Non Local Means - CUDA

Gitopoulos Giorgos 9344, Panagiotou Alexandros 9412

Parallel & Distributed Systems @ ECE AUTh

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, intrinsic or extrinsic, 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. Many algorithms have been proposed for that purpose.

1. Introduction

Our project deals with an image denoising algorithm, called **Non Local Means** [1]. We used images of three different sizes (64×64, 128×128, 256×256) and after adding on purpose **Gaussian noise** (as an approach to 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, being aware of the **high complexity** of the algorithm that will be analyzed in section 3, 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.
The code of our project can be found in the following link: **https://github.com/georgegito/non-local-means-cuda**.

2. NL-means algorithm implementation

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

$$\hat{f}(x) = \sum_{y \in \Omega} w(x,y) f(y), \quad \forall x \in \Omega$$

2.1. Outer Gaussian kernel

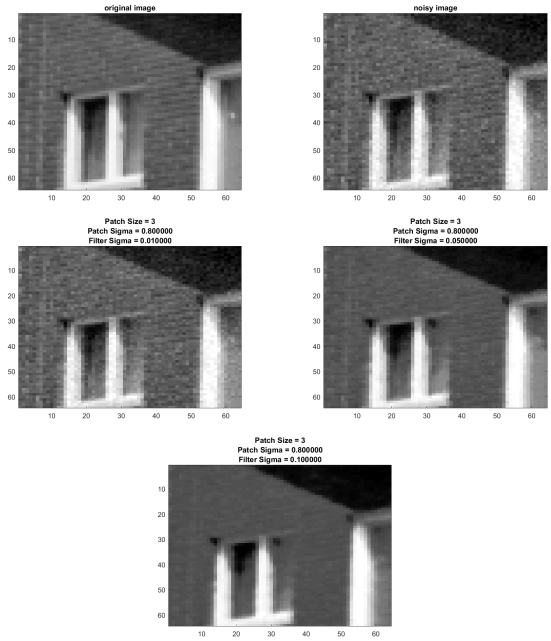
The weight $w(x,y)$ between two pixels x, y is the **Gaussian kernel** of their areas similarity and will be called **outer Gaussian kernel**. As a result, a pixel whose neighborhood is very similar to our reference pixel, will contribute more (high weight) to the result of pixel filtering. In that way, the **NL-means** not only compares the grey level in a single point but the the geometrical configuration in a whole neighborhood:

$$w(i,j) = \frac{1}{Z(i)} \exp\left(-\frac{|| f(N_i) - f(N_j) ||^2_{G(a)}}{\sigma^2}\right),$$

$$Z(i) = \sum_j \exp\left(-\frac{|| f(N_i) - f(N_j) ||^2_{G(a)}}{\sigma^2}\right),$$

where N_k is a neighbourhood around the pixel k, and $G(a)$ is the **inner Gaussian kernel** that is applied in the similarity computation between two neighborhoods (we will discuss about it in subsection 2.2).

The **outer 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 losing details of the original image. On the other hand, a low value of σ decreases the effect of the filter and keeps all the details of the image, including some remaining noise. As we can see in the following figures, it is important to find an optimal value for σ to approach the original image (about 0.05 in this case, for constant other parameters).



2.2. Patches & Inner Gaussian kernel

In the previous paragraph, we mentioned 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 of choosing to make our algorithm efficient. Increasing $patch\,size$, means that we are concerned about a larger area of similarity between two patches, so we are looking for long patterns in the image. Also, in higher resolution images, it is clear that we have to use a larger $patch$ 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 **(inner) Gaussian kernel** in this computation to give more emphasis on the pixels that are closer 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 inner kernel.

3. 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 $n \times n$ pixels (squared image). The weight matrix of the inner Gaussian kernel is precomputed and its size is $patchSize \times pachSize$. 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 $\Theta(n \times n)$ complexity, as it accesses all the pixels of the image. However, it will take place for every single pixel of the image, so the overall complexity becomes:

$$\Theta(n \times n) \times \Theta(n \times n) = \Theta(n^4).$$

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 the time complexity. To solve this problem and decrease the time complexity, we adjusted our code in **GPU** using **CUDA**.

4. 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:

$$\Theta(1) \times \Theta(n^2) = \Theta(n^2)$$

In section 7 we present the speedup of this version and analyze the results. Currently, we utilize **GPU global memory**, where we allocate $(n^2 + n^2 + patchSize^2) \times size\_of\_float$ space.

5. GPU shared memory version

Each block of the **GPU** owns a private part of memory, called **shared memory**. Only threads of the specific block have access in this memory and accessing **shared memory** instead of **global memory** reduces latency rapidly. **Shared memory** behaves as a cache memory: It is much faster than the main memory, but its capacity is limited.

In the last version of our project, we try to take advantage of shared memory to make our code more efficient. As an optimization of the previous version and considering that the threads of each block access global memory to read the same data multiple times, we decided to make a copy of the most used data of every block to shared memory.

As previously, each row of the image is assigned to a block. So, thinking that during the patch-to-patch distance computations many patches of the reference pixels overlay each other, we decided to copy all the patches of the pixels of the specific block to shared memory. In this procedure, we avoid copying the same pixel multiple times, as each thread loads to shared memory only the pixels of its patch that are in the same column with its assigned pixel.

5.1. Bank conflicts

Despite its great access speed, shared memory has an important drawback, called **bank conflicts**. Shared memory is divided into equally sized memory modules (banks) that can be accessed simultaneously in order to increase access speed. However, if multiple threads request addresses map to the same memory bank, the accesses are serialized and the performance gets worse.

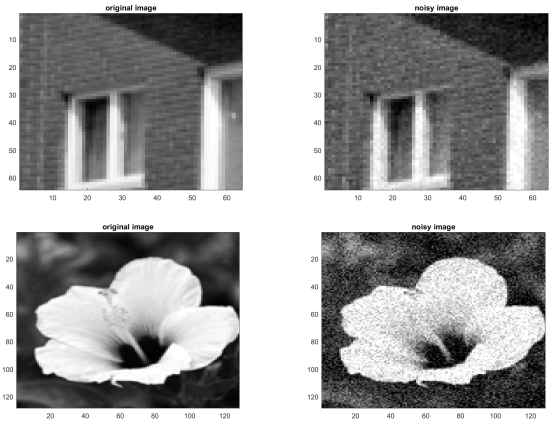
It is obvious that we have to be very careful in our code to avoid bank conflicts. For example, we chose not to load to shared memory the matrix of the inner weights, as all the threads access it continuously and the bank conflicts would affect our performance. In that case, we prefer to load the data from global memory, whose architecture is different and bank conflicts do not take place.

5.2. Complexity

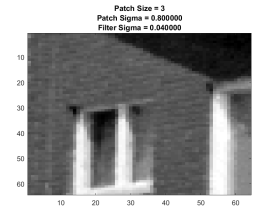
Overall, the **time complexity** is the exact same as in the second version ($\Theta(n^2)$), so does the **space complexity**, but we expect an improvement due to the utilization of shared memory. Also, the **space complexity of shared memory** is $n \times patchSize$. For example, for an input image of 128×128 pixels and $patchSize = 5$, we copy to shared memory of each block $(128 \times 5) \times size\_of\_float = 640 \times size\_of\_float\ bytes$.

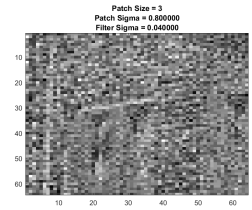
6. Results

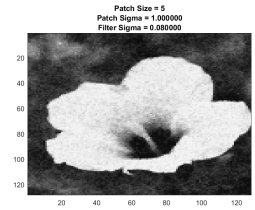
To test our algorithm's effectiveness, we chose three squared images with three different sizes. The first will be called "House" (64×64 pixels), the second one "Flower" (128×128 pixels) and the last one "Lena" (256×256 pixels). We added to all of them Gaussian noise with $\sigma = 0.1$ using Matlab:

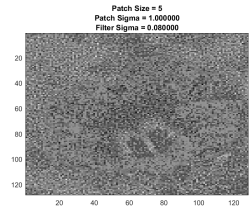


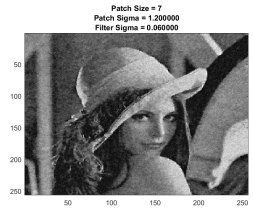
Considering our analysis of the parameters behavior in the previous sections, we adjusted them to some optimal values for each image and we present the results. The following figures show the **filtered images** and the **residual** of each image (the part of the noisy image that was removed). Optimally,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

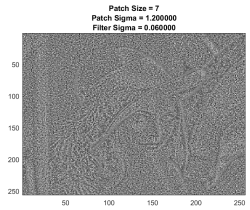












While image size increases, we prefer to increase the size of the patch to keep including a satisfying neighborhood around the reference pixel. $Filter\ \sigma$ and $patch\ \sigma$ where selected by observing the results for different values. The output images are quite close to the original and we can verify from the residual figures that we do not lose many important details.

7. Benchmarking

We tested the performance of our code on the **AUTh High Performance Computing (HPC)** infrastructure and collected results of the execution time. Considering that $filter\,\sigma$ and $patch\,\sigma$ obviously do not affect the speed of our code (only the output image), we will analyze the execution time with respect to $imageSize(n \times n)$ and $patchSize$ for each version of our project. We present our results in the following tables:

Table 1: *CPU version execution time (ms)*

n	patchSize = 3	patchSize = 5	patchSize = 7
64	604	1123	1901
128	9514	18105	32040
256	164090	292121	515052

Table 2: *GPU global memory version execution time (ms)*

n	patchSize = 3	patchSize = 5	patchSize = 7
64	653	671	807
128	853	950	1121
256	1601	2033	2802

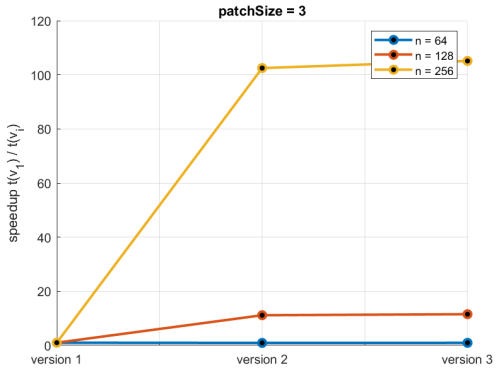
Table 3: *GPU shared memory version execution time (ms)*

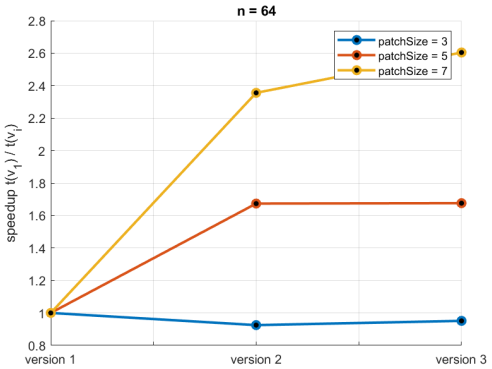
n	patchSize = 3	patchSize = 5	patchSize = 7
64	635	670	730
128	825	950	1072
256	1561	1962	2608

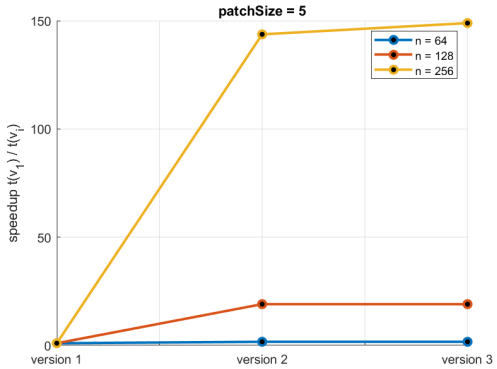
Subsequently, we present some plots to show the **speedup** of every version (version 1 $\rightarrow$ cpu , version 2 $\rightarrow$ gpu global memory, version 3 $\rightarrow$ gpu shared memory). We define the speedup of each version:

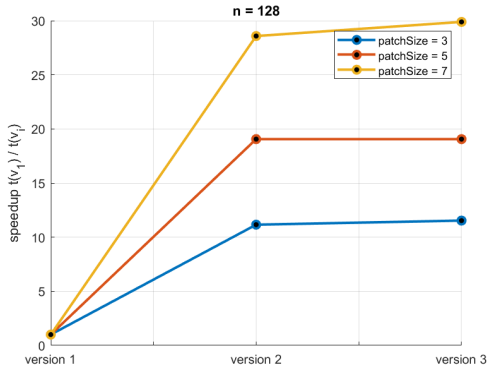
$$speedup(v_i) = \frac{t(v_1)}{t(v_i)}$$

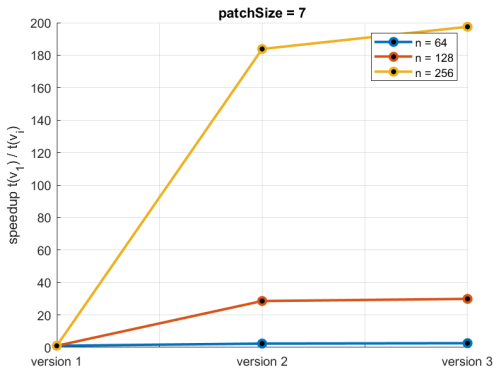
Also, we categorize our plots with both constant $patchSize$ and consant n to analyze the results.

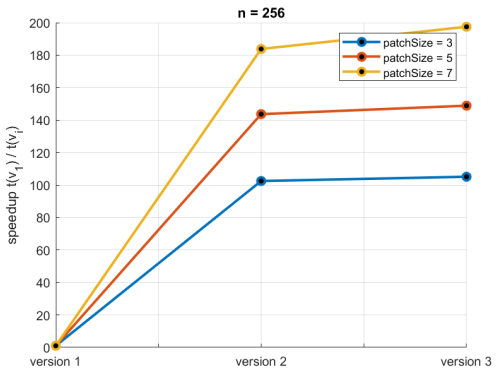












We can clearly see that the larger an image is, the higher acceleration it gets in every version. Moreover, increasing the size of the patch leads to better parallel performance too.
The improvement from **CPU** to **GPU** version is great. Especially for large images and high values of $patchSize$, the speedup ranges between 100 and 200. On the other hand, although shared memory utilization improves performance even more, the difference between second and third version is not that important.

8. Discussion

Overall, our **NL-means** algorithm implementation works as expected, regarding to the output images. For selected values of parameters, the results are very close to the original images. In addition, our goal of accelerating the performance using **GPU** was achieved and the optimization using shared memory improved performance further.

9. References

[1] Antoni Buades, Bartomeu Coll, and JM Morel. "A non-local algorithm for image denoising"