A GPU-accelerated real-time NLMeans algorithm for denoising color video sequences

Bart Goossens, Hiêp Luong, Jan Aelterman, Aleksandra Pižurica, and Wilfried Philips*

Ghent University, TELIN-IPI-IBBT
St.-Pietersnieuwstraat 41, 9000 Ghent, Belgium

Abstract. The NLMeans filter, originally proposed by Buades et al., is a very popular filter for the removal of white Gaussian noise, due to its simplicity and excellent performance. The strength of this filter lies in exploiting the repetitive character of structures in images. However, to fully take advantage of the repetitivity a computationally extensive search for similar candidate blocks is indispensable. In previous work, we presented a number of algorithmic acceleration techniques for the NLMeans filter for still grayscale images. In this paper, we go one step further and incorporate both temporal information and color information into the NLMeans algorithm, in order to restore video sequences. Starting from our algorithmic acceleration techniques, we investigate how the NLMeans algorithm can be easily mapped onto recent parallel computing architectures. In particular, we consider the graphical processing unit (GPU), which is available on most recent computers. Our developments lead to a high-quality denoising filter that can process DVD-resolution video sequences in real-time on a mid-range GPU.

1 Introduction

Noise in digital video sequences generally originates from the analogue circuitry (e.g. camera sensors and amplifiers) in video cameras. The noise is mostly visible in bad lighting conditions and using short camera sensor exposure times. Also, video sequences transmitted over analogue channels or stored on magnetic tapes, are often subject to a substantial amount of noise. In the light of the large scale digitization of analogue video material, noise suppression becomes desirable, both to enhance video quality and compression performance.

In the past decades, several denoising methods have been proposed for noise removal, for still images (e.g. [1–6, 11]) or particularly for video sequences (see [7–14]). Roughly speaking, these video denoising methods can be categorized into:

1. *Spatially and temporally local methods* (e.g. [8, 11]): these methods only exploit image correlations in local spatial and temporal windows of fixed size

* B. Goossens and A. Pižurica are postdoctoral researchers of the Fund for Scientific Research in Flanders (FWO), Belgium.

(based on sparsity in a multiresolution representation). The temporal filtering can either be causal or non-causal. In the former case, only past frames are used for filtering. In the latter case, future frames are needed, which can be achieved by introducing a temporal delay.[1]

2. *Spatially local methods with recursive temporal filtering* [9, 10, 14, 15]: these methods rely on recursive filtering that takes advantage of the temporal correlations between subsequent frames. Because usually, first order (causal) infinite impulse response filters are used and no temporal delay is required.
3. *Spatially and temporally non-local methods* [12, 13]: these methods take advantage of repetitive structures that occur both spatially and temporally. Because of computation time and memory restrictions, in practice these methods make use of a *search window* (this is a spatio-temporal window in which similar patches are being searched for). By the practical restrictions, the methods actually fall under the first class, however we expect that by more effcient parallel computing architectures and larger RAM memory the non-locality of these methods will further be extended in the future.

One popular filter that makes use of the repetitive character of structures in video sequences and hence belongs to the third class, is the NLMeans filter [16]. Suppose that an unknown video signal $\boldsymbol{X}(\boldsymbol{p})$ is corrupted by an additive noise process $\boldsymbol{V}(\boldsymbol{p})$, resulting in the observed video signal:

$$\boldsymbol{Y}(\boldsymbol{p}) = \boldsymbol{X}(\boldsymbol{p}) + \boldsymbol{V}(\boldsymbol{p}) \tag{1}$$

Here, $\boldsymbol{p} = [p_x, p_y, p_t]$ is the spatio-temporal position within the video sequence. $\boldsymbol{X}(\boldsymbol{p})$, $\boldsymbol{Y}(\boldsymbol{p})$ and $\boldsymbol{V}(\boldsymbol{p})$ are functions that map values from $\mathbb{Z}^3$ onto the RGB color space $\mathbb{R}^3$. The NLMeans video filter estimates the denoised value of $\boldsymbol{X}(\boldsymbol{p})$ as the weighted average of all pixel intensities in the video sequence:

$$\hat{\boldsymbol{X}}(\boldsymbol{p}) = \frac{\sum_{\boldsymbol{q}\in\delta} w(\boldsymbol{p}, \boldsymbol{p}+\boldsymbol{q})\boldsymbol{Y}(\boldsymbol{p}+\boldsymbol{q})}{\sum_{\boldsymbol{q}\in\delta} w(\boldsymbol{p}, \boldsymbol{p}+\boldsymbol{q})}, \tag{2}$$

where $\boldsymbol{q} = [q_x, q_y, q_t]$ and where the weights $w(\boldsymbol{p}, \boldsymbol{p}+\boldsymbol{q})$ depend on the similarity of patches centered at positions $\boldsymbol{p}$ and $\boldsymbol{p}+\boldsymbol{q}$. δ is a three dimensional search window in which similar patches are searched for. For simplicity of the notation, we assume that $\boldsymbol{Y}(\boldsymbol{p}+\boldsymbol{q})$ is everywhere defined in (2). In practice, we make use of boundary extension techniques (e.g. mirroring) near the image boundaries. Because of the high computational complexity of the NLMeans algorithm (the complexity is quadratic in the number of pixels in the video sequence and linear in the patch size) and because of the fact that the original NLMeans method performed somewhat inferior compared to other (local) state-of-the-art denoising method, many improvements have been proposed by different researchers. Some of these improvements are better similarity measures [17–19], adaptive patch sizes [20], and algorithmic acceleration techniques [4, 19, 21, 22].

[1] A temporal delay is not desirable for certain applications, such as video communication.

In our previous work [4], we proposed a number of improvements to the NLMeans filter, for denoising grayscale still images. Some of these improvements which are relevant for this paper are:

- An extension of the NLMeans to correlated noise: even though the original NLMeans filter relies on a *white* Gaussian noise assumption, the power spectral densities of noise in *real* images and video sequences is rarely flat (see [23]).
- Acceleration techniques that exploit the symmetry in the weight computation and that compute the Euclidean distance between patches by a recursive moving average filter. By these accelerations, the computation time can be reduced by a factor 121 (for 11×11 patches), without sacrifying image quality at all!

In spite of efforts by many researchers and also our recent improvements, the NLMeans algorithm is not well suited for real-time denoising of video sequences on a CPU. Using our improvements, denoising one 512×512 color image takes about 30 sec. for a modestly optimized C++ implementation on a recent 2GHz CPU (single-threaded implementation). Consequently this technique is not applicable to e.g. real-time video communication.

Nowadays, there is a trend toward the use of parallel processing architectures in order to accelerate the processing. One example of such architecture is the graphical processing unit (GPU). Although the GPU is primarily designed for the rendering of 3D scenes, advances of the GPU in the late 90's enabled many researchers and engineers to use the GPU for more general computations. This led to the so-called GPGPU (General-Purpose computations on GPUs) [24] and many approaches (e.g. based on OpenGL, DirectX, CUDA, OpenCL, ...) exist to achieve GPGPU with existing GPU hardware. Also because the processing power of modern GPUs has tremendously increased in the last decade (even for inexpensive GPUs a speed-up of a factor $20\times$ to $100\times$ can be expected) and is even more improving, it becomes worthwhile to investigate which video denoising methods can efficiently be implemented on a GPU.

Recently, a number of authors have implemented the NLMeans algorithm on a GPU: in [25] a locally constant weight assumption is used in the GPU implementation to speed up the basic algorithm. In [26], a GPU extension of the NLMeans algorithm is proposed to denoise ultrasound images. In this approach, the maximum patch size is limited by the amount of shared memory of the GPU.

In this paper, we focus on algorithmic acceleration techniques for the GPU without sacrificing denoising quality, i.e., the GPU implementation computes the exact NLMeans formula, and without patch size restrictions imposed by the hardware. To do so, we first review how NLMeans-based algorithms can be mapped onto parallel processing architectures. We will find that the core ideas of our NLMeans algoritmic acceleration techniques are directly applicable, but the algorithms themselves need to be modified. By these modifications, we will see that the resulting implementation can process DVD video in real-time on a mid-range GPU. Next, as a second contribution of this paper, we explain how

the filter can be used to remove correlated noise (both spatially as across color channels) from video sequences.

The outline of this paper is as follows: on Section 2, we first review some basic GPGPU programming concepts. Next, we develop an efficient NLMeans algorithm for a GPU and its extension to deal with noise which is correlated across color channels. In Section 3 we give experimental results our method. Finally, Section 4 concludes this paper.

2 An efficient NLMeans algorithm for a GPU

In this Section, we will explain the algorithmic improvements that we made to the NLMeans filter in order to efficiently run the algorithm on a GPU. As already mentioned, many approaches and/or programming language extensions exist for GPGPU programming. Because the GPU technology is quickly evolving, we will present a description of the algorithm that is quite general and that does not rely on specific GPU technology choices. This way, the algorithms we present can still be useful in the future, when newer GPU architectures become available.

2.1 General GPGPU concepts

One core element in GPGPU techniques is the so-called *kernel function.* A *kernel function* is a function that evaluates the output pixel intensities for a specific position in the output image (or even multiple output images) and that takes as input both the position ($\boldsymbol{p}$) in the video sequence, and a number of input images (which we will denote as $\boldsymbol{U}_1^{(i)}, ..., \boldsymbol{U}_K^{(i)}$). A GPGPU program can then be considered to be a cascade of kernel functions $\boldsymbol{f}^{(I)} \circ \boldsymbol{f}^{(I-1)} \circ \cdots \circ \boldsymbol{f}^{(1)}$ applied to a set of input images. Mathematically, the evaluation of one such kernel function (which we will call a *pass*) can be expressed as:

$$\left[\boldsymbol{U}_1^{(i+1)}, ..., \boldsymbol{U}_K^{(i+1)}\right](\boldsymbol{p}) = \boldsymbol{f}^{(i)}_{\boldsymbol{U}_1^{(i)}, ..., \boldsymbol{U}_K^{(i)}}(\boldsymbol{p}) \tag{3}$$

where the kernel function takes as input the output images of the previous pass, $\boldsymbol{U}_1^{(i)}, ..., \boldsymbol{U}_K^{(i)}$ and subsequently computes the inputs for the next pass, $\boldsymbol{U}_1^{(i+1)}, ..., \boldsymbol{U}_K^{(i+1)}$. More specifically, the kernel function $\boldsymbol{f}^{(i)}$ maps a spatio-temporal coordinate ($\boldsymbol{p}$) onto a three-dimensional RGB color vector.

Now, porting an algorithm to the GPU comes down to converting the algorithm into a finite, preferably low number of passes as defined in (3) and with fairly simple functions $\boldsymbol{f}^{(i)}$:

$$\begin{aligned}
\left[\boldsymbol{U}_1^{(2)}, ..., \boldsymbol{U}_K^{(2)}\right](\boldsymbol{p}) &= \boldsymbol{f}^{(1)}_{\boldsymbol{U}_1^{(1)}, ..., \boldsymbol{U}_K^{(1)}}(\boldsymbol{p}), \\
\left[\boldsymbol{U}_1^{(3)}, ..., \boldsymbol{U}_K^{(3)}\right](\boldsymbol{p}) &= \boldsymbol{f}^{(2)}_{\boldsymbol{U}_1^{(2)}, ..., \boldsymbol{U}_K^{(2)}}(\boldsymbol{p}), \\
&\vdots \quad \vdots \\
\left[\boldsymbol{U}_1^{(I+1)}, ..., \boldsymbol{U}_K^{(I+1)}\right](\boldsymbol{p}) &= \boldsymbol{f}^{(I)}_{\boldsymbol{U}_1^{(I)}, ..., \boldsymbol{U}_K^{(I)}}(\boldsymbol{p}).
\end{aligned} \tag{4}$$

We remark that not all passes need to process all input images, i.e. it is completely legal that $\boldsymbol{U}_1^{(i+1)} = \boldsymbol{U}_1^{(i)}$. In this case, we express this formally by saying that the function $\boldsymbol{f}^{(i)}_{\boldsymbol{U}_1^{(i)},\ldots,\boldsymbol{U}_K^{(i)}}(\boldsymbol{p})$ is constant in $\boldsymbol{U}_1^{(i)}$.

2.2 Straightforward GPU implementation of the NLMeans filter

First, we will show that a straightforward (naive) implementation of the traditional NLMeans filter from [13,16] leads to a very high number of passes, hence an algorithm that is inefficient even on the GPU. Next, we will explain how our own algorithmic accelerations can be converted into a program for the GPU as in equation (4). We will do this for a broad range of weighting functions that are a function of the Euclidean distance measure between two patches:

$$w(\boldsymbol{p}, \boldsymbol{p}+\boldsymbol{q}) = g\left(\sum_{(\Delta x,\Delta y)\in[-B,\ldots,B]^2} \left\| \boldsymbol{r}_{\boldsymbol{p},\boldsymbol{q}}^{(\Delta x,\Delta y)} \right\|^2\right) \tag{5}$$

with $\boldsymbol{r}_{\boldsymbol{p},\boldsymbol{q}}^{(\Delta x,\Delta y)} = \boldsymbol{Y}(p_x+q_x+\Delta x, p_y+q_y+\Delta y, p_t+q_t) - \boldsymbol{Y}(p_x+\Delta x, p_y+\Delta y, p_t)$, with $(2B+1)\times(2B+1)$ the patch size and where the function $g(r)$ has the property that $g(0)=1$ (such that the weight $w=1$ if the Euclidean distance between two patches is zero, i.e., for similar patches) and $\lim_{r\to\infty} g(r) = 0$ (the weight $w=0$ for dissimilar patches). In particular, we consider the Bisquare robust weighting function, for which $g(r)$ is defined as follows:

$$g(r) = \begin{cases} \left(1-(r/h)^2\right)^2 & r \le h \\ 0 & r > h \end{cases},$$

with h a constant parameter that is fixed in advance (for more details, see [4]). Substituting (5) into (2) gives:

$$\hat{\boldsymbol{X}}(\boldsymbol{p}) = \frac{\sum_{\boldsymbol{q}\in\delta} g\left(\sum_{(\Delta x,\Delta y)\in[-B,\ldots,B]^2} \left\| \boldsymbol{r}_{\boldsymbol{p},\boldsymbol{q}}^{(\Delta x,\Delta y)} \right\|^2\right) \boldsymbol{Y}(\boldsymbol{p}+\boldsymbol{q})}{\sum_{\boldsymbol{q}\in\delta} g\left(\sum_{(\Delta x,\Delta y)\in[-B,\ldots,B]^2} \left\| \boldsymbol{r}_{\boldsymbol{p},\boldsymbol{q}}^{(\Delta x,\Delta y)} \right\|^2\right)}. \tag{6}$$

Comparing (6) to (3) immediately leads to the kernel function:

$$\boldsymbol{f}^{(1)}_{\boldsymbol{U}_1^{(1)}}(\boldsymbol{p}) = \frac{\sum_{\boldsymbol{q}\in\delta} g\left(\sum_{(\Delta x,\Delta y)\in[-B,\ldots,B]^2} \left\| \boldsymbol{r}_{\boldsymbol{p},\boldsymbol{q}}^{(\Delta x,\Delta y)} \right\|^2\right) \boldsymbol{U}_1^{(1)}(\boldsymbol{p}+\boldsymbol{q})}{\sum_{\boldsymbol{q}\in\delta} g\left(\sum_{(\Delta x,\Delta y)\in[-B,\ldots,B]^2} \left\| \boldsymbol{r}_{\boldsymbol{p},\boldsymbol{q}}^{(\Delta x,\Delta y)} \right\|^2\right)}, \tag{7}$$

with $\boldsymbol{U}_1^{(1)}(\boldsymbol{p}) = \boldsymbol{Y}(\boldsymbol{p})$. We see that the number of operations performed by the kernel function is linear in $|\delta|\,(2B+1)^2$, with $|\delta|$ the cardinality of δ. Although this approach seems feasible, some GPU hardware (especially less recent GPU

hardware) puts limits on the number of operations (more specifically, processor instructions) performed by a kernel function. To work around this restriction, we make use of a weight accumulation buffer (see [19]) and convert every term of the summations in (7) into a separate pass, in which in each pass, one term of the summation $\sum_{\boldsymbol{q}\in\delta}$ is added to the accumulation buffer. This is done for both the numerator and denominator of (7). For $i = 1, ..., |\delta|$, with constants $\boldsymbol{q}_i$ defined for each pass (e.g. using raster scanning), we obtain the kernel function:

$$\boldsymbol{f}^{(i)}_{\boldsymbol{U}_1^{(i)},...,\boldsymbol{U}_3^{(i)}}(\boldsymbol{p}) = \begin{pmatrix} \boldsymbol{U}_1^{(i)}(\boldsymbol{p}) \\ \boldsymbol{U}_2^{(i)} + g\left(\sum_{(\Delta x,\Delta y)\in[-B,...,B]^2} \left\|\boldsymbol{r}^{(\Delta x,\Delta y)}_{\boldsymbol{p},\boldsymbol{q}_i}\right\|^2\right)\boldsymbol{U}_1^{(i)}(\boldsymbol{p}+\boldsymbol{q}_i) \\ \boldsymbol{U}_3^{(i)} + g\left(\sum_{(\Delta x,\Delta y)\in[-B,...,B]^2} \left\|\boldsymbol{r}^{(\Delta x,\Delta y)}_{\boldsymbol{p},\boldsymbol{q}_i}\right\|^2\right) \end{pmatrix} \tag{8}$$

where $\boldsymbol{U}_2^{(i)}$ is an accumulation buffer for the denoised image, and where $\boldsymbol{U}_3^{(i)}$ is a weight accumulation buffer (initially, $\boldsymbol{U}_2^{(1)}(\boldsymbol{p}) = \boldsymbol{U}_3^{(1)}(\boldsymbol{p}) = \boldsymbol{0}$). Next, one last pass is required, to compute the final output image:

$$\boldsymbol{f}^{(I)}_{\boldsymbol{U}_1^{(I)},...,\boldsymbol{U}_3^{(I)}}(\boldsymbol{p}) = \begin{pmatrix} \dfrac{\boldsymbol{U}_2^{(I)}(\boldsymbol{p})}{\boldsymbol{U}_3^{(I)}(\boldsymbol{p})} & \boldsymbol{0} & \boldsymbol{0} \end{pmatrix}^T, \tag{9}$$

with $I = |\delta| + 1$. The number of operations per pass is now multiplied by a factor $1/|\delta|$, but is still very high. To further reduce this number of operations, we could apply a similar split-up technique and convert the summation $\sum_{(\Delta x,\Delta y)\in[-B,...,B]^2}$ into several passes. We note that, even though this way we would obtain a working algorithm for most available GPUs, the number of passes $I = |\delta|\left((2B+1)^2+1\right)+1$ becomes very high. For example, for a $31 \times 31 \times 4$-search window and $B = 4$, we obtain $I = 311365$ passes. If for each video frame, a single pass of the algorithm would take 0.1 msec. on a GPU, the complete algorithm would still require approx. 31 sec. for processing one single frame of a video sequence, which is similar to the computation time of our CPU version mentioned in Section 1. Hence, further algorithmic accelerations are required.

2.3 Actual implementation using algorithmic accelerations

In [4], we pointed out that the term $\sum_{(\Delta x,\Delta y)\in[-B,...,B]^2} \left\|\boldsymbol{r}^{(\Delta x,\Delta y)}_{\boldsymbol{p},\boldsymbol{q}}\right\|^2$ can be interpreted as a convolution operator with a filter kernel with square support. Consequently the Euclidean distance between two patches can efficiently be computed using a moving average filter, and the algorithmic complexity is reduced with roughly a factor $(2B+1)^2/2$. Unfortunately, converting a moving average filter directly into a GPU program as in (4) is not feasible in a small number of passes. Instead, we exploit the separability of the filter kernel and we implement the convolution operator as a cascade of a horizontal and vertical filter. Then by

setting $\boldsymbol{U}_1^{(1)}(\boldsymbol{p}) = \boldsymbol{Y}(\boldsymbol{p})$, $\boldsymbol{U}_2^{(1)}(\boldsymbol{p}) = \boldsymbol{U}_3^{(1)}(\boldsymbol{p}) = \boldsymbol{U}_4^{(1)}(\boldsymbol{p}) = \boldsymbol{0}$, the first pass of our algorithm is as follows:

$$\boldsymbol{f}^{(4i-3)}_{\boldsymbol{U}_1^{(4i-3)},\ldots,\boldsymbol{U}_4^{(4i-3)}}(\boldsymbol{p}) = \begin{pmatrix} \boldsymbol{U}_1^{(4i-3)}(\boldsymbol{p}) \\ \boldsymbol{U}_2^{(4i-3)}(\boldsymbol{p}) \\ \boldsymbol{U}_3^{(4i-3)}(\boldsymbol{p}) \\ \left\| \boldsymbol{r}^{(0,0)}_{\boldsymbol{p},\boldsymbol{q}_i} \right\|^2 \end{pmatrix}. \tag{10}$$

Note that the values $\boldsymbol{U}_1^{(4i-3)}(\boldsymbol{p})$, $\boldsymbol{U}_2^{(4i-3)}(\boldsymbol{p})$, $\boldsymbol{U}_3^{(4i-3)}(\boldsymbol{p})$ are simply passed to the next step of the algorithm. We only compute the Euclidean distance between two pixel intensities (in RGB color space). The next passes are given by:

$$\begin{aligned} \boldsymbol{f}^{(4i-2)}_{\boldsymbol{U}_1^{(4i-2)},\ldots,\boldsymbol{U}_4^{(4i-2)}}(\boldsymbol{p}) &= \begin{pmatrix} \boldsymbol{U}_1^{(4i-2)}(\boldsymbol{p}) \\ \boldsymbol{U}_2^{(4i-2)}(\boldsymbol{p}) \\ \boldsymbol{U}_3^{(4i-2)}(\boldsymbol{p}) \\ \sum_{\Delta x \in [-B,\ldots,B]} \boldsymbol{U}_4^{(4i-2)}(p_x + \Delta x, p_y, p_t) \end{pmatrix}, \\ \boldsymbol{f}^{(4i-1)}_{\boldsymbol{U}_1^{(4i-1)},\ldots,\boldsymbol{U}_4^{(4i-1)}}(\boldsymbol{p}) &= \begin{pmatrix} \boldsymbol{U}_1^{(4i-1)}(\boldsymbol{p}) \\ \boldsymbol{U}_2^{(4i-1)}(\boldsymbol{p}) \\ \boldsymbol{U}_3^{(4i-1)}(\boldsymbol{p}) \\ g\left(\sum_{\Delta y \in [-B,\ldots,B]} \boldsymbol{U}_4^{(4i-1)}(p_x, p_y + \Delta y, p_t) \right) \end{pmatrix}. \end{aligned} \tag{11}$$

The separable filtering reduces the computation complexity by a factor $(2B + 1)/2$. Fortunately, the steps (11) are computationally simple and only require a small number regular memory accesses, which can benefit from the internal memory caches of the GPU. Note that in the last step of (11), we already computed the similarity weights, by evaluating the function $g(\cdot)$.

A second acceleration technique we presented in [19], is to exploit the symmetry property of the weights, i.e. $w(\boldsymbol{p}, \boldsymbol{p} + \boldsymbol{q}_i) = w(\boldsymbol{p} + \boldsymbol{q}_i, \boldsymbol{p})$. To do so, when adding $w(\boldsymbol{p}, \boldsymbol{p} + \boldsymbol{q}_i)\boldsymbol{Y}(\boldsymbol{p} + \boldsymbol{q}_i)$ to the image accumulation buffer at position $\boldsymbol{p}$, we proposed to additionally add $w(\boldsymbol{p}, \boldsymbol{p} + \boldsymbol{q}_i)\boldsymbol{Y}(\boldsymbol{p})$ to the image accumulation buffer at position $\boldsymbol{p} + \boldsymbol{q}_i$. Consequently, the weight $w(\boldsymbol{p}, \boldsymbol{p} + \boldsymbol{q}_i)$ only needs to be computed *once*, effectively halving the size of the search window δ. However, this acceleration technique requires "non-regular" writes to the accumulation buffer, i.e., at position $\boldsymbol{p} + \boldsymbol{q}_i$ instead of $\boldsymbol{p}$ as required by the structure of our GPU program (4). Fortunately, our specific notation here brings a solution here: by noting that $\boldsymbol{q}_i$ is constant in each pass, we could simply translate the input coordinates and perform a "regular" write to the accumulation buffer. This way, we need to add $w(\boldsymbol{p} - \boldsymbol{q}_i, \boldsymbol{p})\boldsymbol{Y}(\boldsymbol{p} - \boldsymbol{q}_i)$ to the accumulation buffer at position $\boldsymbol{p}$. We will call this the *translation* technique. This gives us the next step of our

GPU algorithm:

$$\boldsymbol{f}^{(4i)}_{\boldsymbol{U}_1^{(4i)},\ldots,\boldsymbol{U}_4^{(4i)}}(\boldsymbol{p}) = \begin{pmatrix} \boldsymbol{U}_1^{(4i)}(\boldsymbol{p}) \\ \boldsymbol{U}_2^{(4i)} + \boldsymbol{U}_4^{(4i)}(\boldsymbol{p})\boldsymbol{U}_1^{(4i)}(\boldsymbol{p}+\boldsymbol{q}_i) + \boldsymbol{U}_4^{(4i)}(\boldsymbol{p}-\boldsymbol{q}_i)\boldsymbol{U}_1^{(4i)}(\boldsymbol{p}-\boldsymbol{q}_i)\left[1-\delta(\boldsymbol{q}_i)\right] \\ \boldsymbol{U}_3^{(4i)} + \boldsymbol{U}_4^{(4i)}(\boldsymbol{p}) + \boldsymbol{U}_4^{(4i)}(\boldsymbol{p}-\boldsymbol{q}_i)\left[1-\delta(\boldsymbol{q}_i)\right] \\ \boldsymbol{U}_4^{(4i)}(\boldsymbol{p}) \end{pmatrix} \tag{12}$$

with $\delta(\cdot)$ the Dirac delta function. The Dirac delta function is needed here, to prevent the weights $w(\boldsymbol{p},\boldsymbol{p})$ to be counted twice. In the last pass, again the image accumulation buffer intensities are divided by the accumulated weights, which gives:

$$\boldsymbol{f}^{(I)}_{\boldsymbol{U}_1^{(I)},\ldots,\boldsymbol{U}_4^{(I)}}(\boldsymbol{p}) = \begin{pmatrix} \dfrac{\boldsymbol{U}_2^{(I)}(\boldsymbol{p})}{\boldsymbol{U}_3^{(I)}(\boldsymbol{p})} & \boldsymbol{0}\ \boldsymbol{0}\ \boldsymbol{0} \end{pmatrix}^T, \tag{13}$$

with $I = 4\left(|\delta|+1\right)/2 + 1 = 2\,|\delta| + 3$. The output of the NLMeans algorithm is then $\hat{\boldsymbol{X}}(\boldsymbol{p}) = \boldsymbol{U}_2^{(I)}(\boldsymbol{p})/\boldsymbol{U}_3^{(I)}(\boldsymbol{p})$. Consequently, the complete NLMeans algorithm comprises the passes $i = 1, ..., I$ defined by steps (10)-(13).

2.4 Extension to noise correlated across color channels

In this Section, we briefly explain how our GPU-NLMeans algorithm can be extended to deal with Gaussian noise that is correlated across color channels. Our main goal here is to show that our video algorithm is not restricted to white Gaussian noise. Because of space limitations, visual and quantitative results for color images and color video will be reported in later publications. As we pointed out in [4, p. 6], the algorithm can be extended to spatially correlated noise by using a Mahalanobis distance based on the noise covariance matrix instead of the Euclidean distance similarity metric. When dealing with noise which is correlated across color channels, we need to replace (5) by:

$$w(\boldsymbol{p},\boldsymbol{p}+\boldsymbol{q}) = g\left(\sum_{(\Delta x,\Delta y)\in[-B,\ldots,B]^2} \left(\boldsymbol{r}_{\boldsymbol{p},\boldsymbol{q}}^{(\Delta x,\Delta y)}\right)^T \boldsymbol{C}^{-1}\left(\boldsymbol{r}_{\boldsymbol{p},\boldsymbol{q}}^{(\Delta x,\Delta y)}\right)\right)$$

with $\boldsymbol{C}$ the noise covariance function. In practice, the matrix $\boldsymbol{C}$ can be estimated from flat regions in the video sequence, or based on an EM-algorithm as in [27]. Now, by introducing the decorrelating color transform $\boldsymbol{G} = \boldsymbol{C}^{-1/2}$, and by defining:

$$\boldsymbol{r}_{\boldsymbol{p},\boldsymbol{q}}^{'(\Delta x,\Delta y)} = \boldsymbol{GY}(p_x+q_x+\Delta x, p_y+q_y+\Delta y, p_y+q_y) - \boldsymbol{GY}(p_x+\Delta x, p_y+\Delta y, p_t),$$

the weighting function can again be expressed in terms of the Euclidean distance $\left\|\boldsymbol{r}_{\boldsymbol{p},\boldsymbol{q}}^{'(\Delta x,\Delta y)}\right\|^2$. Hence, removing correlated noise from video sequences solely requires a color transform $\boldsymbol{G}$ applied as pre-processing to the video sequence. Furthermore, this technique can be combined with our previous approach from [4, p.

6] in order to remove Gaussian noise which is both spatially correlated and correlated across color channels.

2.5 Discussion

To optimize the computational performance of a GPU program, minimizing the number of passes I and performing more operations in each kernel function is more beneficial than optimizing the individual kernel functions themselves, especially when the kernel functions are relatively simple (as in our algorithm in Subsection 2.3). This is due to GPU memory caching behavior and also because every pass typically requires interaction with the CPU (for example, the computation time of an individual pass can be affected by the process CPU scheduling granularity). To assess the computational performance improvement, a possible solution would be to use theoretical models to predict the performance. Unfortunately, these theoretical models are very dependent on the underlying GPU architecture: the computational performance can not simply be expressed as a function of the total number of floating point operations, because of the parallel processing. To obtain a rough idea of the computational performance we use the actual number of passes required by our algorithm. For example, when comparing our algorithmic accelerations from Subsection 2.3 to the naive NLMeans-algorithm from Subsection 2.2, we see that the number of passes is reduced with a factor:

$$\frac{|\delta|\left((2B+1)^2+1\right)+1}{4\left(|\delta|+1\right)/2+1} \approx \frac{(2B+1)^2}{2}.$$

For patches of size 9×9, the accelerated NLMeans GPU algorithm requires approximately 40 times less processing passes.

Another point of interest is the streaming behavior of the algorithm: for real-time applications, it is required the algorithm processes video frames as soon as they become available. In our algorithm, this can be completely controlled by adjusting the size of the search window. Suppose we choose:

$$\delta = [-A, ..., A] \times [-A, ..., A] \times [-D_{\text{past}}, ..., D_{\text{future}}]$$

with $A, D_{\text{past}}, D_{\text{future}} \geq 0$ positive constants. A determines the size of the spatial search window; D_{past} and D_{future} are respectively the number of past and future frames that the filter uses for denoising the current frame. For causal implementation of the filter, a delay of D_{future} frames is required. Of course, D_{future} can even be zero, if desired. However, the main disadvantage of a zero delay is that the translation technique from Subsection 2.3 cannot be used in the temporal direction, because the translation technique in fact requires the updating of future frames in the accumulation buffer. Nevertheless, using a small positive D_{future}, a trade-off can be made between the filter delay and the algoritmic acceleration achieved by exploiting the weight symmetry. The number of video frames in GPU memory is at most $4\left(D_{\text{past}} + D_{\text{future}} + 1\right)$.

3 Experimental results

To demonstrate the processing time improvement of our GPU algorithm with the proposed accelerations, we apply our technique to a color video sequence of resolution 720×480 (a resolution which is common for DVD-video). The video sequence is corrupted with artificially added stationary white Gaussian noise with standard deviation $25/255$ (input PSNR 20.17dB). We compare the processing time of our proposed GPU implementation to the modestly optimized (single-threaded) C++ CPU implementation from our previous work [4] (including all acceleration techniques proposed in [4]), for different values of the parameters A and D_{past}. For these results, we use $D_{\text{future}} = 0$ (resulting in a zero-delay denoising filter, as explained in Subsection 2.5), $B = 4$ (corresponding to 9×9 patches) and we manually select h to optimize the PSNR ratio. In particular, we use $h = 0.13$ for $A \leq 3$ and $h = 0.16$ for $A > 3$ (note that the pixel intensities are within the range $0 - 1$).

Both the CPU and GPU version were run on the same computer, which is equipped with a 2.4GHz Intel Core(2) processor with 2048 MB RAM and a NVidia GeForce 9600GT GPU. This card has 64 parallel stream processing units and is considered to be a mid-range GPU.The GPU algorithm is implemented as a HLSL pixel shader in DirectX 9.1 (Windows XP) and makes use of 16-bit floating point values. The main program containing the GPU host code, is written in C# 3.0.

Processing time and output PSNR results (obtained after denoising) are reported in Table 1. We only report PSNR results for the GPU denoising technique, since both CPU and GPU algorithms essentially compute the same formula (i.e. equation (2)). It can be seen that the PSNR values increase when using a larger search window or a larger number of past frames. This is simply because more similar candidate blocks become available for searching, and consequently better estimates can be found for the denoised pixel intensities. Remarkable is also the huge acceleration of the GPU compared to the CPU of a factor 200 to 400. The main reason lies in the massive amount of parallellism in the NLMeans algorithm, which can be fully exploited by the GPU but hardly by the CPU. Especially this huge acceleration leads to a real-time denoising filter. We can determine the optimal parameters for the algorithm by selecting a minimum frame rate and by maximizing the output PSNR of the filter for this minimum frame rate. For our results in Table 1, an optimal combination is a 7×7-search window and $D_{\text{past}} = 1$, in order to attain a frame rate of 25 frames per second (fps).

4 Conclusion

In this paper, we have shown how the traditional NLMeans algorithm can be efficiently mapped onto a parrallel processing architecture such as the GPU. We saw that a naive straightforward implementation inevitably leads to an inefficient algorithm with a huge number of parallel processing passes. We then analyzed our NLMeans algorithmic acceleration techniques from previous work, and we

Table 1. Experimental results for denoising a color video sequence, consisting of 99 frames of dimensions 720×480 and corrupted with additive stationary white Gaussian noise with standard deviation 25/255 (PSNR=20.17dB).

Parameters			GPU			CPU	GPU vs. CPU
A	Search window	D_{past}	FPS	msec/frame	PSNR [dB]	msec/frame	acceleration
2	5x5	0	100.00	10.00	33.09	4021	402.10×
2	5x5	1	69.57	14.37	34.42	N/A	
2	5x5	2	50.79	19.69	34.88	N/A	
2	5x5	3	40.34	24.79	35.04	N/A	
3	7x7	0	52.46	19.06	35.40	7505	393.70×
3	**7x7**	**1**	**30.38**	**32.92**	**36.19**	**N/A**	
3	7x7	2	21.43	46.67	36.26	N/A	
3	7x7	3	16.58	60.31	36.33	N/A	
5	11x11	0	18.46	54.17	36.34	18230	336.55×
5	11x11	1	10.22	97.81	37.11	N/A	
5	11x11	2	7.07	141.35	37.26	N/A	
5	11x11	3	5.42	184.48	37.23	N/A	
10	21x21	0	4.32	231.56	36.79	50857	219.63×
10	21x21	1	2.36	424.27	37.20	N/A	
10	21x21	2	1.62	615.73	37.24	N/A	
10	21x21	3	1.24	805.83	37.17	N/A	

noted that these techniques can not be applied "as is". Therefore, we adapted the core ideas of these acceleration techniques (i.e. the moving averaging filter for the fast computation of Euclidean distances and the exploitation of the weight symmetry) to GPGPU programming methodology and we arrived at a GPU-NLMeans algorithm that is two to three orders of magnitudes faster (depending on the parameter choices) than the equivalent CPU algorithm. This technique can process video sequences in real-time on a mid-range GPU.

References

1. L. Rudin and S. Osher, "Total variation based image restoration with free local constraints," in *IEEE Int. Conf. Image Proc. (ICIP)*, Nov. 1994, vol. 1, pp. 31–35.
2. J. Portilla, V. Strela, M. Wainwright, and E.P. Simoncelli, "Image denoising using scale mixtures of gaussians in the wavelet domain," *IEEE Trans. Image Processing*, vol. 12, no. 11, pp. 1338–1351, 2003.
3. K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian, "Image denoising by sparse 3d transform-domain collaborative filtering," *IEEE Trans. Image Processing*, vol. 16, no. 8, pp. 2080–2095, 2007.
4. B. Goossens, H. Luong, A. Pižurica, and W. Philips, "An improved Non-Local Means Algorithm for Image Denoising," in *2008 Int. Workshop on Local and Non-Local Approx. in Image Processing*, 2008, (invited paper).
5. B. Goossens, A. Pižurica, and W. Philips, "Removal of correlated noise by modeling the signal of interest in the wavelet domain," *IEEE Trans. Image Processing*, vol. 18, no. 6, pp. 1153–1165, jun 2009.

6. B. Goossens, A. Pižurica, and W. Philips, "Image Denoising Using Mixtures of Projected Gaussian Scale Mixtures," *IEEE Trans. Image Processing*, vol. 18, no. 8, pp. 1689–1702, aug 2009.
7. J.C. Brailean, R. P. Kleihorst, S. Efstraditis, K. A. Katsaggeleos, and R. L. Lagendijk, "Noise reduction filters for dynamic image sequences: a review," *Proc. IEEE*, vol. 83, no. 9, pp. 1272–1292, 1995.
8. Ivan W. Selesnick and Ke Yong Li, "Video denoising using 2D and 3D dual-tree complex wavelet transforms," *Proc. SPIE Wavelet Applications in Signal and Image Processing*, pp. 607–618, August 2003.
9. A. Pižurica, V. Zlokolica, and W. Philips, "Combined wavelet domain and temporal video denoising," *Proc. IEEE Int. Conf. on Advanced Video and Signal based Surveillance (AVSS)*, pp. 334–341, 2003.
10. V. Zlokolica, A. Pižurica, and W. Philips, "Recursive temporal denoising and motion estimation of video," in *IEEE Int. Conf. Image Proc. (ICIP)*, 2004, pp. 1465–1468.
11. B. Goossens, A. Pižurica, and W. Philips, "Video denoising using motion-compensated lifting wavelet transform," in *Proceedings of Wavelets and Applications Semester and Conference (WavE2006)*, Lausanne, Switzerland, 2006.
12. K. Dabov, A. Foi, and K. Egiazarian, "Video denoising by sparse 3D transform-domain collaborative filtering," in *European Signal Processing Conference (EUSIPCO-2007)*, Poznan, Poland, 2007.
13. A. Buades, B. Coll, and J.-M. Morel, "Nonlocal Image and Movie Denoising," *Int J. Comput. Vis.*, vol. 76, pp. 123–139, 2008.
14. S. Yu, M. O Ahmad, and M.N.S. Swamy, "Video Denoising using Motion Compensated 3D Wavelet Transform with Integrated Recursive Temporal Filtering," *IEEE Trans. Cir. and Sys. for Video Technol.*, 2010, In press.
15. T. Mélange, M. Nachtegael, E. E. Kerre, V. Zlokolica, S. Schulte, V. De Witte, A. Pizurica, and W. Philips, "Video denoising by fuzzy motion and detail adaptive averaging," *Journal of Elec. Imaging*, vol. 17, no. 4, pp. 43005–01 – 43005–19, 2008.
16. A. Buades, B. Coll., and J.M Morel, "A non local algorithm for image denoising," in *Proc. Int. Conf. Comp. Vision and Pat. Recog. (CVPR)*, 2005, vol. 2, pp. 60–65.
17. N. Azzabou, N. Paragias, and Guichard F., "Image Denoising Based on Adapted Dictionary Computation," in *Proc. of IEEE International Conference on Image Processing (ICIP)*, San Antonio, Texas, USA, Sept. 2007, pp. 109–112.
18. C. Kervrann, J. Boulanger, and P. Coupé, "Bayesian Non-Local Means Filter, Image Redundancy and Adaptive Dictionaries for Noise Removal," in *Proc. Int. Conf. on Scale Space and Variational Methods in Computer Visions (SSVM'07)*, Ischia, Italy, 2007, pp. 520–532.
19. A. Dauwe, B. Goossens, H.Q. Luong, and W. Philips, "A Fast Non-Local Image Denoising Algorithm," in *Proc. SPIE Electronic Imaging*, San José, USA, Jan 2008, vol. 6812.
20. C. Kervrann and J. Boulanger, "Optimal spatial adaptation for patch-based image denoising," *IEEE Trans. Image Processing*, vol. 15, no. 10, pp. 2866–2878, 2006.
21. J. Wang, Y. Guo, Y. Ying, Y. Liu, and Q. Peng, "Fast non-local algorithm for image denoising," in *IEEE Int. Conf. Image Proc. (ICIP)*, 2006, pp. 1429–1432.
22. R. C. Bilcu and M. Vehvilainen, "Fast nonlocal means for image denoising," in *Proc. SPIE Digital Photography III*, Russel A. Martin, Jeffrey M. DiCarlo, and Nitin Sampat, Eds. 2007, vol. 6502, SPIE.
23. J. Aelterman, B. Goossens, A. Pižurica, and W. Philips, *Recent Advances in Signal Processing*, chapter Suppression of Correlated Noise, IN-TECH, 2010.
24. "General-Purpose Computation on Graphics Hardware," http://www.gpgpu.org.
25. A. Kharlamov and V. Podlozhnyuk, "Image denoising," 2007, CUDA 1.1 SDK.
26. F. P. X. De Fontes, G. A. Barroso, and P. Hellier, "Real time ultrasound image denoising," *Journal of Real-Time Image Processing*, 04 2010.
27. B. Goossens, A. Pižurica, and W. Philips, "EM-Based Estimation of Spatially Variant Correlated Image Noise," in *IEEE Int. Conf. Image Proc. (ICIP)*, San Diego, CA, USA, 2008, pp. 1744–1747.