Aristotle University of Thessaloniki
School of Electrical & Computer Engineering

Parallel and Distributed Computer Systems
3rd Assignment

Pakas Marios, 9498, mariospakas@ece.auth.gr

Meta Louis-Kosmas, 9390, louismeta@ece.auth.gr

Github URL: https://github.com/Mavioux/Parallel-And-Distributed-Systems-Exercise-3

0. Abstract

This report is about the implementation of a nonlocal means algorithm in both serial and parallel ways. For a given image the algorithm calculates the denoised image by removing the noise. This is particularly useful for low-dose X-ray computed tomography[1] where we need to denoise natural images corrupted by additive Gaussian noise. Essentially, by using nonlocal means filters for image-guided surgeries and image-guided radiotherapy, which need to have high-quality images, the noise is reduced by replacing the intensity of each pixel with a weighted average of its neighbors according to similarity. Furthermore, we use applications from this algorithm in movie denoising methods in order to provide better video quality. However, although its implementation is not significantly difficult, it needs run time optimization to be efficiently applicable to each of its operations. According to this, parallel implementation is undoubtedly necessary in order to get the desired results in a reasonable time frame.

1. Nonlocal Image Algorithm Sequential implementation

The serial implementation of the algorithm looped through each image pixel and calculated its final value. To achieve that, it created an array(pixel_patch) with a size of patchsize^2, which would be then compared with the comparison_patch array of each pixel of the image. For that, a second loop was used to go through each pixel of the image, and the comparison_patch was stored and then subtracted from the pixel_patch. This, divided by sigma^2, given a negative sign and then passed as a power of e would be multiplied with the value of the comparison pixel and added to the value of the original pixel. All those weights would be added to a variable zeta, with which we will divide the value of the original pixel, so as to make sure that in the end, the value is between 0 and 1. When the initial loop is finished (O(n^4)) the image produced would have been smoothed out and the Gaussian noise should not be visible anymore.

2. Nonlocal Image Algorithm Parallel implementation

This implementation uses parallel methods in order to speed up the calculation of the value of each pixel by using Cuda programming. The general idea here was to call a width*height kernel and assign each pixel to a different GPU thread. This enabled us to calculate the final value the same way as serially, but simultaneously

for each pixel. This way the complexity of the code was reduced down to $O(n^2)$ with dramatic differences in the time it took to finish the job, especially as the size of the image or the patch size increases.

3. Helpful functions

In order to manipulate images in C, a library found on Github was used that enabled image loading and storing in 1D arrays[2]. For the addition of white gaussian noise to the original image, the box-muller algorithm was used. There were two awgn_generator functions created, one was our own implementation, and the second one was a function found on a site[3]. Since the second function seemed to provide better noisy images, we proceeded with that implementation.

4. Experimental Results

All of the experimental results mentioned below occurred after testing our implementations in the AUTh High Performance Computing (HPC) infrastructure. Both parallel and sequential implementations were executed for images with 64px, 128px and 256px and patch sizes values of 3,5 and 7.

Sequential implementation using only CPU

	3	5	7
64	4,25	9,41	15,86
128	72,42	171,56	309,92
256	1190,61	2940,71	4616.09

Parallel implementation using CUDA

	3	5	7
64	0,14	0,18	0,23
128	0,37	1,14	2,75
256	5,52	39,23	122,25

First of all, regarding the serial implementation, it is clear that an increase in the number of patch size value causes a small increase in the execution time, though the biggest factor is definitely the image size, due to complexity $O(n^4)$. That is why the execution time in parallel is way smaller since the complexity is $O(n^2)$.

In the following diagrams, we can see for the image we used for testing sequential and parallel implementations the execution time compared for each case. Specifically, in every figure from 1 to 9, we have bar charts to show the differences in run time durations. The magenta bar charts represent the CPU and the GPU (multiplied by 10% to be visible in the diagram) time durations relative to the left y-axis. The green bar represents the GPU execution time duration without the scaling relative to the right y-axis. We can see the huge improvements parallelization brings to the table. Moreover, the speedup diagram of each image and patch size that we can see in the 10th figure proves that we perform from 30 to 215 times faster execution time in parallel than in serial implementation. Finally, the Cuda implementation is incomparably better and more efficient but obviously presupposes that the resources are available to implement it.

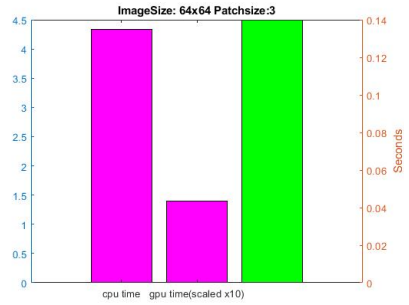

Fig.1

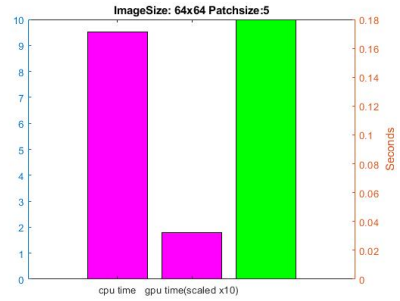

Fig.2

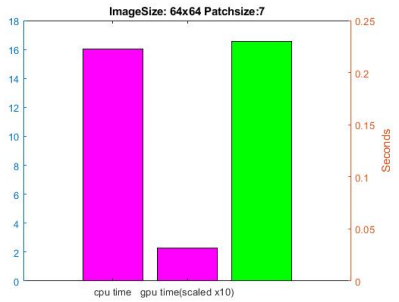

Fig.3

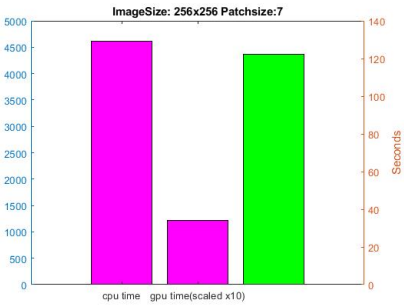

Fig.4

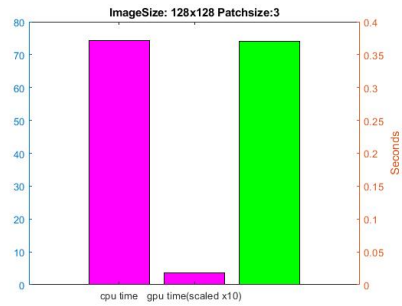

Fig.5

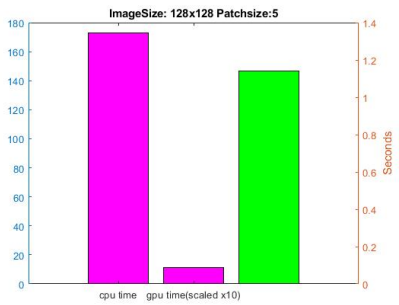

Fig.6

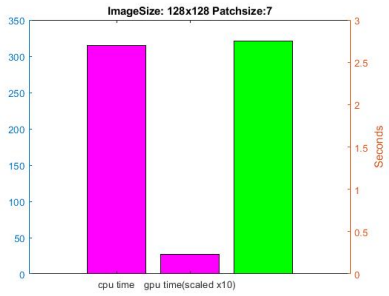

Fig.7

Fig.8

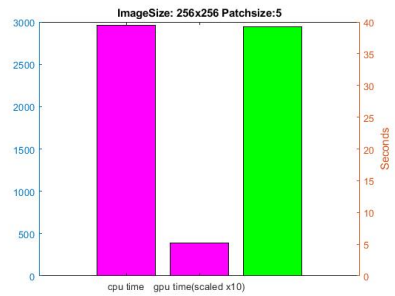

Fig.9

Note: All run time durations were measured in seconds.

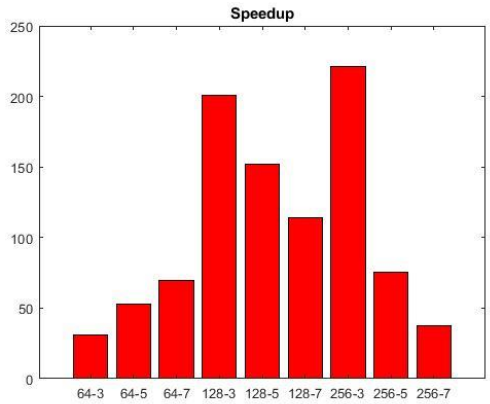

Fig.10

Fig. 1 Fig. 2 Fig. 3 Fig. 4 Fig. 5

Fig. 1 Fig. 2 Fig. 3 Fig. 4 Fig. 5

Fig. 1: Original Image with Colours
Fig. 2: Black and White Image
Fig. 3: Image with Gaussian Noise
Fig. 4: Denoised Image using the Algorithm
Fig. 5: Noise extracted

After experimenting with different values, we decided on a compromise in the following values:
filter_sigma: 0.2
patch_sigma: 1.67
Those numbers work quite well with many images we tried them on, though each image may require some experimentation to find the perfect pair values.

In the first image, we can see that the algorithm does an exceptional job in clearing up the Gaussian noise while preserving most of the original details. That is not the case with the second image, which admittedly has much more detail than the first one. In both cases though, it is clear that the noise has been removed.

Note: Further explanation and analysis for each implementation mentioned above are available as comments in the source code and the Github repository.

Sources:

[1] Low-dose X-ray computed tomography: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5381744/

[2] Image Manipulation Library: https://github.com/nothings/stb/

[3] AWGN: https://www.embeddedrelated.com/showcode/311.php

Extra PDF on NonLocal Image Algorithm: https://hal.archives-ouvertes.fr/hal-00271147/PDF/ijcvrevised.pdf