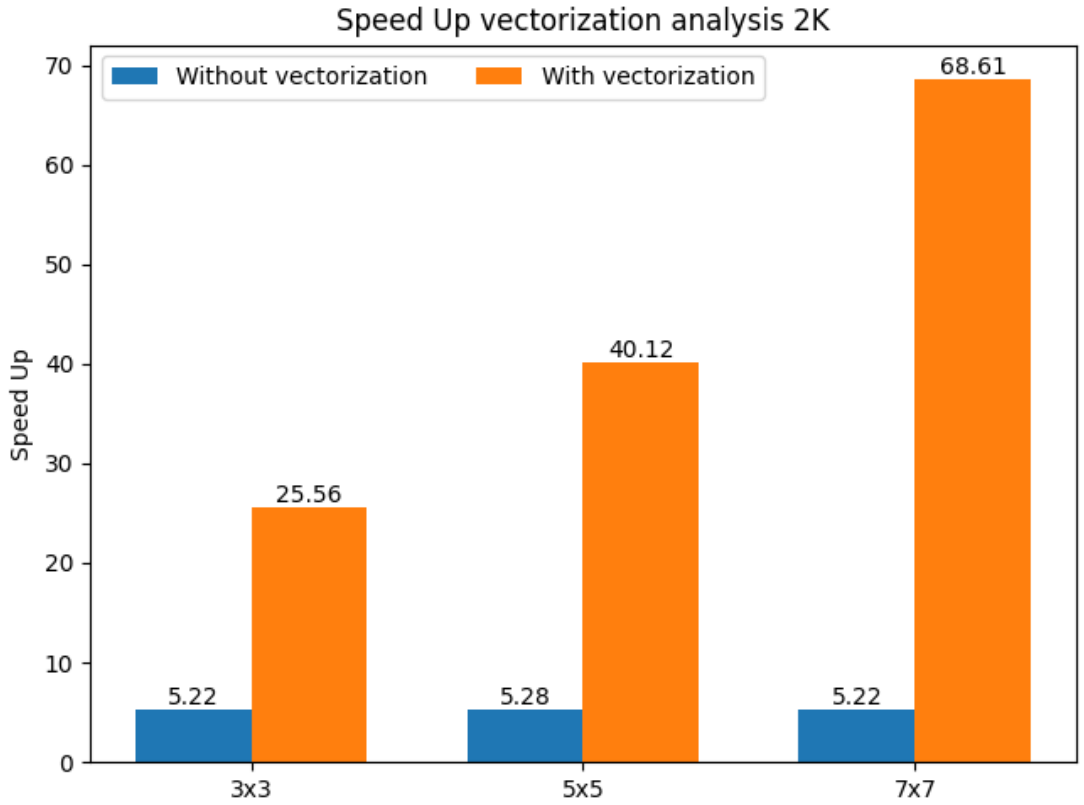




- With this type of implementation, we achieved a significant speedup compared to the previous version.
- As the kernel size increases, a significant improvement in terms of speedup is observed this time.
- The highest recorded speedup value, **70.56**, is achieved with HD images using 16 processes for the computation.

Comparison between the two implementations

SpeedUp comparison



- With a 3x3 kernel, an improvement of nearly 5 times is recorded. With a 5x5 kernel, the improvement is about 8 times, and finally, with a 7x7 kernel, the speedup is approximately 13 to 14 times greater.

In conclusion, it can be said that, despite Python not being the optimal choice for this type of task, excellent results can still be achieved using vectorization techniques.