Non Local Means- CUDA

Gitopoulos Giorgos
Panagiotou Alexandros

Abstract

One of the main challenges in the field of image processing is image denoising, where the goal is to restore the original image by removing noise from a noisy version of the image. Image noise may be caused by different conditions, such as sensors or environment which are not possible to avoid. Therefore, image denoising plays an important role in a wide range of applications (i.e. visual tracking, image classification), where obtaining the original image is crucial for performance and many algorithms have been proposed for that purpose.

Introduction

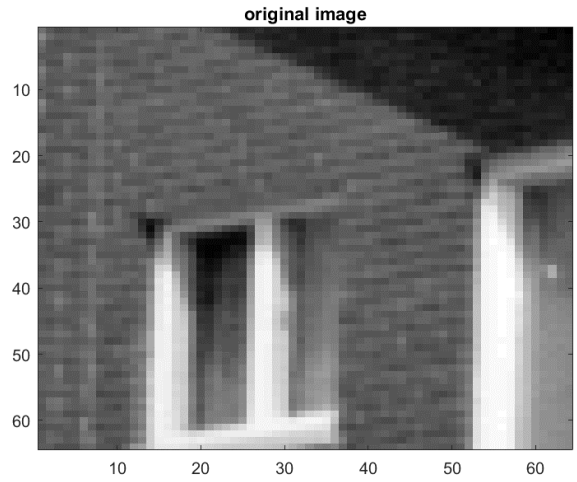
Our project deals with an image denoising algorithm, called Non Local Means [reference to paper]. We used images of three different sizes (64*64, 128*128, 256*256) and after adding on purpose gaussian noise (as an approach of actual noise), we implemented the NL-Means algorithm to remove that noise. The first version of our project is the sequential implementation using C++. Afterwards, been aware of the high complexity of the algorithm that will be analyzed in [reference to paragraph], in the second version we used GPU programming in CUDA, to improve our performance. In the third version, we tried to take advantage of GPU shared memory to accelerate our code even more.

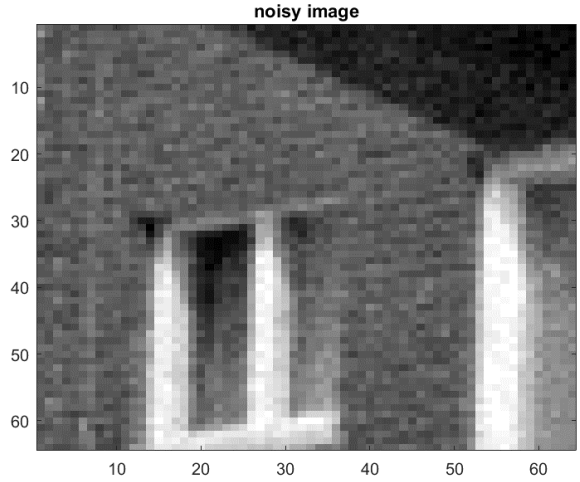
NL-means algorithm implementation

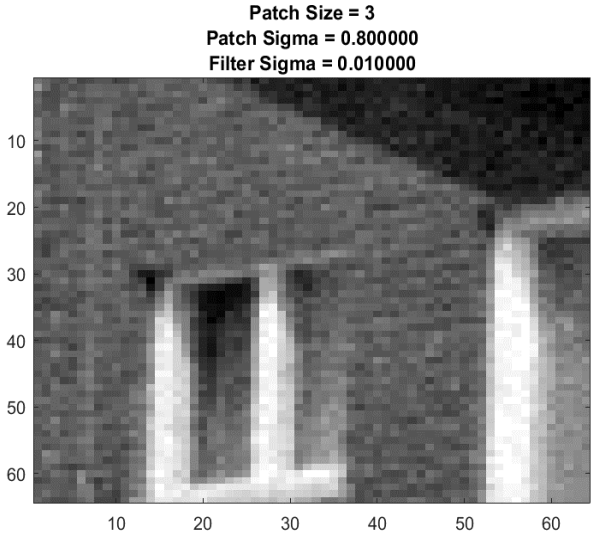
As described in the original paper, NL-means algorithm is a filter that computes, for a every single pixel of the image, a weighted average of all the other pixels. [give formula 1]

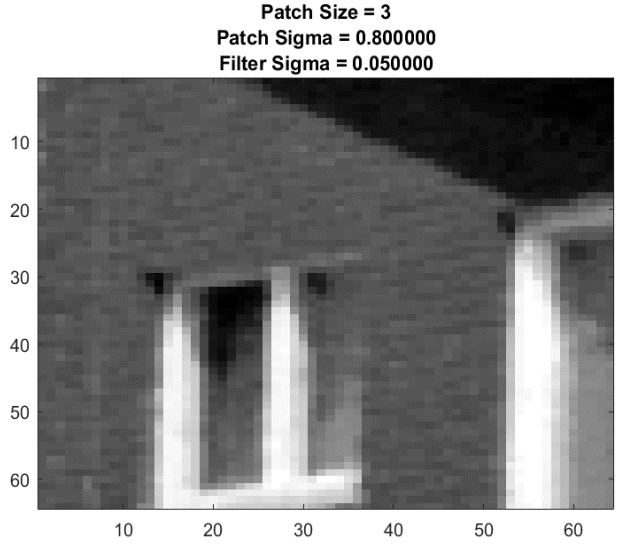
The weight between two pixels is a gaussian kernel of their areas similarity. As a result, a pixel whose neighborhood is very similar to our reference pixel, will contribute more (high weight) to the result of pixel filtering. In that way, the NL-means not only compares the grey level in a single point but the the geometrical configuration in a whole neighborhood. [give formula 2]

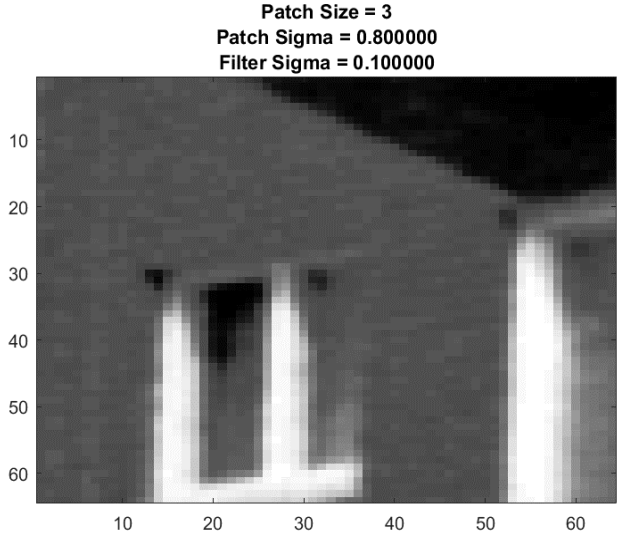
Gaussian kernel gives us the ability to control the strength of the filter with the parameter filter sigma. Increasing filter sigma leads to stronger denoising and makes the image smooth with the risk of loosing details of the original image. On the other hand, a low value of sigma decrease the effect of the filter and keeps us closer to the reality. As we can see in the following figures, it is important to find an optimal value for sigma to approach the original image (about 0.05 for constant other parameters).


original image
10
20
30
40
50
60
10
20
30
40
50
60


noisy image
10
20
30
40
50
60
10
20
30
40
50
60


Patch Size = 3
Patch Sigma = 0.800000
Filter Sigma = 0.010000
10
20
30
40
50
60
10
20
30
40
50
60


Patch Size = 3
Patch Sigma = 0.800000
Filter Sigma = 0.050000
10
20
30
40
50
60
10
20
30
40
50
60


Patch Size = 3
Patch Sigma = 0.800000
Filter Sigma = 0.100000
10
20
30
40
50
60
10
20
30
40
50
60

In the previous paragraph, we stated about the neighborhood of a reference pixel. To make this term more specific, we are talking about a squared patch of pixels, that has as a central pixel our reference pixel. So, this squared patch will be called pixel patch (of the specific pixel). The size of the patch is another parameter we are responsible to choose to make our algorithm efficient. Increasing patch size, means that we are concerned about a bigger area of similarity between two patches, so we are looking for long patterns in the image. Also, in a higher resolution images, it is clear that we have to use bigger patch size to cover a satisfying area of the picture, while in lower resolution images, a comparatively smaller patch size is required.

As a metric of the similarity between two patches we use their Euclidean distance. However, we apply the gaussian kernel in this computation to give more emphasis on the pixels that are more close to the reference pixel. As a result, the central pixels of the two comparing patches have increased weight in the distance computation. This effect can be controlled by patch sigma that is used in the gaussian kernel.

CPU version

In our original implementation, we used the NL-means algorithm, as described in the previous section, to compute the output of the denoising filter sequentially. We suppose that the size of our noisy input image is n*n pixels (squared image). The matrix of the weights of the inside gaussian kernel is precomputed and its size is patchSize*patchSize. For each pixel of the image, we compute the patch-to-patch distance with every other pixel to find the weighted average. This operation for a single pixel has Θ(n*n) complexity, as it passes all the pixel of the image. However, it will take place for every single pixel of the image, so the overall complexity becomes Θ(n*n)*Θ(n*n)=Θ(n^4).

It is obvious that for increasing resolution images, the complexity of the sequential algorithm is prohibitive. Just think that multiplying the one dimension by two, leads to 2^4=16 times increasement in time complexity. To solve this problem and decrease time complexity, we adjusted our code in GPU using CUDA.

GPU version

In this version, we utilized the GPU advantages to accelerate our code. We created a CUDA kernel to filter the image as follows: We asked for n GPU blocks, each of them consists of n GPU threads. Each thread now, is responsible for the filtering of a single pixel (in the exact same way as previously) and all the n^2 operations can be executed in parallel, if we have enough resources. So, by dividing the total work to n^2 workers the complexity drops to Θ(1)*Θ(n^2)=Θ(n^2). In [section benchmarking] we present the speedup of this version and analyze the results.

+[space complexity on global memory]

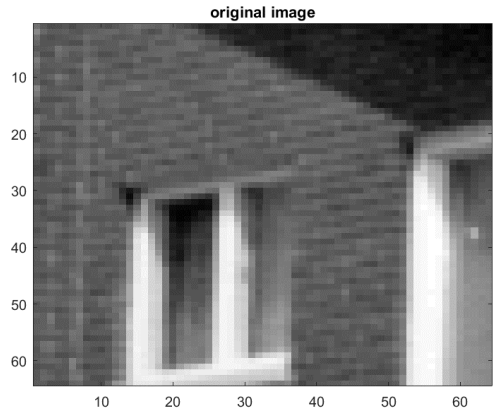
GPU shared memory version

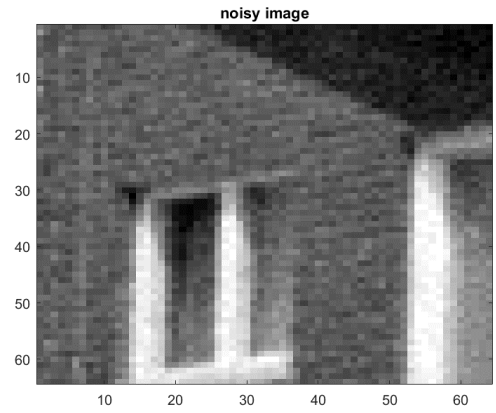
[…….]

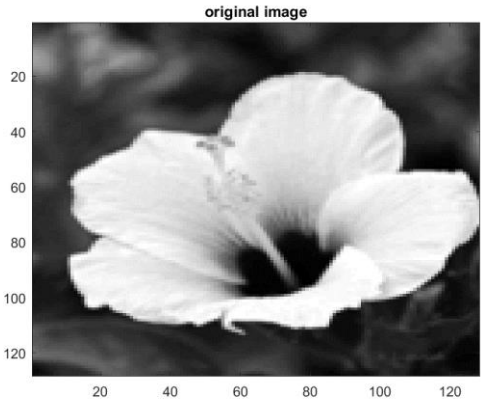
+[time and space complexity on global an shared memory]

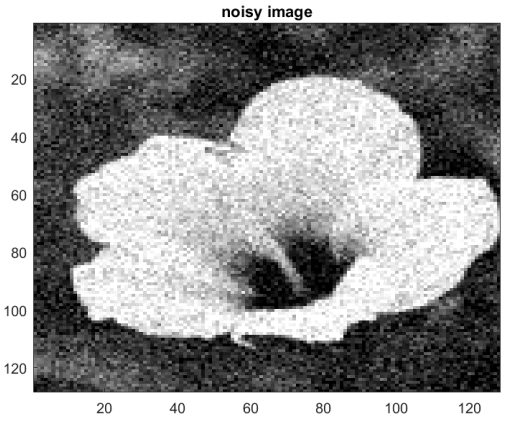
Results

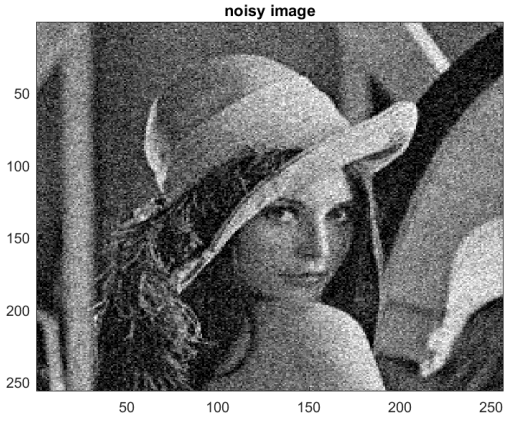
To test our algorithm effectiveness, we chose three squared images with three different sizes. The first will be called “House” (64*64 pixels), the second one “Flower” (128*128 pixels) and the last one “Lena” (256*256 pixels). We technically added to all of them gaussian noise with sigma = 0.1 using Matlab.



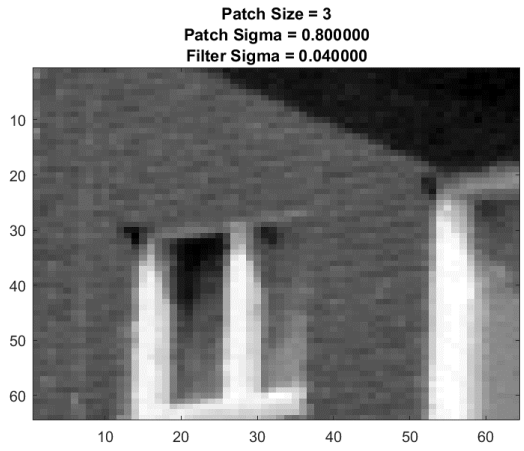


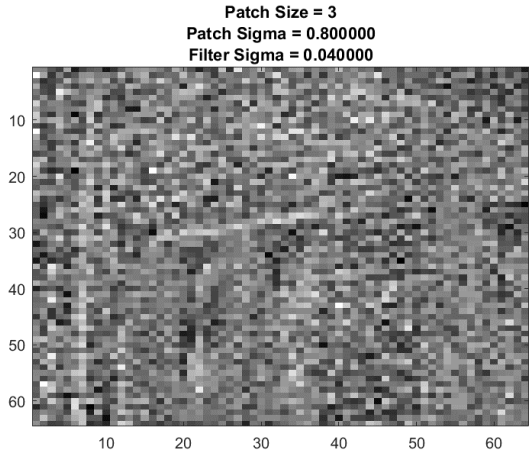


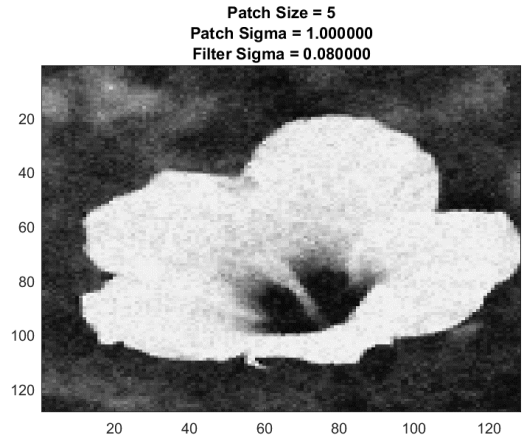


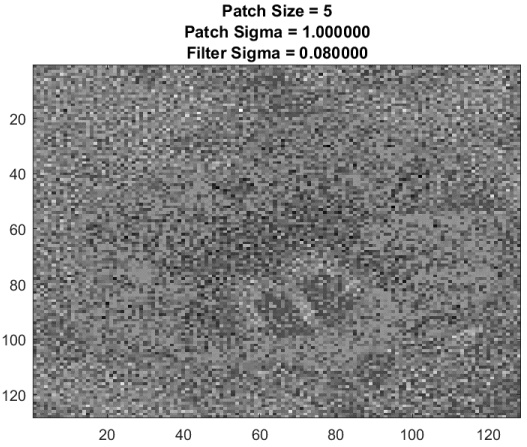


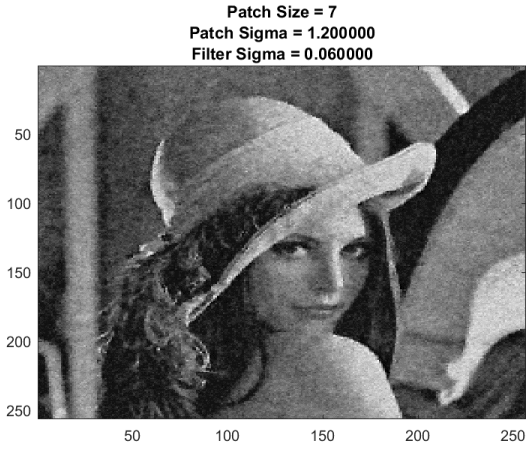
Considering the analysis of the parameters behavior, we adjusted them to some optimal values for each image and we present the results. The following figures show the filtered images and the residual of each image, that is the part of the noisy image that was removed. Optimally, the residual should be clear noise and no details of the original image should be visible.

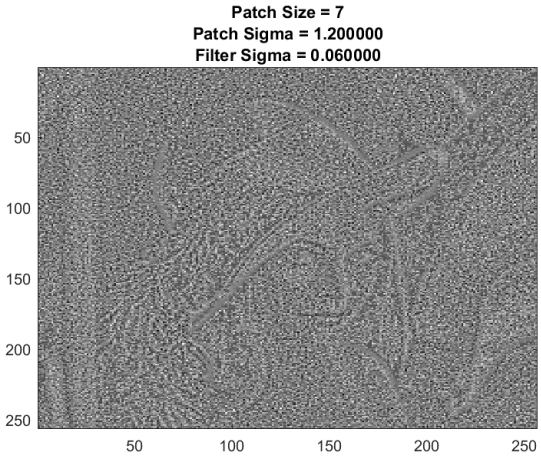












[comments on results and parameters]

Benchmarking

[table: imageSize, patchSize -> executionTime for each version (3 tables)]

[speedup plot: y=speedup(t_cpu/t), x={version1, version2, version3} for every image and patchSize]

[comments on results]

Discussion