Manuscript

Click here to view linked References

Journal of Real-Time Image Processing manuscript No.
(will be inserted by the editor)

Donatella Granata · Umberto Amato · Bruno Alfano

MRI denoising by NonLocal Means on multi-GPU

Received: date / Revised: date

Abstract A critical issue in image restoration is noise removal, whose state-of-art algorithm, NonLocal Means, is highly demanding in terms of computational time. Aim of the present paper is to boost its performance by an efficient algorithm tailored to GPU hardware architectures. This algorithm adapts itself to several variants of the methodologies in terms of different strategies for estimating the involved filtering parameter, type of noise affecting data, multicomponent signals, spatial dimension of the images. Numerical experiments on brain Magnetic Resonance images are provided.

Keywords NonLocal Means · Image denoising · Magnetic Resonance Imaging · Graphical Processing Unit.

1 Introduction

Image denoising is a mainstay of the of medical image preprocessing. Keeping the integrity of image while removing its noise remains a critical issue, especially for ultrasound and Magnetic Resonance (MR) images, due to the presence of structures with a signal barely recognizable above the noise level.

Work done under the project MEdical Research in ITaly RBNE08E8CZ and POR Campania FESR 2007/2013 project Bersagli funded by the Italian Ministry of University and Research.

Donatella Granata
Istituto per le Applicazioni del Calcolo 'Mauro Picone' CNR, Napoli, Italy
E-mail: d.granata@na.iac.cnr.it

Umberto Amato
Istituto per le Applicazioni del Calcolo 'Mauro Picone' CNR, Napoli, Italy

Bruno Alfano
Istituto di Biostrutture e Bioimmagini CNR, Napoli, Italy

NonLocal Means (NLM) filter (Buades et al., 2005) is nowadays considered the state-of-art algorithm for denoising MR images. It belongs to the class of denoising methodologies for which denoised intensity at a voxel is estimated as a weighted mean of intensities in suitably chosen voxels. The rationale behind computation of NLM weights relies on two different arguments: firstly, weights among voxels depend on the similarity of intensity in voxels rather than on their spatial distance; as such, in principle the averaging process can involve the whole image. Secondly, similarity of intensity is defined in a region of voxels, instead of at single voxels, better capturing in this way the local textures of the image. Despite of its success, by its very nature NLM algorithm is highly demanding from the computational point of view, becoming unsuitable for real time applications and impracticable even for offline processing and for tuning its parameters without introducing severe limitations on the regions of voxels. To overcome this drawback, several adaptations have been proposed in the literature, such as the optimized blockwise version by Coupé et al. (2008b); Tristán-Vega et al. (2012); Coupé et al. (2008a), the approach of Coupé et al. (2008a) based on the mixing of wavelet sub-bands.

Most MR Images are affected by Rician noise, coming from the the FFT of the k-space where the noise is Gaussian. A notable exception is given by phased array coils (Constantinides et al., 2008; Koay and Basser, 2006) and parallel acquisition techniques such as sensitivity encoding for fast MRI (SENSE; Pruessmann et al. (1999)) and the generalized autocalibrating partially parallel acquisitions (GRAPPA; Griswold et al. (2002)). In this case the k-space is affected by non-central Chi distribution noise, that produces MR images with spatially non uniform Rician noise. To address spatially nonuniform noise variance local noise estimation of (Manjón et al., 2010) has been introduced.

During the last decade, Graphical Processing Units (GPU) have received much attention as a hardware platform that is complementary to Central Processing Units (CPU) in modern computers. This is due to the low cost of the GPU boards–even included in most consumer personal computers–and to their growing technological advancements that boosted its computational performance over two orders of magnitude faster than CPUs for algorithms suitable for massive parallelism. On the other hand the intrinsic massively parallel nature of many denoising algorithms straightforwardly matches the hardware characteristics of GPUs, making them a perfect tool to speed up algorithms.

As a consequence algorithms for NLM were specifically developed for GPU architectures. Kharlamov and Podlozhnyuk (2007) introduced a first NVIDIA CUDA SDK implementation. GPU versions of the algorithm were also presented in Palhano et al. (2011); Goossens et al. (2010); Huang et al. (2009). Palma et al. (2013) and Cuomo et al. (2014) proposed a full 3D NLM implementation on a multi-GPU artchitecture.

Likewise, a GPU-based fast block-wise NLM algorithm for 3D ultrasound image was presented in Li et al. (2013). Actually GPU architecture is prone to boost computational performance in many other image processing methodologies, e.g., image enhancement (de Araujo, 2014).

As thoroughly reviewed by Gudbjartsson and Patz (1995), noise affecting most MR images actually obeys a Rician density function.

Gaussian assumption currently adopted in most denoising methodologies for MR biases the weighted average involved in NLM especially at a low Signal to Noise Ratio (SNR). Therefore, Rician adapted versions of NLM have been recently proposed to avoid such bias (Manjón et al., 2008; Wiest-Daesslé et al., 2008; Manjón et al., 2010). Unfortunately, the local Rician noise is underestimated on regions where the signal is low; to prevent such a problem a correction factor based on the local SNR can be applied as described by Gudbjartsson and Patz (1995) and Koay and Basser (2006).

Denoising filters usually work on each component of an MR study separately, without taking into account the multicomponent intrinsic nature of the studies, composed of different types of images of the same patient (e.g., T_1, T_2, PD, FLAIR). The first truly multicomponent methodology is due to Gerig et al. (1992), that supports 3D and multiecho MRI, incorporating higher spatial and spectral dimensions. Later, Manjón et al. (2009) introduced a multicomponent version of the NLM procedure, evaluated on synthetic and real clinical data. They proved that multicomponent methodologies improve one component ones due to the increased redundancy of data.

Aim of the present paper is to develop an efficient algorithm for removing noise from MR images by NLM for multi-GPU architectures. The algorithm will deal with one component or multicomponent signals, spatially uniform or locally adaptive variance of the noise, Guassian or Rician models for the noise.

The paper is organized as follows. Section 2 introduces the NLM filter and its features. Section 3 focuses on the specific NLM methodology adopted for the present paper, introducing its GPU implementation. Section 4 includes some experiments on MR images to show algorithm efficiency under several settings. Comparisons of computational time with state-of-art GPU NLM algorithms are also shown.

2 Materials and methods

2.1 NLM Filter

Let Ω be the full 3D volume of the image and x_i the corresponding voxels, $x_i \in \Omega$. We denote by $\mathbf{u}_i \equiv (u_i^1, \ldots, u_i^C)$ the multicomponent intensity of the MR study at voxel x_i, assuming that C components are provided. Analogously we denote by $\tilde{\mathbf{u}}_i$ the intensity at pixel x_i after noise removal. In the full 3D formulation of the NLM filter (Buades et al., 2005) the restored intensity at a generic voxel x_i is a weighted average (also called kernel convolution) of the intensities at the voxels belonging to a 3D search volume $V_i \subseteq \Omega$, centred at the voxel x_i:

$$\tilde{u}_i^c = \sum_{x_j \in V_i} w^c(x_i, x_j) u_j^c, \ c = 1, \ldots, C, \tag{1}$$

where $w^c(x_i, x_j)$ is the weight assigned to u_j^c in removing noise from voxel x_i. To be more specific, the weight is an estimate of the similarity between the

intensities of two 3D neighborhoods $D_i \subseteq \Omega$ and $D_j \subseteq \Omega$ centred at voxels x_i and x_j, respectively, such that

$$w^c(x_i, x_j) \in [0,1], \sum_{x_j \in V_i} w^c(x_i, x_j) = 1, \; c = 1, \ldots, C.$$

The definition of the classical NLM filter does not make any assumption about the search volume, it only requires that each voxel can be linked to the others. However for computational reasons it is usually assumed that both neighbors V_i and D_i are taken as cubes of size $(2v_x+1)(2v_y+1)(2v_z+1)$ and $(2d_x+1)(2d_y+1)(2d_z+1)$, with (v_x, v_y, v_z) and (d_x, d_y, d_z) being the window radii of V_i and D_i respectively, along each spatial direction x, y and z.

We mention that in the 2D formulation of NLM weight average occurs over voxels belonging to a same slice, so that noise removal is performed slice by slice and processing of each slice does not involve any information on other slices. Furthermore, in the single component formulation the convolution is computed component by component (e.g., T_1, T_2, PD, FLAIR, etc.) and the result is not affected by the remaining components.

The similarity between two voxels x_i and x_j is expressed by an index based on the multicomponent intensity in the regions D_i and D_j. According to Efros and Leung (1999), the L^2 distance is a reliable measure for estimating the similarity of image windows in a texture patch. Therefore it can be estimated as the (possibly Gaussian weighted) Euclidean distance $\|u(D_i) - u(D_j)\|_2^2$ between the patches $u(D_i)$ and $u(D_j)$. The weights associated to the quadratic distance are introduced in (Buades et al., 2005) and written in the multicomponent general case as (Manjón et al., 2009)

$$w(x_i, x_j) = \frac{1}{Z_i} \exp\left(-\frac{1}{C} \sum_{c=1}^{C} \frac{\|u^c(D_i) - u^c(D_j)\|_2^2}{(h^c)^2} \right),$$

where Z_i is a normalization constant to ensure that $\sum_{x_j \in V_i} w(x_i, x_j) = 1$ and $h^1, \ldots, h^C$ are multicomponent filtering parameters that control the decay of the exponential function.

Indeed, for each voxel in the volume, the distances between the intensity neighborhoods $u^c(D_i)$ and $u^c(D_j)$ for all voxels x_j contained in V_i and for each component c need to be computed. Denoting by N the size of the 3D image, the complexity of the filter is of the order of $\mathcal{O}(NC(2v_x+1)(2v_y+1)(2v_z+1)(2d_x+1)(2d_y+1)(2d_z+1))$. For isotropic images we can assume $v_x = v_y = v_z = v$ and $d_x = d_y = d_z = d$.

2.2 Choice of the filtering parameters h^c

Buades et al. (2005) proved that the NLM algorithm is consistent for every non negative h^c. Nevertheless, they experimentally showed that the value of the filtering parameter h^c must be taken of the order of the noise standard deviation σ^c. Essentially the filtering parameter controls how much two

intensity patches that are different can be considered as similar; to choose h^c of the order of the noise standard deviation is equivalent to assume that intensity patches can be considered similar if they differ by a quantity that is of the order of the noise standard deviation.

Estimate of noise variance is often made by visual inspection of denoised images after trial experiments with different values of variance. In a less subjective and data driven approach, $(\sigma^c)^2$ is estimated from the signal external to the skull (air) where only noise is present.

Assumption of a uniform variance over the image leads to suboptimal results in images where noise is nonstationary. Therefore Manjón et al. (2010) proposed the following local adaptive estimate of noise variance at a voxel m:

$$(\sigma_m^c)^2 = \frac{1}{2}\min d(u^c(S_i), u^c(S_j)),\ \ S_i, S_j \subseteq \Omega,\ \ i \neq j,$$

S_k being a volume centred at the voxel x_k. Such estimation can be obtained by observing that the expectation of the squared Euclidean distance of two noisy 3D patches $u^c(S_i)$ and $u^c(S_j)$ is (Buades et al., 2008)

$$\begin{aligned} E\left(d\left(u^c(S_i), u^c(S_j)\right)\right) &= E\left\|u^c(S_i) - u^c(S_j)\right\|_2^2 \\ &= \left\|u_0^c(S_i) - u_0^c(S_j)\right\|_2^2 + 2\left(\sigma_m^c\right)^2, \end{aligned} \quad c = 1, \ldots, C,$$

where $\mathbf{u}_0 \equiv (u_0^1, \ldots, u_0^C)$ is the noise-free intensity. Whenever two patches S_i and S_j have the same noise-free intensity $\mathbf{u_0}$ then $d\left(u^c(D_i), u^c(D_j)\right) = 2(\sigma_m^c)^2$. This condition does not occur in practice, so that, basing on experimental arguments, Manjón et al. (2010) proposed the following estimate:

$$(\sigma_m^c)^2 = \min_{S_j}(d(R^c(S_m), R^c(S_j))),\ \ j \neq m, R^c(S_j) := u_j^c - \psi(u_j^c),\ \ x_j \in V_m,$$

where $\psi(u_j^c)$ is a low-pass filtered intensity, estimated as its average in S_j. The removal of $\psi(u_j^c)$ allows one to compensate the intensity inhomogeneities of MR images and therefore to find more similar patches; incidentally this will also reduce the overestimation of noise variance occurring when almost similar patches are compared.

In practical implementation S_i is chosen as a cube of size $(2s_x+1)(2s_y+1)(2s_z+1)$, with s_x, s_y, s_z being variance radii.

2.3 Rician Noise

As discussed in the Introduction, noise in most MR images follows a Rician density function, therefore a Gaussian based methodology will bias the weighted average in (1) due to asymmetry of the Rician distribution. Manjón et al. (2008) and Wiest-Daesslé et al. (2008) proposed adapted versions of the NLM filter able to remove the bias due to the Gaussian approximation. In the present paper we use the correction scheme by Wiest-Daesslé et al. (2008). It is based on the second-order moment of a Rician distribution

$$E\left[X^2\right] = \mu^2 + 2\sigma^2, \tag{2}$$

where σ^2 is the variance of the Gaussian noise in the k-space before the Fourier transform. In the blockwise version the Rician corrected intensity, $\tilde{u}^c$, $c = 1, \ldots, C$, is given by

$$\tilde{u}_i^c = \sqrt{\max\left(\sum_{x_j \in V_i} w^c(x_i, x_j)\left(\tilde{u}_j^c\right)^2, 0\right)}, \ c = 1, \ldots, C \tag{3}$$

(with abuse of notation we indicate by $\tilde{u}^c$ intensity estimated both by the Gaussian noise assumption and the next Rician correction).

3 A multi option NLM filter on multi-GPU

We have developed multi-GPU NLM algorithms using CUDA toolkit for NVIDIA GPUs. CUDA is a general purpose C-like API which gives more control on how a task is computed on the GPU hardware.

In CUDA, a system consists of a host (the CPU), and one or more devices, which are massively parallel processors. The host can move application data between host and device memory, and invoke a variable number of operations (called kernels) to execute on the device. The kernels exhibit the important property of data parallelism, allowing arithmetic operations to be simultaneously performed on different parts of the data. Kernels are concurrently executed by a very large number of threads. Threads executing the same kernel are organized in levels, i.e. a grid of thread blocks. Each thread block contains up to 1024 threads depending on the device properties. Built-in variables such as (*threadIdx*, *threadIdy*, *threadIdz*) and (*blockIdx*, *blockIdy*, *blockIdz*) provide the thread grid and block index configurations to each thread, and are used to divide the work among the threads. Threads in the same thread block are executed by the same streaming multiprocessor (SM) and are able to synchronize with each other and to use the same available shared memory. Continuous sections of 32 threads in a thread block are defined to be part of a warp, in which all threads execute the same instruction in parallel. Unfortunately whenever threads in a warp take different execution paths, sometimes the suppression of certain threads is required to obtain the complete execution producing lack of performance.
CUDA supports different memory types. Global memory is the largest memory, but with high latency; it is typically used to store input and output data structures. It generally limits the performance of CUDA kernels if no other memory types are utilized. Constant memory and shared memory are both on-chip memories. Constant memory is a small read-only memory, supporting low latency, high bandwidth access when all threads simultaneously access the same location. Shared memory can be allocated to thread blocks and accessed at very high speed in a highly parallel manner. As all threads in a thread block can read and write their own shared memory, it is a very efficient way for threads to share their input data and intermediate results.

In order to take the major advantage from the GPU architecture available, we created a multi-GPU adapted version. As there exists a limitation on

the amount of data that each GPU can process, portions of the original data are processed in any kernel call according to the available memory. Furthermore, to ensure that every pixel can be evaluated in the same way, we created a larger initial copy of the image data where extra borders are present; we fill this extra space with a mirroring of the edge image. To allow the execution of all the operations all over the available GPUs, we need to split the volume in portions proportional to the number of GPUs (G) available on the machine. To guarantee the accuracy of algorithms we adopted larger subdata, enough to contain the data partition and some extra borders that are filled by data coming from the adjacent partitions. For a given full 3D volume Ω of $N = (N_x, N_y, N_z)$ data, whenever the amount of data exceeds the size $m_g, g = 1, \ldots, G$, of each processor global memory, additional workload splits are necessary. All these splits must consider radii size s, d and v and the necessary extra memory to allocate the intermediate kernel structures, indispensable to collect the kernel results and to copy them from the GPU to the CPU unit. Denoting by M the total amount of data for all the G GPUs, the total number of splits is $(N + \epsilon)/M$, where ϵ is the extra data whose rows are filled by extended borders of each split image. Furthermore, the data of each split are divided in G substructures, so $\Omega_{i,j}^{g,c}$ are the subdata belonging to the slice i at split j of the image data component c processed by the GPU g. In addition, a phase of merging is implemented, to gather the restored splitted data $\tilde{\Omega}_{i,j}^{c}$, from the image portions $\Omega_{i,j}^{g,c}$. In particular, this phase handles the presence of the extra borders, needed to ensure the denoising correctness. For an easy notation, we suppose that data and split indices i, j are already scheduled to be processed by a specific GPU $g = 1, \ldots, G$, then all over the paper we use the simplified notation $\Omega_{i,j}^{c}$.

A simple way to implement the denoising filter in CUDA is to load a block of the image data into the shared memory array, dynamically allocated. Each block image is processed by a corresponding thread block that executes the filter on the image portion and writes the restored values to the output structure from the device memory to the CPU structure.

For any filter kernel, the pixels at the edge of the shared memory block depend on pixels that are not loaded in the block shared memory. Therefore to correctly implement the filter, it is necessary to load additional aprons of block pixels into the shared memory. The size of the apron depends on the part of the code: in the case of (eventual) adaptive estimate of the variance (Algorithm 2) aprons are given by the variance radius, $\mathbf{a}_v = (s_x, s_y, s_z)$; for the actual computation of the filter (Algorithm 3) the size of aprons is the sum of the window and of the similarity radii, $\mathbf{a}_d = (v_x + d_x, v_y + d_x, v_z + d_z)$. Therefore threads with *threadIdx.y*$< a_y$ will be loaded at the bottom and top boundary of each block; threads with *threadIdx.y*$< a_x$ will be loaded at the right and left boundaries. Furthermore all these threads will be loaded at the corner voxels (see Fig. 1). For practical reasons, it is suggested to use radii relatively small otherwise there will be many idle threads during the filter computation. Whenever the needed occupancy exceeds memory limits the value a_{d_z} is reduced accordingly.

We implement two different variants of the denoising filter, a single component and a multicomponent one. In the latter case, three different images

of the same patient are provided to the algorithm and a different configuration of the kernel is used. The aim is to create smaller blocks and to limit the amount of shared memory in order to contain all the image data coming from all the three components and to overcome the memory limitation. We use a $(16, 8, 1)$ block size configuration for the multicomponent method and $(16, 16, 1)$ otherwise.

The code is actually organized in three main functions illustrated in Algorithm 1:

1. Initialization and splitting phase;
2. Eventual computation of the adaptive variance (Algorithm 2);
3. Computation of the NLM filtered signal according to the options set with the parameters provided by the user (Algorithm 3).

Algorithms 1–3 present the multicomponent version, aware of the practical equivalence of the two versions that differ for the amount of data used for the weights computation (see Section 2.1) and for the presence of three images for each patient. Algorithm 1 describes the function call with needed Input/Output (I/O) parameters, together with the call to the (eventual) adaptive estimate of noise variance and to the actual filter computation. The denoising filter is computed slice by slice, for all the voxels, and corresponding window and volume. The I/O parameters are driven by some flags, whose settings denote the type of noise affecting images, the number of components of the images, the dimensions of the volume and defining the size of the involved windows:

ADAPTIVE_SIGMA 0: constant noise variance; 1: local adaptive estimate of variance (Section 2.2);
RICIAN 0: Gaussian approximation for noise; 1: Rician correction (Eq. 3);
VOLUME 0: 2D filtering; 1: 3D filtering.
SIGMA_BLOCK_RADIUS, NLM_BLOCK_RADIUS: Variance and similarity radii ((s_x, s_y, s_z), (d_x, d_y, d_z), respectively).
NLM_WINDOW_RADIUS: Window search radius ((v_x, v_y, v_z)).

Furthermore some support structures are defined as device buffers and host buffers for each component c and split j of subdata i:

$\Omega^c_{i,j}$: device buffers to load the data;
$(\sigma^c_{i,j})^2$: device buffers to load the estimate of noise variance;
$\tilde{\Omega}^c_{i,j}$: host buffers containing the restored subdata.

Use of memory has to be carefully managed because of contrasting effects. On one side the use of shared rather than global memory for the NLM algorithms, speeds up the access to data. On the other side access patterns must be carefully aligned and the memory usage should be evaluated in representative test cases.

The shared memory used for the single component NLM algorithm is approximately $M \approx (1 + 2a_d)WIDTH$, where a_d is the apron size and $WIDTH = (16 + 2a_d)^2$ takes account of the slices needed to be simultaneously stored. For a block size $(16, 16, 1)$ similarity and search windows are $d = 1$, $v = 5$, so that $M \approx 20KB$. In the multicomponent algorithm the

Algorithm 1 NLM_GPU($\Omega^1,\ldots,\Omega^C$, $(\sigma^1)^2,\ldots,(\sigma^C)^2$, N_x, N_y, N_z, $\tilde{\Omega}^1,\ldots,\tilde{\Omega}^C$, *SIGMA_BLOCK_RADIUS, NLM_BLOCK_RADIUS, NLM_WINDOW_RADIUS, ADAPTIVE_SIGMA, VOLUME, SINGLE_COMPONENT, RICIAN*)

for $i = 1$ **to** N_z **do**
  Generate data splits $\Omega^1_{i,j},\ldots,\Omega^C_{i,j}$ and schedule them on G GPUs.
  if *ADAPTIVE_SIGMA* ***then***
    for *each split j of subdata i* ***do***
      allocate and initialize host-side input data, device buffers and streams for asynchronous command execution
      call ***COMPUTE_ADAPTIVE_SIGMA***($\Omega^1_{i,j},\ldots,\Omega^C_{i,j},(\sigma^1_{i,j})^2,\ldots,(\sigma^C_{i,j})^2$, *SIGMA_BLOCK_RADIUS, NLM_WINDOW_RADIUS*)
    end for
    for *each split j of subdata i* ***do***
      call ***NLM_KERNEL***($\Omega^1_{i,j},\ldots,\Omega^C_{i,j},(\sigma^1_{i,j})^2,\ldots,(\sigma^C_{i,j})^2,N_x,N_y,N_z,\tilde{\Omega}^1_{i,j},\ldots,\tilde{\Omega}^C_{i,j}$, *SIGMA_BLOCK_RADIUS,NLM_BLOCK_RADIUS, NLM_WINDOW_RADIUS,ADAPTIVE_SIGMA, VOLUME, SINGLE_COMPONENT, RICIAN*)
    end for
  end if
  collect the restored data $\tilde{\Omega}^1,\ldots,\tilde{\Omega}^C$.
end for

Algorithm 2 COMPUTE_ADAPTIVE_SIGMA($\Omega^1_{i,j},\ldots,\Omega^C_{i,j}$, $(\sigma^1_{i,j})^2$, $\ldots$,$(\sigma^C_{i,j})^2$, SIGMA_BLOCK_RADIUS, NLM_WINDOW_RADIUS)

for all the voxel x_i estimate the local variance **do**
  for each CUDA block $b^i = (b^i_x, b^i_y, 0)$ **do**
    for each image $\Omega^c_{i,j}$ and each component c **do**
      load the block of the image and corresponding aprons to the shared memory
      Syncthreads();
      compute $(\sigma^1_{i,j})^2,\ldots,(\sigma^C_{i,j})^2$ (Section 2.2)
    end for
  end for
end for

required shared memory is $M \approx 3(1+2a_d)WIDTH$ where $WIDTH = (16+2a_d)(8+2a_d)$, so that $M \approx 43KB$

When the required shared memory exceeds the hardware limit (48KB in any configuration), the apron size a_{d_z} has to be accordingly reduced.

4 Results and Discussion

In this section we provide results of experiments aimed at evaluating performance of the NLM algorithm. To this purpose we use sample MR images before and after noise removal. However it is not the purpose of the present paper to assess accuracy of NLM under various settings (e.g., adaptive noise, noise density function), rather we shall focus on the computational time of

Algorithm 3 **NLM_KERNEL**($\Omega^1_{i,j}, \ldots, \Omega^3_{i,j}, (\sigma^1_{i,j})^2, \ldots, (\sigma^C_{i,j})^2, N_x, N_y, N_z$, $\tilde{\Omega}^1_{i,j}, \ldots, \tilde{\Omega}^C_{i,j}$, *SIGMA_BLOCK_RADIUS, NLM_BLOCK_RADIUS, NLM_WINDOW_RADIUS, ADAPTIVE_SIGMA, VOLUME, SINGLE_COMPONENT, RICIAN)*

```
for all the voxel x_i do
  for each CUDA block b^i = (b^i_x, b^i_y, 0) do
    for each image Ω^c_{i,j} do
      load the block of the image and corresponding aprons to the shared memory
      Syncthreads()
      compute the restored intensity ũ^c_i (Eq. (1))
      if RICIAN then
        apply Rician correction to ũ^c_i (Eq. (3))
      end if
      normalize the restored value ũ^c_i and copy it to the global memory
    end for
  end for
end for
```

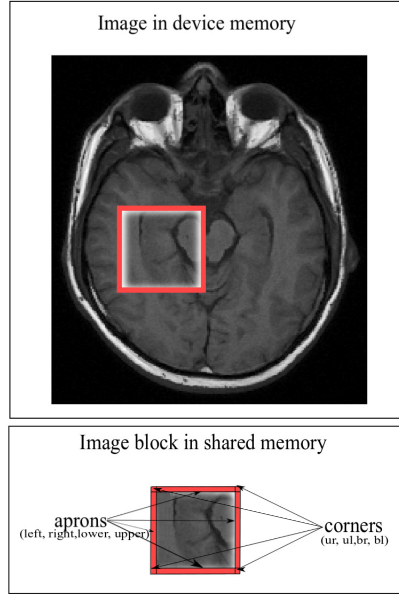

Fig. 1 Illustration of the aprons added to the blocks.

the GPU implementation and on its speed-up with respect to the CPU implementation.

Our GPU computing system consists of 2 commodity NVIDIA GTX 590 cards, each one having 2 GPU units and 1,536 MB of RAM memory; each GPU is equipped with 512 CUDA cores. The NLM algorithm has been implemented within the version 6.0 of the CUDA development toolkit. The code uses single precision arithmetics. As a comparison NLM has also been

implemented in the C++ language on a CPU-based system equipped with an Intel Core i7-2600K CPU @3.40GHz and 16 GB RAM.

As sample images we considered two different experiments. In the first one a realistic digital brain phantom (Alfano et al., 2011), available at `http://lab.ibb.cnr.it`, namely Phantomag, was used. Three different components were used in the evaluation, T1w, T2w and PDw. The original model contains $256 \times 256 \times 150$ near isotropic ($0.9375 \times 0.9375 \times 1$ mm) voxels.

The corresponding simulated signals are provided in the form of T1w (510/15ms TR/TE) and PDw, T2w (1867/1590ms TR/TE) axial slices with a thickness of 4 mm for a conventional spin echo sequence, and 1mm (TR = 9.9 ms, flip angle = 10°, TE = 3.5 ms) for a 3D T1w FFE sequence. Then we used 37 slices of thickness 4 mm.

The second experiment is a real case study consisting of T1w, T2w and PDw axial slices covering the entire brain volume, acquired using Philips Intera 1.5T MR scanner with a 2D conventional spin-echo sequence T1w (510/15ms TR/TE) and PDw, T2w (1867/15-90ms TR/TE) with an in-plane resolution of 0.9375 mm x 0.9375 mm (256x256 acquisition matrix) and thickness 1.20 mm. The Real dataset contains $189 \times 157 \times 130$ voxels.

Tables 1–4 report the GPU running time and the corresponding speed-up under different settings concerning the Volume (2D or 3D), the size of the block d (1 or 2) and search window v (3 or 5). For each instance, 8 runs are executed for different noise models (Gaussian, Rician, spatially uniform, spatially adaptive with a variance window[1] equal to 1 or 2). The range of computational CPU and GPU times and of the corresponding speed-up are shown.

In particular, the seventh and eighth columns indicate how much the parallel versions are faster than the CPU. For practical reasons due to the size of the available shared memory we limit the aprons size a_v of the filter (Algorithm 3) to be less than 6. Tables 1 and 2 refer to the single component algorithm, Tables 3 and 4 concern the 3 components (T1w, T2w and PDw) algorithm version.

In the case of a single component from Table 1 (Phantomag dataset), the minimum speed-up factors reached with 1 GPU and 2 GPUs units are 503× and 758×, respectively. On the Real dataset (Tab. 2) the values drop to 340× and 551×, respectively. This is due to the higher overhead coming from time consuming procedures as splits of data, allocation of kernel memory, data transfer between kernel and local memory, calls to the kernel and gathering of data; this overhead becomes significant when images with smaller size are processed.

Average speed-up for the Phantomag dataset is 1302× for 2 GPUs and 670× for 1 GPU configurations, giving a factor 1.94 close to the ideal value 2; this factor is approximately the same for the Real dataset (1.86).

In the case of 3 components (Tables 3 and 4) the minimum speed-up factors with 1 GPU and 2 GPUs units are 116× and 172× for the Phantomag dataset, respectively. On the Real dataset, the values become 85× and 119×,

[1] Experiments on the spatially adaptive noise variance are run for the only purpose of evaluating the computational time, since the considered data have spatially uniform noise variance.

Table 1 Performance of the single component NLM algorithm on the Phantomag dataset (size 256×256×37), for different volume configurations (2D and 3D, column 1), $d = 1, 2$ (column 2), $v = 3, 5$ (column 3); range of CPU, 2 GPUs, 1 GPU times (seconds; columns 4, 5 and 6, respectively) and corresponding speed-up (columns 7 and 8) for different noise models.

Vol.	d	v	CPU Time	2 GPUs Time	1 GPU Time	2 GPUs speed-up	1 GPU speed-up
3D	2	5	5080–6628	3.8–4	7.6–7.9	1319–1686	659–847
3D	1	5	1561–1624	1.3–1.4	2.6–2.7	1174–1243	599–619
3D	2	3	2241–2870	1.4–1.5	3.3–3.4	1590–1984	677–845
3D	1	3	515–545	0.36–0.42	0.64–0.69	1283–1447	781–819
2D	2	5	174–282	0.14–0.29	0.28–0.55	1005–1271	519–640
2D	1	5	70–145	0.067–0.14	0.13–0.26	931–1335	503–690
2D	2	3	71–141	0.051–0.11	0.12–0.23	1082–1745	507–778
2D	1	3	29–61	0.028–0.062	0.043–0.084	758–1417	567–896

Table 2 Performance of the single component NLM algorithm on the Real dataset (size 189 × 157 × 130), for different volume configurations (2D and 3D, column 1), $d = 1, 2$ (column 2), $v = 3, 5$ (column 3); range of CPU, 2 GPUs, 1 GPU times (seconds; columns 4, 5 and 6, respectively) and corresponding speed-up (columns 7 and 8) for different noise models.

Vol.	d	v	CPU Time	2 GPUs Time	1 GPU Time	2 GPUs speed-up	1 GPU speed-up
3D	2	5	8829–9521	7.9–8.3	19–20	1091–1164	446–472
3D	1	5	2667–2755	2.7–2.9	5.3–5.6	964–1021	493–512
3D	2	3	3695–4721	3–3.2	6–6.3	1189–1482	604–756
3D	1	3	852–884	0.8–1	1.3–1.5	944–1082	626–664
2D	2	5	278–448	0.28–0.58	0.65–1.3	781–1019	345–430
2D	1	5	121–226	0.15–0.31	0.27–0.54	601–1044	340–562
2D	2	3	113–231	0.12–0.25	0.22–0.44	755–1282	422–689
2D	1	3	46–97	0.065–0.2	0.089–0.2	551–948	507–697

Table 3 Performance of the multicomponent NLM algorithm on the Phantomag dataset (size 256×256×37), for different volume configurations (2D and 3D, column 1), $d = 1, 2$ (column 2), $v = 3, 5$ (column 3); range of CPU, 2 GPUs, 1 GPU times (seconds; columns 4, 5 and 6, respectively) and corresponding speed-up (columns 7 and 8) for different noise models.

Vol.	d	v	CPU Time	2 GPUs Time	1 GPU Time	2 GPUs speed-up	1 GPU speed-up
3D	2	5	10846–11325	42–43	63–65	255–270	170–179
3D	1	5	3221–4127	14–14	20–21	229–290	153–193
3D	2	3	4733–5691	17–18	25–26	266–327	188–230
3D	1	3	1075–1102	4.3–4.8	6.1–6.6	231–252	167–178
2D	2	5	359–553	1.5–2.5	2.2–4	220–249	140–166
2D	1	5	137–275	0.63–1.2	1–1.9	187–278	116–188
2D	2	3	146–226	0.59–1.1	0.82–1.6	218–305	146–218
2D	1	3	56–115	0.27–0.52	0.37–0.73	172–278	121–208

Table 4 Performance of the multicomponent NLM algorithm on the Real dataset (size $189 \times 157 \times 130$), for different volume configurations (2D and 3D, column 1), $d = 1, 2$ (column 2), $v = 3, 5$ (column 3); range of CPU, 2 GPUs, 1 GPU times (seconds; columns 4, 5 and 6, respectively) and corresponding speed-up (columns 7 and 8) for different noise models.

Vol.	d	v	CPU Time	2 GPUs Time	1 GPU Time	2 GPUs speed-up	1 GPU speed-up
3D	2	5	18536–18954	108–111	162–166	170–173	114–115
3D	1	5	5393–6077	34–36	51–53	153–175	103–116
3D	2	3	7668–8826	37–39	56–58	201–236	134–156
3D	1	3	1735–1786	9.3–10	14–15	173–189	120–127
2D	2	5	568–872	3.5–6	5.2–8.8	144–174	98–116
2D	1	5	218–421	1.5–2.8	2.2–4	119–187	85–127
2D	2	3	230–354	1.3–2.3	1.8–3.2	157–228	111–158
2D	1	3	88–177	0.59–1.2	0.8–1.5	121–202	93–151

respectively. Phantomag dataset shows an average speed-up of 167× for 1 GPU and of 244× for 2 GPUs configurations; values become 117× and 171× for the Real dataset, respectively. The decrease of speed-up factors for the 3 components algorithm is essentially due to the higher number of data splits (4 times more than the single component algorithm) and consequent kernel calls and related operations.

In general we assert that the speed-up significantly increases when 3D filtering is applied, moderately increases with the size of the window d and it is less affected by the other factors.

As an example Figure 2 shows the restored images and removed noise for the Phantomag dataset under the following configuration: 3D volume, singe and multicomponent, $v = 3$, $d = 1$, Rician noise and uniform variance. Figure 3 refers to the Real dataset under the same configuration. The Figures confirm effectiveness of NLM.

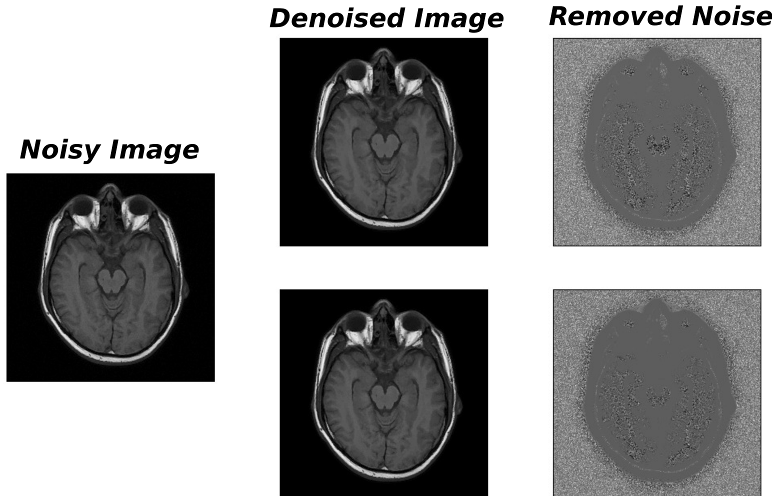

Fig. 2 Examples of NLM filter applied to the 24th slice of the Phantomag experiment. Left: original noisy image. Upper panel: denoised image and corresponding removed noise under the configuration of 3D volume, single component, Rician noise, spatially uniform variance, $v = 3$, $d = 1$. Lower panel: denoised image and corresponding removed noise under the configuration of 3D volume, multicomponent, Rician noise, spatially uniform variance, $v = 3$, $d = 1$.

Now we compare performance of our algorithm with some state-of-art ones, namely Cuomo et al. (2014) and Li et al. (2013).

To be fair, we run our algorithm on data configurations similar to those used in the above mentioned papers. Therefore we considered our single component NLM algorithm with uniform noise variance. We also compare the hardware architectures where algorithms were run.

In Cuomo et al. (2014) an NVIDIA TESLA S2059 processor was used, consisting of 4 GPGPU units each one with 3 GB of RAM and 448 processing cores working at 1.15 GHz, with compute capability of 2.0. Table 5 compares the computational GPU time of our NLM algorithm (with $d = 1$, $v = 5$, block size configuration $(16, 16, 1)$ and $3D$ volume) with the partial and full unrolling algorithms (Cuomo et al., 2014) under several image sizes. The table shows that our algorithm is at least two times faster in almost all cases. The hardware features have similar compute capability as our architecture.

Experiments by Li et al. (2013) are run on a GeForce GTX 660 Ti card, equipped with 1344 processor cores and 2 GB of memory; the compute capability of 3.0. We considered the same similarity and search windows

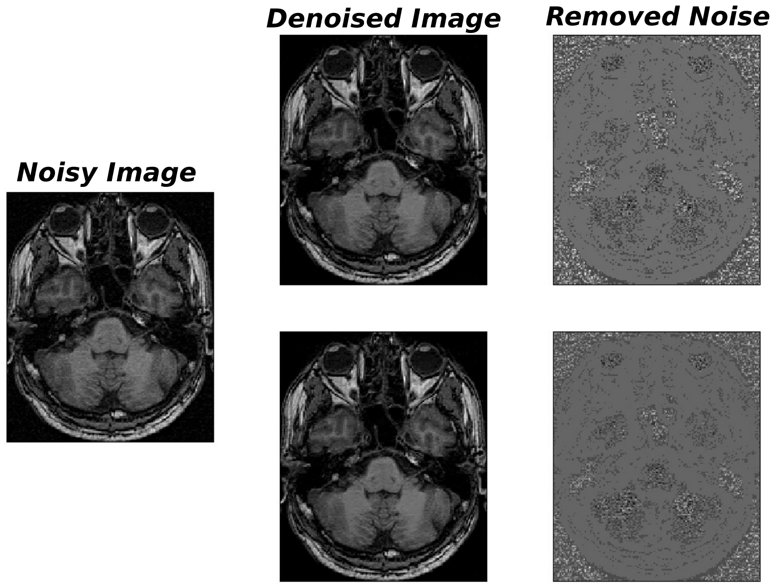

Fig. 3 Examples of NLM filter applied to the 24th slice of the Real data experiment. Left: original noisy image. Upper panel: denoised image and corresponding removed noise under the configuration of 3D volume, single component, Rician noise, spatially uniform variance, $v = 3$, $d = 1$. Lower panel: denoised image and corresponding removed noise under the configuration of 3D volume, multicomponent, Rician noise, spatially uniform variance, $v = 3$, $d = 1$.

($d = 1, 2, 3, 4, 5$; $v = 5$), block size configuration $(16, 16, 1)$ and $3D$ volume. We recall that Li et al. (2013) introduced a parameter Stp_S representing the distance of reference blocks (that in our case is implicitly frozen to 1). Their choice dramatically reduces the total number of patches to be processed in the filter in detriment of the accuracy of the denoising. Two different data sets were considered: Ultrasound fetus, having a resolution of $428 \times 354 \times 209$ pixels, and Carotid artery, with resolution $396 \times 297 \times 338$. Different configurations of d and v were evaluated as reported in Li et al. (2013).

Results are shown in Table 6. We observe that the Li et al. (2013) algorithm is faster than ours for increasing d (but not for $d = 5$). However we recall that our algorithm uses full resolution reference blocks while their number is reduced by Stp_S. For example in the case $d = 4$ the Li et al. (2013) algorithm is 2.7 times faster then ours, but the number of reference blocks is 5 times smaller. We also remark that the hardware features of Li et al. (2013) are more powerful that ours.

Table 5 Computational time (s) of our single component NLM algorithm vs. partial unrolling and full unrolling algorithms by Cuomo et al. (2014). Configuration used is $d = 1$, $v = 5$, block size configuration $(16, 16, 1)$ and $3D$ volume.

	Our algorithm		Partial unrolling alg.		Full unrolling alg.	
Dataset size	1 GPU	2 GPUs	1 GPU	2 GPUs	1 GPU	2 GPUs
64^3	0.56	0.55	0.99	0.49	0.59	0.12
$128^2 \times 64$	1.30	0.79	2.31	1.15	2.33	0.87
128^3	2.67	1.63	1.63	2.31	4.46	1.94
$256^2 \times 128$	8.01	4.81	16.9	8.44	17.9	7.73
256^3	16.25	9.76	33.79	16.9	34.9	16.3
$512^2 \times 128$	39.12	26.07	65.2	32.6	71.4	30.9
$512^2 \times 256$	64.73	38.89	131	65.2	140	65
512^3	129.62	77.90	264	131	276	133

Table 6 Computational time (s) of our single component NLM algorithm vs. Li et al. (2013) on datasets Ultrasound fetus and Carotid artery and different configurations of d and v.

		Ultrasound fetus				Carotid artery dataset			
		Our algorithm		(Li et al., 2013)		Our algorithm		(Li et al., 2013)	
d	v	*Stp_S*	1 GPU	*Stp_S*	1 GPU	*Stp_S*	1 GPU	*Stp_S*	1 GPU
1	5	1	28.73	2	26.66	1	38.01	2	33.58
2	5	1	84.18	3	51.15	1	112.27	3	57.22
3	5	1	154.74	4	75.54	1	207.23	4	90.64
4	5	1	266.86	5	99.11	1	358.27	5	127.96
5	5	1	158.59	6	177.34	1	225.87	6	356.19

5 Conclusions

GPUs represent a cost effective solution to implement suitable algorithms, with computing performances comparable or better than dramatically more expensive cluster computers. NLM, due to its intrinsic massively parallel nature, is prone to be implemented on GPU hardware. Actually NLM is considered as the state-of-art noise removal algorithm. It has been proved to be very effective in applicative problems where images are inherently noisy, like the MR ones considered in the present paper. By its construction, NLM is highly demanding in terms of computational time needed. For example an efficient C++ implementation on modern consumer *Intel i7* architectures requires about half a minute to process a single slice of 256×256 pixels; this makes real time applications impossible and even offline ones impracticable. On the contrary the computational time drops to a few hundredths of second for a 2 GPUs configuration.

In the case of multicomponent methodology the time needed by CPU and 2 GPUs to process a slice is around 5 minutes and a half of second, respectively. Since noise removal often needs a guided trial set of experiments for tuning parameters involved in the methodology (e.g., filtering parameter), the high CPU cost prevents to run enough experiments to this purpose.

The algorithm is a firm basis for future improvements that concern both the hardware platform and the methodology. From one side it can be easily adapted to the next-generation Kepler NVIDIA GPU architecture featuring Compute Capability 3.0 and higher with some limitations in the multi-GPU

environment, due to the exclusive ownership of the device memory. This fixes the main drawback of the present NLM algorithm given by the management of the memory usage. From the other side any advancement in the NLM methodology that is intrinsically massively parallel can be implemented in the algorithm.

The C++ and CUDA algorithms are available from the corresponding author upon request.

References

Alfano B., Comerci M., Larobina M., Prinster A., Hornak JP., Selvan SE., Amato U., Quarantelli M., Tedeschi G., Brunetti A., Salvatore M. (2011) An MRI digital brain phantom for validation of segmentation methods. Medical Image Analysis 15(3):329–339

Buades A., Coll B. and Morel JM. (2005) A review of image denoising algorithms, with a new one. Simul 4:490–530

Buades A., Coll B. and Morel JM. (2008) Nonlocal image and movie denoising. International Journal of Computer Vision 76(2):123–139

Constantinides C. D., Atalar E., and McVeigh E. R. (1997). Signal-to-Noise Measurements in Magnitude Images from NMR Phased Arrays. Magnetic Resonance in Medicine: Official Journal of the Society of Magnetic Resonance in Medicine / Society of Magnetic Resonance in Medicine, 38(5), 852–857.

Coupé P., Hellier P., Prima S., Kervrann C. and Barillot, C. (2008a) 3D wavelet subbands mixing for image denoising. Journal of Biomedical Imaging 2008

Coupé P., Yger P., Prima S., Hellier P., Kervrann C. and Barillot C. (2008b) An optimized blockwise nonlocal means denoising filter for 3-D magnetic resonance images. IEEE Transactions on Medical Imaging 27(4):425–441

Cuomo S., De Michele P. and Francesco P. (2014) 3D data denoising via non-local means filter by using parallel GPU strategies. Computational and Mathematical Methods in Medicine

de Araujo A. F., Constantinou C. E. and Tavares J. M. R.S. (2014) New artificial life model for image enhancement, Expert Systems with Applications,vol 41, Issue 13, 2014, pp 5892–5906

Efros A. and Leung T. (1999) Texture synthesis by non-parametric sampling. In: Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on, vol 2, pp 1033–1038

Gerig G., Kubler O., Kikinis R. and Jolesz F (1992) Nonlinear anisotropic filtering of MRI data. IEEE Transactions on Medical Imaging, 11(2):221–232

Li L., Hou W., Zhang X. and Ding M. GPU-Based Block-Wise Nonlocal Means Denoising for 3D Ultrasound Images. Computational and Mathematical Methods in Medicine, vol. 2013.

Palhano Xavier de Fontes F., Andrade Barroso G. and Coupé, Hellier P. (2011) Real time ultrasound image denoising. Journal of Real-Time Image Processing 6(1):15–22

Griswold M.A., Jakob P.M., Heidemann R.M., Nittka M., Jellus V., Wang J., Kiefer B., Haase A. (2002) Generalized autocalibrating partially parallel acquisitions (GRAPPA). Magnetic Resonance in Medicine 47(6):1202–1210

Goossens B., Luong H., Aelterman J., Pižurica A. and Philips W.(2010) A GPU-Accelerated Real-Time NLMeans Algorithm for Denoising Color Video Sequences. In Advanced Concepts for Intelligent Vision Systems, Lecture Notes in Computer Science, vol 6475, Springer Berlin Heidelberg, pp 46–57

Gudbjartsson H and Patz S (1995) The Rician distribution of noisy MRI data. Magnetic Resonance in Medicine

Huang K., Zhang D. and Wang K. (2009) Non-local means denoising algorithm accelerated by GPU. vol 7497, pp 749711–8

Kharlamov A. and PodlozhnyukV. (2007) Image denoising. Tech. rep., NVIDIA, Inc.

Koay CG. and Basser P. (2006) Analytically exact correction scheme for signal extraction from noisy magnitude MR signals. Journal of Magnetic Resonance 179(2):317–322

Li L., Hou, W., Zhang X. and Ding M.(2013) GPU-Based Block-Wise Nonlocal Means Denoising for 3D Ultrasound Images, Computational and Mathematical Methods in Medicine

Manjón J.V., Carbonell-Caballero J., Lull J.J., García-Martí G., Martí-Bonmatí L. and Robles M. (2008) MRI denoising using non-local means. Medical Image Analysis 12(4):514–523

Manjón J.V., Thacker N.A., and Lull J. J., and García-Martí G., Martí-Bonmatí L. and Robles M. (2009) Multicomponent MR image denoising. International Journal of Biomedical Imaging 2009

Manjón J.V., Coupé P., Martí-Bonmatí L., Collins D. L. and Robles M. (2010) Adaptive non-local means denoising of MR images with spatially varying noise levels Journal of Magnetic Resonance Imaging 31(1): 192–203

Palma G., Comerci M., Alfano B., Cuomo S., De Michele P., Piccialli F. and Borrelli P. (2013) 3D non-local means denoising via multi-GPU. In: Computer Science and Information Systems (FedCSIS), 2013 Federated Conference, pp 495–498

Pruessmann KP., Weiger M., Scheidegger MB. and Boesiger P. (1999) SENSE: Sensitivity encoding for fast MRI. Magnetic Resonance in Medicine 42(5):952–962

Tristán-Vega A. , García-Pérez V., Aja-Fernández S. and Westin C.F. (2012) Efficient and robust nonlocal means denoising of MR data based on salient features matching. Computer Methods and Programs in Biomedicine 105(2):131–144

Wiest-Daesslé N., Prima S., Coupé P., Morrissey S. and Barillot C. (2008) Rician noise removal by non-local means filtering for low signal-to-noise ratio MRI: Applications to DT-MRI. In: Metaxas D, Axel L, Fichtinger G, Székely G (eds) Medical Image Computing and Computer-Assisted Intervention MICCAI 2008, Lecture Notes in Computer Science, vol 5242, Springer Berlin Heidelberg, pp 171–179