GPU acceleration of NL-means, BM3D and VBM3D

Axel Davy, Thibaud Ehret

Noname manuscript No.
(will be inserted by the editor)

GPU acceleration of NL-means, BM3D and VBM3D

Axel Davy · Thibaud Ehret

the date of receipt and acceptance should be inserted later

Abstract Denoising is an essential part of any image or video processing pipeline. Unfortunately, due to time processing constraints, many pipelines do not consider the use of modern denoisers. These algorithms have only CPU implementations or suboptimal GPU implementations. We propose a new efficient GPU implementation of NL-means and BM3D, and, to our knowledge, the first GPU implementation of the video denoising algorithm VBM3D. The performance of these implementations enable their use in real-time scenarios.

Keywords Image Denoising · Video Denoising · OpenCL · GPU · NL-means · BM3D · VBM3D

1 Introduction

Denoising is a fundamental problem of image and video processing. Its relevance has increased with the democratization of mobile imaging devices such as smartphones. These small devices are often used in poor lighting condition. This degrades the quality of the output image, and the dominant factor in that quality loss is noise. Non-optical imaging devices also suffer from high noise and have seen the development of relevant denoising algorithms, such as MRI images [18], ultrasound images [16], fluorescence microscopy images [8] and SAR images [7,6].

The most common noise model is the additive white Gaussian noise. In the following we denote by u a clean data (either an image or a video) that has been contaminated by an additive Gaussian white noise n of (known) standard deviation σ, meaning that only $v = u + n$ is observed. While simplistic, this white noise model is sufficient to deal with the more realistic Poisson-Gauss model. Indeed, Poisson noise can be brought back to a nearly white Gaussian noise by the application of a variance stabilizing transform [46].

Several of the current state-of-the-art image denoising algorithms are based on an image redundancy assumption stating that for each image patch there exist similar ones in other locations of the image (by patch we mean a small rectangular part of an image). This assumption was first used for denoising purposes by Buades *et al.* in their non-local means (NL-means) [12] denoising algorithm. For every image patch, NL-means computes a weighted average of patches located in a local region. These estimations are then aggregated by averaging all contributions to a pixel to produce the final denoised image. This idea is generalized by Dabov *et al.* in BM3D [19]. For every patch of an image, BM3D searches for similar patches in the image and produces a denoising estimate by transform domain thresholding or Wiener filtering. Other more recent methods such as NL-Bayes [41] or WNNM [34] are other extensions of NL-means achieving state-of-the-art.

More recently deep learning methods have also brilliantly tackled the problem of image denoising. DnCNN [55] and FFDNet [56] achieve currently the best denoising performance. Their main difficulty was a training requiring a large amount of pairs of noisy and clean images. However, Noise2Noise [42] has recently shown how to train with real data without the necessity of disposing of (noisy, clean) pairs. Contrarily to the classic methods presented previously, deep learning methods can, straight out of the box, deal with different types of noise without having to change the architecture, as long as they were trained for each specific noise. A similar principle is used in [28] for a model-blind video denoising method.

Axel Davy · Thibaud Ehret
CMLA, ENS Cachan, CNRS, Université Paris-Saclay 94235 Cachan, France
Axel Davy
E-mail: axel.davy@ens-cachan.fr
Thibaud Ehret
E-mail: thibaud.ehret@ens-cachan.fr

Video denoising is very similar to image denoising. In fact image denoising can be applied frame per frame to produce video denoising. However while the denoising is good when looking at each frame independently, it does not produce a good denoised video. Indeed, temporal consistency in a video is crucial and needs to be imposed in the denoised estimate: When the residual noise of the output video is not temporally consistent, annoying flickering effects appear. Indeed, each frame of the video should be well predicted from the previous frames by following the motion of the objects in the scene. Exploiting this redundancy is crucial to produce the result with the best peak signal-to-noise ratio (PSNR). Another major difference between image and video denoising is the increased computation time efficiency challenge. While computation time efficiency is usually not an issue for image denoising, it can become a major bottleneck for video denoising.

There are two current trends in video denoising. The first one aims at producing the best video denoising possible whatever the computation required, while the second trend focuses on producing the fastest causal video denoising algorithm with the goal of real-time processing.

The methods trying to produce the best denoising are often extensions of image denoising methods. VBM3D [20] and VBM4D [44] extend BM3D. VNLB [4], Global denoising [27] and SPTWO [14] are extensions of NL-Bayes. They all exploit better the self-similarity inside videos, namely the fact that patches have several similar patches around them temporally following the movement. Moreover some of these methods also use patches with a third temporal dimension to better take into account the motion coherence. This helps with the temporal consistency. While these algorithms perform very well, they are often impractical: they use a frame's past and future and therefore can only be used off-line. However they can be adapted to only use the past frames. Though because of their running time they are unfit for high resolution video processing. CNNs have also caught back with the state-of-the-art recently. In particular VNLnet [22,21] extends DnCNN to video.

Fast algorithms rely on much simpler principles which can be implemented efficiently on GPUs or FPGAs. For example [50] relies on a bilateral filter and a Kalman filter to produce a real-time video denoising algorithm. A recursive version of NL-means is proposed in [1]. Ehmann *et al.* [25] combine optical flow and temporal averaging. These methods use simple denoising strategies that generally result in poor denoising quality for high noise levels. A recent trend tries to include mechanisms from state-of-the-art methods into these fast methods to help with the denoising quality; see [29] and [5]. However current implementations of these methods are not fast enough to be real-time.

An interesting middle ground is to produce optimized implementations of the well performing denoising methods for optimized hardware such as GPUs or FPGAs. This has often been done for image denoising with optimized implementations of NL-means [47,23] and BM3D [35,54]. However, even though performance is even more crucial for video, little work has been done to optimize video denoising methods. The only exception being for video versions of NL-means [33].

In this work, we propose real-time GPU implementations of an improved patchwise NL-means and BM3D. Both implementations outperform all previous proposed GPU implementations, without diminishing the quality of the results. We also use the same backbone to get the first optimized GPU implementation of VBM3D. The performance of this implementation is sufficient for real-time video filtering. This is achieved by regrouping all the filtering operations (fetching the patch data, the data transpositions, the 1D filterings and the thresholding) into one kernel without requiring an intermediate buffer for the patches being processed. This significantly reduces the use of memory bandwidth. Moreover, all memory accesses are designed so as to reduce bandwidth and benefit from the cache.

The paper is organized as follows. Section 2 presents the studied methods. Section 3 presents the specificity of GPU implementations. Section 4 presents the efficient GPU implementations for the methods presented in Section 2. The efficiency of the proposed implementations are measured in Section 5. Finally Section 6 concludes.

2 Architecture of the implemented Non-local denoising algorithms

Both NL-means and BM3D (and their video extensions) are based on a self-similarity principle. This means that their architectures are relatively similar; built around a patch search step as well as a patch group processing step. We present the different methods in detail in Sections 2.1, 2.2 and 2.3 as well as the different implementation options.

2.1 NL-means and its extension to video

Original NL-means. NL-means [12] is a popular image and video denoising algorithm, due to its simplicity and speed. It introduced the self-similarity hypothesis: for a patch (a small region of the image) of a natural image, many similar patches can be found in its neighborhood. For noisy images, these similar patches will have different noise realizations and therefore their information can be combined to estimate the restored information. The original NL-means works as follows: for each position (x, y) of an image u, consider the corresponding reference patch $P(x, y)$ centered at this position. The value of the estimated image $\hat{u}$ at position (x, y) is computed as a weighted average of all patches in the small local neighborhood $N(x, y)$ centered on (x, y). While the self-similarity hypothesis says that similar patches should be found, not all patches in the local region are similar. The weights are therefore used to reduce the impact of patches that are too different. The original NL-means suggested the weighted average presented in Equation (1) where the only parameter h is here to take into account the noise.

$$\hat{u}(x, y) = \frac{1}{C} \sum_{x',y' \in N(x,y)} w(x', y')u(x', y'). \tag{1}$$

with

$$w(x', y') = \exp\left(-\frac{\|P(x, y) - P(x', y')\|_2^2}{h^2}\right) \tag{2}$$

and

$$C = \sum_{x',y' \in N(x,y)} w(x', y') \tag{3}$$

Improved NL-means. In this work, an alternative version of NL-means is implemented. The alternative version was preferred since it is more efficient (it requires less computations) while still achieving denoising on par with the original implementation. The improved version follows the same global structure as the NL-means presented in the previous paragraph. The alternative relies on four major differences.

First, we use Equation (4) instead of Equation (2). This improvement proposed in [13] was shown to reduce a bias that could lead to overfitting to noise in [31].

$$w(x', y') = \exp\left(-\frac{\left(\|P(x,y)-P(x',y')\|_2^2 - 2\sigma^2\right)_+}{h^2}\right) \tag{4}$$

We decided to limit the number of patches used in Equation (1). Indeed instead of using all patches from the local search region $N(x, y)$, only use a subset comprised of the n best nearest neighbors is used. This is possible since most patches in the search contribute very little in Equation (1). Very often only a few patches have a distance small enough to contribute. The exception of the flat regions is mentioned later. This choice is principally for computation reasons as it speeds up the method as shown in [45] and [17]. However it can also improve the quality of the results, acting as a regularizer as shown in [32] and [39].

We also added a “flat patch trick” similar to NL-Bayes [41]. For that, a simple test on the variance of the n patches found is added. A flat region is detected when this variance is too close to the variance of the noise (see Equation (5) with $\beta = 1.05$).

$$Var[\{P_i \mid i \in \{1, \ldots, n\}\}] < \beta\sigma^2 \tag{5}$$

In that case we simply average the contribution of all pixels to produce the estimated patch (all pixels of the estimated patch have the same value). This means that $n \times p \times p$ contributions are averaged instead of only n contributions where p is the width of the patch. This is done to improve the quality of the denoising in flat regions, especially when the number of patches used is small.

Table 1 Difference between the original NL-means and the improved NL-means version used for this work.

	Original NL-means	Improved NL-means
Patches used	All patches in the local region	n patches in the local region closest to the reference
Flat patches	Same as other patches	Average all pixels
Weights	$e^{\left(-\frac{\|p-p'\|_2^2}{h^2}\right)}$	$e^{\left(-\frac{\left(\|p-p'\|_2^2-2\sigma^2\right)_+}{h^2}\right)}$
Aggregation	Center pixel of the estimated patch	All pixels of the estimated patch with bilinear weights

The last improvement is the use of a patchwise NL-means as proposed in [13]. In the patchwise NL-means variant, the entire patch is denoised by averaging and the entire estimated patch is aggregated instead of just its center pixel. However, instead of using the uniform weighting of [13] for the aggregation, we use bilinear weighting. This means that pixels close to a patch center will be more impacted by that patch. The advantage of the patchwise NL-means variant is that an aggregation step can be introduced to skip some pixels like in BM3D [19] and NL-Bayes [41] and thus limits the amount of computation. The final estimate of these pixels comes from the aggregation of the overlapping results from the other patches. The improved version of NL-means is described in Algorithm 1 and the differences are summarized in Table 1.

This is not the first work to try to improve NL-means running time or improve the denoising performance. Examples of acceleration are [11] which uses a cluster tree for the patch search and [52] which use FFT and the Summed Squares Image for the filter computation. Examples of denoising performance improvements are [39] which uses a Bayesian model to replace the NL-means weights. Instead of using all the patches in the window, a locally adapted dictionary is extracted locally from the image in a first pass, and from a first denoising result - or oracle - in a second, [24] proposes a local adaptation of the smoothing parameter h using a method based on SURE, [51] improves the output of NL-means, especially on regions with few good neighbors, by minimizing a global cost function involving the NL-means weights, the NL-means output and the TV norm, [37] replaces the NL-means weights with triangular kernels and a scheme to automatically adapt the smoothing parameter. The authors claim that their proposed kernels can be compared to the one proposed in [13] which we use. How-

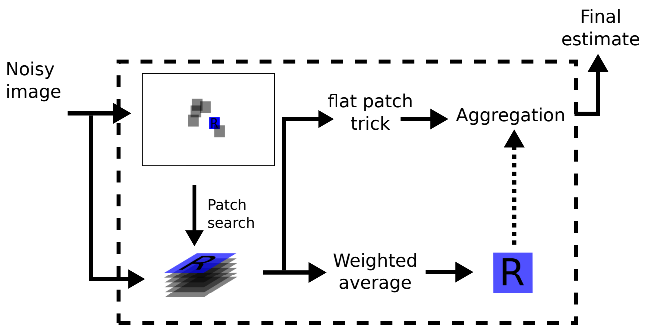

Fig. 1 Scheme of the NL-means algorithm

ever all these works focus on CPU implementations. Indeed their modifications are not adapted for GPU contrary to our improved NL-means that is specifically adapted for GPU.

Algorithm 1: Proposed improved NL-means version

```
Input: Input image $u$, $N$ the number of similar neighbors, $s$ the patch step
1  Create an empty image $v$ (using $u$ size)
2  for each patch $p$ of $u$ on a grid of step $s$ do
3      Search $p_1, \dots, p_N$ the $N$ nearest neighbors of $p$
4      if $p_1, \dots, p_N$ are all flats (test with (4)) then
5          Average all pixels into a flat patch $\hat{p}$
6          Aggregate all pixels of $\hat{p}$ in $v$
7      else
8          Compute the aggregation weights $w_i$ for $p_i$ with (4)
9          $\hat{p} = \sum_{i=1}^{N} w_i p_i$
10         Aggregate all pixels of $\hat{p}$ in $v$
11 return $v$
```

NL-means extension to video. The extension of NL-means to video is simply done by searching patches in a 3D neighborhood as mentioned in [45].

2.2 BM3D by Dabov *et al.*

The denoising principle of BM3D exploits the redundancy of similar patches. Groups of similar 2D patches are assembled in a 3D stack. A separable 3D orthonormal transform is applied to this stack. The stack is denoised by applying a shrinkage operator to the coefficients in a transformed domain. The transform is selected to reveal sparsity in the transformed domain. Most DCT patch coefficients are for example negligible in natural images. However the transform being a patch isometry, the noise remains spread evenly among all DCT coefficients. The algorithm follows four basic steps:

1. Search for similar patches in the image and group them in 3D stacks,
2. Apply a 3D linear domain transform to the 3D blocks,
3. Shrink the transformed coefficients,
4. Apply the inverse transform,
5. Aggregate the resulting patches in the image.

The construction of the estimated image is done by aggregation, *i.e.* by combining the results of all the different estimations. This principle is applied twice, once to compute a basic estimate and a second time, using the basic estimate as a guide, for a final estimation. In the second stage, this process is indeed repeated but uses the denoised patches of the first "basic" step to find similar patches. Furthermore, transform thresholding is also replaced by a Wiener denoising using the denoised patch as oracle. The method is synthesized in Figure 2. In short, we implement exactly the same method as the one presented in [19].

2.3 VBM3D by Dabov *et al.*

VBM3D is a straightforward extension of BM3D to video. The only major difference lies in the patch search. While BM3D looks for similar patches in a square search region centered on the query patch, the patch search of VBM3D takes advantage of the temporal dimension of videos. It also starts by looking for similar patches in a square search region centered on the query patch, but then also searches in neighboring frames in smaller square search regions centered at the same spatial positions as the patches found in the neighboring frames. The method is synthesized in Figure 3. Parameters are also different. Most importantly, the patch size for the Wiener step is different. It was 8×8 for BM3D and is 7×7. Since 7×7 is more complicated to implement we decided to continue using 8×8. We checked, using the implementation from [26], that this choice incurs in no significant performance loss. See Table 2. This is the only difference compared to the method proposed in [20].

3 An introduction to GPU architecture

To understand some of the implementation choices in Section 4, some knowledge about GPU computing workloads is required. We shall thus give an overview of the main particularities and challenges of GPU programming. In this section, we define a coherent hardware terminology.

GPUs and OpenCL for highly parallel workloads. Modern GPUs have a highly superior computing power compared to modern CPUs; they are highly multithreaded. There can be 10^5 to 10^6 threads running at the same time. While parallelization of many data processing algorithms is generally straightforward on CPU since CPUs' cores are fairly independent, GPUs need more care to take advantage of their highly parallelized architecture. In OpenCL, the programmer

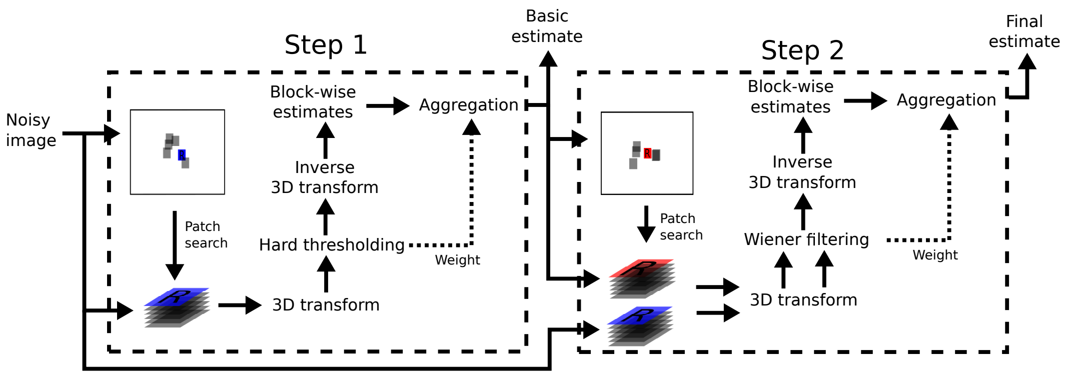

Fig. 2 Scheme of the BM3D algorithm

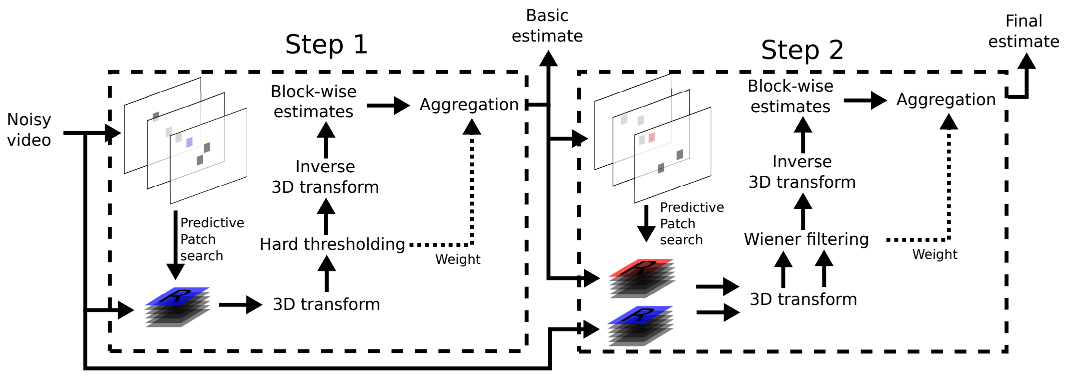

Fig. 3 Scheme of the VBM3D algorithm

Table 2 Comparison of denoising quality using either 7×7 or 8×8 patches for the Wiener estimation of VBM3D. Both have very similar results and therefore justify our usage of 8×8 patches instead of the suggested 7×7 in [26].

σ	Method	Crowd	Park	Pedestrians	Station	Sunflower	Touchdown	Tractor	Average
10	VBM3D (7×7)	35.57	34.66	40.84	38.60	40.18	39.14	37.08	38.01
	VBM3D (8×8)	35.55	34.68	40.88	38.70	40.30	39.16	37.12	38.06
20	VBM3D (7×7)	32.06	31.16	36.88	35.22	36.14	36.13	33.11	34.39
	VBM3D (8×8)	32.06	31.20	36.97	35.33	36.27	36.15	33.19	34.45
40	VBM3D (7×7)	28.40	27.68	32.61	31.84	32.36	33.37	29.45	30.82
	VBM3D (8×8)	28.40	27.69	32.62	31.85	32.36	33.38	29.44	30.82

writes *kernels*, which are functions executing on the GPU. All assigned threads execute the same kernel code but start with a different work-item index. This leads each kernel to do different portions of the work that needs to be parallelized. Alongside kernels, an OpenCL program also needs CPU code to manage the GPU memory and define when and with which arguments these kernels should be executed.

The programmer has some control on how indices are assigned to GPU threads. First, threads are assigned linearly. This enables optimizations so to have consecutive threads access consecutive memory regions. Second, the programmer can define a *workgroup* size (called *local size*). The global size is divided into regions of size defined by the local size. Each region is assigned to a workgroup. The main interest of a workgroup is that its threads have access to a common *local shared memory* only visible to them. This memory enables fast communications between threads. It can be used to store common computations or to cache memory accesses.

A single instruction multiple threads (SIMT) model When assigning indices to threads, the hardware groups consecutive indices of a same workgroup into a *warp*. Each thread inside a warp will execute in *lockstep*. Thus every thread will execute the same instruction at the same time. For example their memory accesses will be simultaneous. This knowledge can be used to reduce the cost of memory accesses. However this model also implies that if several threads need to take different code paths – this phenomenon is called *thread divergence* – all code paths taken need to be executed by the warp, disabling the threads that should not take this code path.

Memory accesses with GPUs Memory accesses are affected by both *latency* and *bandwidth*. Latency is the time taken for an operation to complete. GPUs have two ways of hiding latency. First the code instructions can be organized to have non-dependent computations inserted between a memory access and the code requiring its result. In this case the hardware only needs to wait if the memory access has not finished when the code requiring its results needs to be executed. Second several warps are scheduled on the same computing resources. When a warp has to wait, other warps can be executed.

The bandwidth is the maximum amount of data the hardware can read or write from the memory per second. To maximize the usage of available bandwidth, optimized memory access patterns are required. Any access to read memory at a given location actually loads a batch of consecutive elements (called *cache line*).

Memory accesses are cached to improve latency and bandwidth. Reading the same data again while it is still in cache enables faster access. However, GPU caches are very small relative to the number of threads executing. To compensate for this, GPU threads have access to a significant number of registers, which are local storage only accessible by a given thread.

More details on GPUs can be found in technical documents such as [38,2,49].

4 Implementation details

4.1 Patch search with a large, 2D or 3D, fixed window

In this section we shall describe how to implement an efficient patch search in a predefined 2D (or 3D in the case of video) search window. The problem can be described as such: Given an image subgrid with a pixel spacing of $step$ pixels, we want to compute the n best neighbors for each position (the distance between patches used can be L_1 or L_2 for example). The patch is of size $p \times p$ and the $w \times w$ search window is centered on the reference patch. In the case of a video, a temporal depth is added to the window. Typical values for the parameters are $p = 8$, $w = 21$, n in $\{8, 16, 32\}$ and $step$ in $\{3, 4\}$.

4.1.1 Design choices

Implementing this algorithm raises several challenges. Memory accesses must be optimized to make optimal use of the cache. Indeed some computation is common for neighboring patches when $p > step$ and therefore can be shared. Writing in memory all the computed distances would be heavy and thus only a table of the best n patches must be maintained during patch comparison. Only the positions (and if needed the distances) of the n best patches are written at the end.

For the best efficiency, tables of best patch positions and their weights must be stored in GPU registers. This leads to solutions where one thread will manage all the best neighbors for a given location. This also discourages solutions where a given thread would compute, for a given window offset, the distances between several reference patches and their compared patches. Or solutions where a given thread would handle all distances for a compared patch rather than a reference patch. Thus in the following a given thread will always track distances for a fixed reference position.

When a given thread computes the distances between a reference patch and patches in a window, a lot of data is shared between the computations: the reference patch data is used for all computations and most of the data is in common between a compared patch and its neighbor. The distances must however be recomputed from scratch as different pixels are compared. Because of the restricted register space, storing in registers the equivalent of two $p \times p$ patches per thread may not be possible or be desirable. This encourages spreading distance computations for a given reference patch among several threads. These threads would store a part of the reference patch as well as a part of compared patches and reuse them across computation.

Efficient nearest neighbor table Since writing all patch distances in memory would not only require a lot of memory space, but also bandwidth, we want to only store the final best n neighbors. This implies at any point of the algorithm to remember the best n neighbors seen so far (their positions and their distances), and forget the other positions and distances.

On many hardware, accessing an array with a dynamic index (an index unknown at compilation time) requires storing the array in memory rather than registers. Even if the array ends up staying in cache, the accesses would suffer from latency and bandwidth limitations. Thus to prevent suboptimal performance, the table accesses must always be done with a fixed index. When accessing the array with an index depending on a loop counter, for example, the loop must be unrolled (compiler hints enable that operation). Unrolling the loop means the generated code does not contain the loop jump. Instead the loop content is repeated for each loop index value, thus the array index is known during compilation. Small arrays with only fixed indices accesses can have their content assigned to registers instead of memory.

We propose two alternative algorithms to keep the n best distances and positions: Algorithms 2 and 3. Keeping an ordered table (Algorithm 2) has several advantages over an unordered table (Algorithm 3): The maximum distance is known directly and does not need to be recomputed, and thread divergence is lower. Indeed in Algorithm 3, threads are more likely to insert their elements at different indices than in Algorithm 2.

Algorithm 2: Keeping an ordered table of distances and positions

```
Input: New position $pos$ and distance $dist$, two tables Positions
       and Distances of length $n$ each.
1  if $dist$ < Distances[n-1] then
2      for i from n-1 to 1 do
3          insert ← Distances[i-1] ≤ $dist$
4          Positions[i] ← $pos$ if insert else Positions[i-1]
5          Distances[i] ← $dist$ if insert else Distances[i-1]
6          quit function if insert
7      Positions[0] ← $pos$
8      Distances[0] ← $dist$
```

Algorithm 3: Keeping an unordered table of distances and positions

```
Input: New position $pos$ and distance $dist$, two tables Positions
       and Distances of length $n$ each.
1  max_distance ← max(Distances[0], ..., Distances[$n$-1])
2  if $dist$ < max_distance then
3      for i from 0 to n-1 do
4          if Distances[i] = max_distance then
5              Positions[i] ← $pos$
6              Distances[i] ← $dist$
7              quit function
8          end
9      end
10 end
```

We found Algorithm 2 to be slightly faster indeed in our tests on a NVIDIA GPU. However, in our tests on INTEL and AMD GPUs, the performance was significantly lower due to the compiler reorganizing the iterations in an unoptimized way for this algorithm, while not having issues with Algorithm 3.

4.1.2 Naïve algorithm

In this naïve algorithm, each thread is assigned a different reference patch for which to compute all the distances and maintain the best n neighbors. No particular effort is spent to encourage the compiler to use registers as a cache to reduce the number of memory accesses.

Threads of the same workgroup are assigned to reference patches on the same row. Thus all memory accesses are done on the same image rows, to minimize the number of cache lines per memory accesses in a warp, and more generally improve the cache usage in our scenario. The efficient nearest neighbor table presented above is used.

4.1.3 Convolution algorithm

This algorithm conceptually implements distance computations as separable convolutions while avoiding storing distances. Threads of a workgroup get assigned consecutive positions on a row. For each window offset, each thread will compute the distance between the p pixels of the column at its reference position and at the compared position. This partial result is written in local shared memory. Each thread then reads the results of neighboring threads in order to determine the distance for the whole current $p \times p$ patch. Only threads with positions on the image subgrid maintain the best n neighbors. For $step = 1$, this algorithm is the best performing among the proposed algorithms, due to its very efficient work-sharing and memory pattern. However for higher $step$s, some computation resources are wasted as a lot of threads do not maintain any nearest neighbor table, an operation with non-negligible cost (due to the phenomenon described in Section 3). Note that no threads are assigned to rows which do not intersect with the image subgrid of reference patches.

As p is small enough, the reference pixels can be stored in registers and not reloaded. As for the pixels of the compared patches, on systems with fast local shared memory, memory accesses can be sped up by storing once (and reusing multiple times) the data required for all the comparisons for a given row of the search window.

4.1.4 Square algorithm

This algorithm is optimized for the case $p = 8$ and $step = 4$. In that specific case, the distances between a reference patch and its compared patches can be split into four distances of 4×4 smaller patches, and these sub-distances are exactly shared among four reference patches. Thus this algorithm proposes to divide the image subgrid into 16×16 squares. Each square is assigned to a workgroup (thus workgroups of 256 threads). Each thread tracks the distances for a given reference patch, but only needs to compute the distances for the top 4×4 region of its reference patch. Each thread writes its intermediate result to local shared memory. They can then retrieve the distance for the entire 8×8 patch by reading the results from three other threads. Naturally, threads at the bottom and right borders of the square workgroup cannot compute the full distances, as no thread computed the required remaining distances outside the square. Thus the workgroup computes the neighbors for a 15×15 region of the subgrid. Some overlapping is required to cover the whole image.

This work subdivision reduces efficiently memory accesses (the reference 4×4 patch can be stored in registers, and the memory accessed for the compared patch can be reused for the next patch and kept in registers). Moreover, threads can use memory commands to load four consecutive pixels per call, which reduces the memory instruction count.

4.1.5 Column computations algorithm

This algorithm is an extension of the convolution algorithm, adapting ideas from the square algorithm. It is less efficient than the square algorithm due to p not being as neatly divided by $step$, but is more efficient than the convolution algorithm when $step > 1$. Similarly to the convolution algorithm, a group of threads is assigned on a row to compute distances between a reference column and a compared column each. However, contrarily to the convolution algorithm, each thread computes not one, but $k + 1$ distances. The columns are composed of $p + k * step$ pixels, and the computed distances are the $k + 1$ partial distances for the intersecting reference patches on the subgrid. The partial results are written in local shared memory and some threads (possibly the same ones) access the results to track the distances for a given reference patch. While for the convolution algorithm, the number of reference patches tracked per thread was low (on average $1/step$), in this algorithm k can be controlled to have almost one patch tracked per thread.

Optionally, the partial distance computations and the neighbor tracking can be done on different threads (from separate warps preferably to not waste computational resources), and thus the total register need can be reduced per thread. Indeed if threads do either the partial distance computations or the best distances tracking, the registers used to keep the best weights and distances can be the same as the ones used to store the column reference pixels for example. While for some algorithms and hardware, it is advised to use a low number of registers to have better memory latency reduction, in our case performance was not affected. We believe latency was not an issue for our algorithm.

4.2 VBM3D's patch search

In this section we shall describe how to implement an efficient video patch search on the model of VBM3D's patch search. This patch search is similar to the one of Section 4.1. However this time the search is performed in multiple frames of the video. First, the q most similar patches are found in each frame. Then, the n best among these neighbors are kept for the rest of the processing. The reference frame (frame containing the reference patch) has a search window of size $w_1 \times w_1$ centered on the reference patch while the other frames have search regions that are the union of q search windows of size $w_2 \times w_2$ centered on the best neighbors of the previous frame. Each time the reference patch is of size $p \times p$ and is sampled on a subgrid of pixel spacing of $step$ pixels. Default VBM3D parameters are $p = 8$, $w_1 = 7$, $w_2 = 5$, $q = 2$, $n = 8$, $step = 6$ for the first pass and $step = 4$ for the second.

4.2.1 Design choices

The challenge of this patch search compared to the one of Section 4.1 is that, except for the reference frame, the search windows are shifted by an offset that depends on the patch. While threads treating patches on the same row of the subgrid would access the same rows at the same time for Section 4.1, in this section different rows would be accessed because of the patch-dependent shifts. Moreover a $step$ of 6 means there is quite a gap between the areas accessed by the threads, thus more cache lines would be accessed if using such a partition.

To keep efficient memory access patterns, several threads can be assigned to the same reference patch. Threads can either handle comparisons for different parts of the search windows or the reference patch can be separated into subpatches where each thread is assigned the comparisons of one subpatch. In both cases, the partial results computed from each thread must be communicated to a main thread that will combine them.

4.2.2 Naive algorithm

For this naive algorithm, each reference patch is assigned one thread that computes all distances and determines the n best neighbors with the required temporal constraints (a maximum of q neighbors per frame). Except for the differences in handling the search region, it is essentially similar to the naive algorithm of Section 4.1.

4.2.3 Subdividing the test window

For this algorithm, t threads are assigned per reference patch. For example t can be set to w_1 or w_2. The search windows are divided into columns, and each thread searches for the q nearest neighbors in its assigned column(s). One thread combines the results of the other threads using local shared memory. This work partition optimizes the memory access pattern. Indeed all t threads will access neighboring data.

However, as said in Section 4.1.1, due to the restricted register space, storing the equivalent of two $p \times p$ patches in registers may not be possible or desirable. Thanks to the subdivision in columns, the number of distances computed per thread is small enough so that distances can be computed in several steps by storing partial results in registers. In order to do that, the patches are divided into l blocks (composed of columns of the patch). We compute all the distances for a given block, then the next, until the whole distances have been computed. With this method, the partial patch data is small enough to be kept in registers during the computation of the partial distances for a given block. This optimization reduces the number of memory operations significantly.

4.2.4 Subdividing the tested patches

For this algorithm, p threads are assigned per reference patch. Each thread computes all partial distances for one column of the patch. The partial distances are added to get the actual distance (*reduce-add*). Since p is small enough, each thread

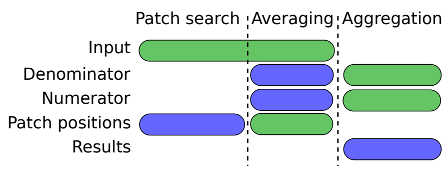

Fig. 4 Memory layout of NL-means. Buffers in green are read-only and buffers in blue are read-write. Note that there is no intermediate buffer for the processing of the patches.

can keep the reference data in registers and thus avoid reloading the full compared data every time a different position is tested. Moreover the memory accesses are optimized as each thread loads consecutive data.

Compared to Section 4.2.3, this algorithm requires much more communication between threads. However, some hardware support specific fast instructions to share data between threads, which can be used to implement an efficient reduce-add operation. For hardware with such support, this variant can be faster than the previous algorithm.

4.3 NL-means' averaging and aggregation

4.3.1 Design choices

Once the best neighbors and their distances are computed, patch distances need to be converted to NL-means weights. We chose to do this computation before saving the results of the patch search, thus saving directly NL-means weights. While it is possible to do both averaging and aggregation at the same time (using a scheme inspired from Section 4.1.4), we did not obtain good performance. Thus we separated the averaging and the aggregation steps. The memory layout of the proposed improved NL-means is presented in Figure 4.

4.3.2 Naive averaging

In the naive algorithm, a thread computes the averages for a single patch. In order to keep register usage low, all computations are divided per row, moving to the next row only when the results of the previous row has been saved for aggregation.

4.3.3 Optimized averaging

GPUs have specific instructions to access several consecutive data elements per threads. However depending on the data type and the patch width p, it is not possible to read or save a whole patch row in a single memory instruction when using a single thread. Keeping that in mind, in this optimized variant several threads are used to reduce even further the number of memory commands required. Instead of assigning one thread per patch, p threads can be assigned per patch. Each thread

Algorithm 4: Proposed NL-means filtering kernel

```
Input: Input image $u$, the output accumulator's numerator $num$
       and denominator $den$, the positions of the nearest
       neighbors $positions$ and the corresponding weights $w$,
       the reference patch position and the column $j$ to process
1  mean ← 0
2  var ← 0
3  for i from 0 to p-1 do
4      for k from 0 to n-1 do
5          data ← u[k-th neighbor row i col j]    // for patches of
             width $p$, $p$ consecutive threads access the same patch
             at consecutive columns
6          mean ← mean + data
7          var ← var + data * data
8  mean ← reduce_add(mean) / (p*p*n)
9  var ← reduce_add(var) / (p*p*n)  // The reduce operation is only
     among the $p$ threads treating a same patch
10 var ← var - mean * mean
11 flatpatch ← var <= $\beta\sigma^2$
12 for i from 0 to p-1 do
13     if flatpatch then
14         denoised_pixel ← mean
15     else
16         denoised_pixel ← 0
17         for k from 0 to n-1 do
18             denoised_pixel ← denoised_pixel + w[k-th
                 neighbor] * u[k-th neighbor row i col j]
19     Accumulate denoised_pixel on $num$ and $den$ with bilinear
         weighting
```

handles the averaging for a column of the patch. Reading or writing a row is done in one memory command, thus guaranteeing optimal memory access patterns. The resulting kernel is summarized on Algorithm 4.

4.3.4 Aggregation using atomics

For NL-means, only the results of the denoised version of the reference patch are aggregated (contrarily to BM3D and VBM3D where all similar patches are denoised and aggregated). Thus, the number of patches covering a given pixel aggregated together is known in advance. In this case, it is not required to keep count of the number of covering patches for each pixel and their weights. Each result can be added with atomics on a single buffer, which will then be normalized correctly for each pixel. Atomics are necessary because several threads from different warps can write at the same positions. However, the normalization code is required to handle explicitly all patch sizes and $step$. Thus our code does not implement this optimization, and simply uses an accumulation buffer for the sum of the weighted denoised patches, and a second accumulation buffer for the sum of the weights. Then a simple shader divides the former by the later.

NL-means weights being floats, the averaging results are floats. Depending on the data dynamic and the required precision, int or long atomics can be used to aggregate data thresholded up a given precision (for example 1.000123 gets written as the integer 10001). Float atomics can also be used, but unfortunately to date most hardware do not support float atomics and hardware which do don't support them in OpenCL. Emulating float atomics can be done with int atomics, but with a significantly higher cost. Indeed, a float add needs to be implemented as several int atomic operations: 'reading the data', 'writing the data plus our result if the data is still set to the one we read previously' (this operation returns the value of the data before the operation). This needs to be repeated if the return value of this last operation is not what we expected (it means other threads updated the value after you accessed it). This atomic float add emulation is presented on Algorithm 5.

Algorithm 5: Emulated atomic float add

```
Input: Write address $p$ and content to add $a$
1 current ← p[0]                                   // Reads initial value
2 repeat
3     expected_current ← current
4     next ← expected_current + a                  // Compute float add
5     current ← atomic_cmpxchg(p, expected_current, next)
        // Try to update
6 until as_uint(current) = as_uint(expected_current)  // Repeat if
    value changed before updating
```

4.3.5 Aggregation without atomics

As said in Section 4.3.4, for fixed p and $step$, only one accumulation buffer is enough for the aggregation since the patch aggregation weights are known in advance. If $step$ is large enough, for example $p = 8$ and $step = 4$, it can be advantageous to not even use an accumulation buffer and thus avoid atomics completely. Instead of having a single buffer containing the aggregated value, we write the different values to aggregate per pixel to several different buffers (four buffers in the example above). These buffers can then be read, summed and normalized all a once to get the final output.

4.4 BM3D and VBM3D's filtering and aggregation

4.4.1 Design choices

Both BM3D's first and second pass as well as VBM3D's first pass use 8×8 patches. While VBM3D uses 7×7 patches for its second pass. We believe implementing VBM3D's second pass using 8×8 patches does not change the algorithm performance (see Section 2.3). For this reason, we will focus on using 8×8 patches only. The number of considered patches per location is always a power of two, 1, 2, 4 or 8 (VBM3D and BM3D), 16 or 32 (BM3D).

Algorithm 6: Proposed BM3D filtering kernel (for the first pass, when the maximum number of neighbors is 8, the 2D transform is the DCT and the 1D transform is the Hadamard transform)

```
Input: Input image $u$, the output accumulator's numerator $num$
       and denominator $den$, the positions of the nearest
       neighbors $positions$, the reference patch position to
       process for this kernel call
1  Retrieve the patch stack from $positions$      // Each of the 64
     threads starts with a table $T$ of 8 elements containing one row
     of the stack
2  Apply the 1D DCT on $T$
3  Apply Algorithm 9 on $T$
4  Apply the 1D DCT on $T$
5  Apply Algorithm 10 on $T$
6  Apply Hadamard on $T$
7  Apply BM3D's hard thresholding on $T$
8  Apply Hadamard on $T$               // Hadamard is its own inverse
9  Apply the inverse of Algorithm 10 on $T$
10 Apply the 1D DCT on $T$        // The normalized DCT is its own
     inverse
11 Apply Algorithm 9 on $T$        // Algorithm 9 is its own inverse
12 Apply the 1D DCT on $T$
13 Accumulate the results on $num$ and $den$ with BM3D's
     weighting scheme
```

One way of solving the filtering is first to read the nearest neighbors for each position, form the associated 3D stack of patches, and write it to memory. The filtering can then be implemented as successive passes on the buffer containing all the stacks. However, the bandwidth needed to write and process the 3D stacks would lead to a suboptimal algorithm. Instead we propose to use the high number of registers available on a GPU to save bandwidth. All considered hardware have workgroups of 64 threads and at least 128 registers available per thread. Thus it is possible to store the $8 \times 8 \times 32$ elements of a 3D stack of patches in the registers of the threads of a given workgroup.

Our proposal is to assign a workgroup of 64 threads for each position to process. The 3D filtering can be implemented as three separate 1D filtering operations (one for each dimension), followed by a shrinking operation (thresholding or Wiener) and finally the three separate inverse 1D filtering operations. At all times, one thread only has direct access to a subset of the 3D stack, such as a few rows. To perform the 3D filtering, each thread performs a 1D filtering operation, and then exchanges its data with the other threads so to have a new dimension of the data (for example columns instead of rows). We call that operation *transposition*. A transposition, for example, moves from a state where each thread contains one (or a few) patch column(s) to a state where each thread contains one (or a few) patch row(s). Implementing efficient 1D filtering or shrinking operations is not hard, thus in the following we focus on the transposition operations. The overall pseudo-code of BM3D's filtering kernel is presented in Algorithm 6 for the specific case of the first pass with a max-

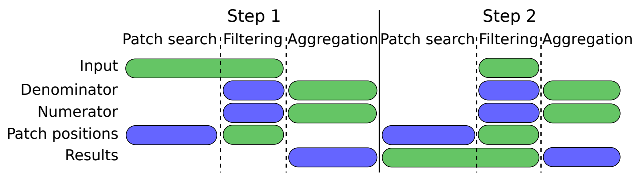

Fig. 5 Memory layout of BM3D and VBM3D. Buffers in green are read-only and buffers in blue are read-write. Note that there is no intermediate buffer for the processing of the patch stack.

imum number of 8 neighbors. For 16 or 32 neighbors, each thread reads not one, but two or four rows, respectively, and the operations are repeated for each. The memory layout of the proposed design choices is presented in Figure 5.

4.4.2 Efficient transpose operations

We describe here the case of eight patches (of width $p = 8$) and will then extend to the other number of patches.

Initially, each of the 64 threads contains a column of one of the patches. Thus each thread contains eight elements and can apply one of the 1D transforms.

swapping columns and rows The first step is to swap the data so that each thread gets one row of one of the patches instead of a column. To proceed with the swap, each thread writes the eight pixel values it has in local shared memory, which is a fast memory reserved for the 64 threads. Then each thread can read from the local shared memory its target eight pixel values. Doing that efficiently is not as simple as it sounds. Extra care must be taken when reading and writing data so to achieve good performance.

Let us first formalize the transpose operation. It starts with values stored in a table T of size $[8][8][8]$ representing the 3D patch stack. T is stored in the threads, each thread containing the section $[z_i][.][y_i]$ (corresponding to the column y_i of patch z_i) where i is the thread index (from 0 to 63), and z_i, y_i thread coordinates, with $y_i = i \bmod 8$ and $z_i = i/8$. The goal is to have the data reorganized as in a second table T'' such as $T''[z_i][y_i][.] = T[z_i][.][y_i]$. In practice the memory of T is reused for T''.

The most “naive” way of performing the swap is to simply write T in local shared memory in an arbitrary order, and then read the indices in the right order. Examples of such "naive" algorithms are described in Algorithms 7 and 8. Both algorithms are equivalent.

We shall analyze more in depth Algorithm 8. In this naive version, the data is written into an intermediate 3D table T' of size 512 in local shared memory. The table T' is organized as a buffer of size $[8][8][8]$. Each thread (z_i, x_i) writes its data so as $T'[y_i][z_i][.] = T[z_i][.][y_i]$. It just remains to read T' in the right order to obtain T'', $T''[z_i][y_i][.] = T'[.][z_i][y_i]$. Elements are written and read one by one in each thread (loop on k in Algorithm 8), thus resulting into

Algorithm 7: Swapping columns and rows (Naive - variant 1)

```
Input: $z_i$, $y_i$ thread indices, $T$ thread-specific table of length 8,
       $T'$ table in local shared memory
1 for $k$ from 0 to 7 do
2     $T'[64k + 8z_i + y_i] \leftarrow T[k]$
3 end
4 barrier()              // Wait for all threads to finish writing
5 for $k$ from 0 to 7 do
6     $T[k] \leftarrow T'[64y_i + 8z_i + k]$
7 end
```

Algorithm 8: Swapping columns and rows (Naive - variant 2)

```
Input: $z_i$, $y_i$ thread indices, $T$ thread-specific table of length 8,
       $T'$ table in local shared memory
1 for $k$ from 0 to 7 do
2     $T'[64y_i + 8z_i + k] \leftarrow T[k]$
3 end
4 barrier()              // Wait for all threads to finish writing
5 for $k$ from 0 to 7 do
6     $T[k] \leftarrow T'[64k + 8z_i + y_i]$
7 end
```

eight write instructions and eight read instructions for each thread (ignoring the instruction merging mentioned later). While at read time, the threads load 64 consecutive elements and thus guaranteeing no bank conflicts, at writing time the proposed pattern triggers bank conflicts Indeed threads 0, 4, 8, etc, write in their first call at cases 0, 32, 64, which map to the same memory bank (all current hardware have either 16 or 32 banks). The same problem happens with the other similar groups of threads. The bank conflicts cause the accesses to the same banks to be serialized, thus being executed as several separate calls. The write performance will thus be suboptimal. The solution to improve performance is to 'shift' the locations of the data written by the threads. The corrected algorithm is identical except T' has padding. It is of size $[8][64 + x]$, but is interpreted as a table of size $[8][8][8]$ as before, but with an offset of x times the index of the first dimension.

This performance trick is well documented and is described for example in [49], which suggests $x = 1$ in the case of a matrix transpose. However, it does not give the best performance on all hardware: Indeed optimal performance is achieved by maximizing the use of banks per calls. However, some hardware can write or read several consecutive elements per thread in one call. The tested INTEL hardware, which can have a warp size of eight, can support reading or writing four elements per thread in one call. Reading or writing only one element results in suboptimal performance as the banks are underutilized. The tested AMD hardware, which has a warp size of 64, executes local shared memory commands in four parts, first the first 16 threads, etc, until the last 16 threads, and ideally each of these subcalls should use all the banks (they have 32). Like for the INTEL GPU, the tested AMD GPU can write or read several consecutive elements per thread in one call (two per call). The tested NVIDIA hardware has warps of 32 threads, has 32 banks, and does not process them in two subcalls. Note that we assume that a bank covers elements of four bytes, which is the size of a float (The NVIDIA hardware can configure it to eight bytes).

Based on this variety of behaviors, it is difficult to design read or write patterns that are optimal on all hardware as it depends on the warp size, the number of banks and how they are accessed. However, assuming the number of banks is a divisor of 64 - our workgroup size -, we noticed that having each thread write (or read) the maximum number of consecutive elements it can write (or read) in one instruction is always an optimal pattern. The same can be noticed when having threads write (or read) from consecutive addresses modulo 64 (thus to consecutive banks). We will use these access patterns in our algorithms for best performance and compatibility (we expect these patterns will still be optimal for future generations).

Algorithm 9: Swapping columns and rows

```
Input: $z_i, y_i$ thread indices, $T$ table of 8 registers for the thread,
       $T'$ table in local shared memory
1 for k from 0 to 7 do
2     $T'[(64+x)y_i + 8z_i + k] \leftarrow T[k]$
3 end
4 barrier()                    // Wait for all threads to finish writing
5 for k from 0 to 7 do
6     $T[k] \leftarrow T'[(64+x)k + 8z_i + y_i]$
7 end
```

This leads us to Algorithm 9. The best x corresponds to the maximum number of consecutive elements the hardware can read or write per thread per access. Thus, for the tested hardware, $x = 4$ should be best for INTEL, $x = 2$ for AMD, and $x = 1$ for NVIDIA. We verified experimentally that indeed these parameters proved to be the best performing for each hardware. We also checked the generated assembly code to confirm that the compiler combines the writes of consecutive data into write packets of x elements. We expect the best value of x to change for future hardware generations and should be set accordingly. When memory organization and the memory command packing abilities are not known for a specific hardware, the best x can be determined empirically.

The proposed writing order covers all the banks per groups of threads in a warp and uses all the banks. It is therefore the best possible pattern. For example, on the INTEL hardware with $x = 4$, the first write of thread 0 writes to banks 0, 1, 2, 3, thread 1 writes to banks 4, 5, 6, 7, etc. This corresponds indeed to the optimal pattern we suggested.

Swapping rows and channels Thanks to the transpose presented previously, the first two 1D transforms can be applied on both the rows and the columns. The last 1D transform must be applied on the third dimension of the 3D patch stack. This means that each thread must contain the third dimension of the 3D stack for a given patch pixel. The only constraint for this transposition is for the position $(0, 0)$ of each transformed patch. The rest of the elements does not require any specific ordering as long as the inverse transposition puts back the data in the correct thread. In our case we will require that the position $(0, 0)$ of each transformed patch is put in the thread 0. Furthermore we will assume x is either 1, 2, 4 or 8.

To reformulate the problem, we start with data as a table T of size $[8][8][8]$ stored in the threads. Each thread starts with the section $[z_i][y_i][.]$ where i is the thread index, and z_i, y_i thread coordinates. We want the data to be stored as $[.][\mu_i][\nu_i]$, with μ_i, ν_i thread coordinates. We will use local shared memory as intermediate table T' of size $[8][8/x + 64][8][x]$, and assign

$$T'[z_i][j/x + z_i * 8][y_i][j \bmod x] = T[z_i][y_i][j].$$

This writing pattern verifies the conditions set in the previous paragraph. It avoids bank conflicts and enables full use of the ability of the hardware to write x elements in one call. At reading time, we assign $T[j][z_i][y_i] = T'[j][a_i + j*8][b_i][c_i]$, where $a_i * 8 * x + b_i * x + c_i = z_i * 8 + y_i$. As among threads, y_i increases first, then z_i , the pattern also prevents bank conflicts (it verifies again the conditions set previously). Contrary to when writing T', the read commands cannot be combined to read more than one value per call.

One problem with this proposed algorithm is the size of T' which can be huge: It is of length $512(1 + x)$. However T' can be fit in a much smaller table of size $[8][8/x][8][x]$, that is of length 512, while maintaining an optimal access pattern. To do so, modulus will be used in the writing and read address computations.

Algorithm 10: Swapping rows and channels

```
Input: $z_i, y_i$ thread indices, $T$ table of 8 registers for the thread,
       $T'$ table in local shared memory
1  for h from 0 to (8/x - 1) do
2      for j from 0 to x-1  // The x memory operations are merged
3      do
4          $T'[64z_i + (8z_ix + 8hx + y_ix + j) \bmod 64] \leftarrow$
             $T[hx + j]$
5      end
6  end
7  barrier()                    // Wait for all threads to finish writing
8  for k from 0 to 7 do
9      $T[k] \leftarrow T'[64k + (8kx + 8z_i + y_i) \bmod 64]$
10 end
```

Algorithm 10 presents the entire algorithm (with T' a 1D table of length 512). It is easier to visualize on Algorithm 10 that the algorithm is correct and that it indeed gives the pattern we described in the previous paragraph: Since $(8z_i + y_i)$ indexes threads from 0 to 63, the loops of Algorithm 10 access indeed consecutive addresses modulo 64 for a fixed h and a fixed k. Thus optimal performance is guaranteed.

Generalizing to 1, 2, 4, 16 *or* 32 *patches* The swapping operations presented previously assumed we have to process eight patches. In order to generalize to more patches, we simply need to store several columns of data initially per thread, and call the transposition functions presented previously several times. When having fewer patches, some operations could be avoided. However, we simply called the function for eight patches while setting the missing patches to 0 and therefore just ignoring them in the 1D transforms.

4.4.3 On the use of atomics for the aggregation

Due to the non-uniform weighting, the weighted pixel values being aggregated have a wide range and, contrarily to NL-means, many elements are aggregated per pixel for BM3D and VBM3D. We found that even with 8-bit RGB images, 32 bits integers could be insufficient to store the weighted sum in some corner cases. Long atomics (64-bits integers) are recommended. When not supported, the emulation of float atomics discussed in the Section 4.3.4 should be used. Integer atomics could be used if reducing the range of the weights used for the aggregation, though the PSNR could be affected.

4.5 Conclusion

In this section, we have explained in detail different optimized implementations (as well as their naive versions) of the pieces of code required for NL-means, BM3D and VBM3D. In our implementation, which we compare to other implementations in Section 5, we have selected the best optimized codes among the ones described, depending on the parameters. For the patch search of NL-means and BM3D, we use the fast variant described in Section 4.1.4 when possible ($p = 8$, $step = 4$), else use the variant of Section 4.1.5, except for $step = 1$, in which case the variant of Section 4.1.3 is fastest. The naive variant is only used for border handling when needed. For NL-means, we use the optimized averaging, rather than the naive one. We do aggregation using atomics for all methods. Last, VBM3D's patch search uses Section 4.2.4 for INTEL GPUs, else the algorithm of Section 4.2.3.

Table 3 Compared configurations: The first accelerator is a 16-core CPU, while the other accelerators are GPUs with different levels of performance. The bandwidth and number of operations per seconds were obtained with Clpeak, which estimates the actual peak figures which can be obtained with OpenCL.

Acc.	Description	Bandwidth	Float ops
0	Intel Xeon W-2145 CPU (18.1.0.0920)	63 GB/s	1.8 T/s
1	INTEL i7-6600U GPU (NEO 18.21.10858)	23 GB/s	380 G/s
2	AMD Radeon RX 480 (2766.4)	209 GB/s	5.7 T/s
3	NVIDIA TITAN V (390.129)	611 GB/s	13.7 T/s

Table 4 Comparison of NL-means' denoising performance (average PSNR) on several datasets with the compared implementations (noise standard deviation of level 20).

Method	BSD68	Kodak	IPOL
[13]	28.81	29.93	31.89
OpenCV	28.27	29.33	31.53
OpenCV GPU	28.24	29.31	31.55
Ours P5,S1	28.92	30.13	32.60
Ours	28.51	29.76	32.12

5 Benchmarks

In this section we will compare the performance, both in terms of running time and PSNR, of our implementations with several reference implementations. To remove the effects of differences in color handling, all the experiments in this paper are grayscale. The denoising performance will be compared on several datasets: BSD68 a set of 68 images from the Berkeley segmentation dataset [48], Kodak a set of 24 images from the Kodak dataset [30] and IPOL a set of 16 images from [15][1]. In addition, we compare the denoising performance and running time for two images: Lena and CMLA (from [10]). The former is a 512×512 image, while the latter is 4608×3456. Table 3 shows the different OpenCL devices, called accelerators (acc.) later on, considered. We consider a high-end multicore CPU (0), an integrated GPU (1), and middle-range consumer grade GPU (2) and a high-end GPU (3).

NL-means In this section, we compare our GPU implementation to the reference CPU code of [13] and to OpenCV (version 4.1.0 [9]) with both a CPU and a GPU implementation. We compare the denoising performance with additive white Gaussian noise of standard deviation $\sigma = 20$. For this noise level, [13] recommends 5×5 patches and $h = 0.4\sigma = 8$. However, likely due to differences in the implementation choices, $h = 8$ did not give good results for OpenCV. By maximizing the PSNR on a serie of images of the Waterloo dataset [43], we found the best parameter for OpenCV to be

[1] Available on `http://mcolom.info/download/no_noise_images/no_noise_images.zip`

Table 5 Comparison of NL-means' denoising performance (PSNR) and execution time with the compared implementations. CPU methods were run on a single core of an Intel(R) Xeon(R) W-2145 CPU @ 3.70GHz. GPU methods were run on accelerator 3. The running times and the 95% confidence intervals were estimated from 100 runs. Running time for GPU methods corresponds to the sum of time spent running GPU kernels, obtained with nvprof or OpenCL profiling information. Running time for CPU methods corresponds to execution time after the image is loaded and before the result is saved.

Method	Lena (PSNR-Time)	CMLA (PSNR-Time)
[13]	31.59 - 8.5 $\pm$0.2 s	31.75 - 522 $\pm$5 s
OpenCV	31.20 - 279 $\pm$2 ms	31.12 - 16.0 $\pm$0.3 s
OpenCV GPU	31.18 - 6 $\pm$1 ms	Out of memory
Ours P5,S1	32.05 - 4.27 $\pm$0.02 ms	32.32 - 219.2 $\pm$0.5 ms
Ours	31.86 - 0.699 $\pm$0.007 ms	31.99 - 20.7 $\pm$0.1 ms

Table 6 Comparison of the time taken on the compared accelerators for the main passes (NL-means with our default parameters) on the CMLA. The running times and the 95% confidence intervals were estimated from 100 runs.

Accelerator	search	filtering	Total
0	1672 $\pm$6 ms	282 $\pm$4 ms	1950 $\pm$7 ms
2	40 $\pm$1 ms	24.0 $\pm$0.3 ms	64.5 $\pm$0.5 ms
3	15.91 $\pm$0.08 ms	4.44 $\pm$0.02 ms	20.7 $\pm$0.1 ms

$h = 20$ for this noise level with 5×5 patches. Due to the differences in our implementation (number of neighbors, patch size, flat patch trick, aggregation), a different parameter h was needed as well. We used $h = \sigma$. We compare our implementation for two patch sizes. By default, our implementation uses 8×8 patches with a subgrid step of four. Thus a pixel is covered by four patches. We also show results for patches of size 5×5 and a subgrid step of one, which corresponds to the parameters of the reference implementation.

We illustrate the denoising performance of the implementations on standard datasets in Table 4. As can be seen, OpenCV performs about the same with the CPU and GPU backends, but underperforms compared to the reference implementation. Our code with the default parameters (*Ours*) was slightly outperformed by the reference implementation on BSD68 and Kodak, but performed better on the IPOL dataset. When analyzing the results, one can see that our method has better denoising performance on images with flat regions thanks to the flat patch trick. Due to the use of larger patches than recommended for this noise level, the performance is slightly lower elsewhere. When using our code with 5×5 patches and a subgrid step of 1 (*Ours P5, S1*), our method outperforms the reference implementation on all datasets tested. The gains over [13] can be explained by the use of a bilinear scheme for the combination of the results of overlapping patches, instead of using a constant weight, and by the flat patch trick.

In Table 5, we compare both the denoising performance (PSNR) and the execution time on Lena and CMLA. One can observe that OpenCV's CPU implementation is more than 30 times faster than the reference code, although at the cost of a lower PSNR. Our GPU code runs significantly faster with the default parameters than with 5×5 patches and a subgrid step of 1, but the denoising quality is slightly lower. Both compared versions of our code ran faster than OpenCV's GPU implementation on Lena, while obtaining better denoising results. OpenCV could not denoise CMLA on the GPU as it required intermediate buffers that would not fit in the 12GB GPU memory. One can see that the execution time of our implementation does not scale linearly with the image resolution. The difference in running time per pixel between both images can be explained by the effects of cache (Lena fits completely in the GPU cache) and better use of the parallelization capabilities of a GPU in the case of CMLA which is bigger. The running time of our implementation on three different accelerator is analyzed in Table 6. Most of the running time is spent on the patch search, rather than the NL-means weighting (filtering). The aggregation of the results (normalization of an aggregation buffer) is small and therefore not displayed. It is nonetheless included in the total running time. Accelerator 1 could not compute the result on CMLA as the execution time was more than the timeout threshold of this GPU. We can notice our code executing on CPU (accelerator 0) is competitive with the compared CPU implementations (the equivalent single core time is 31s, compared to *OpenCV*'s 16s and [13]'s 522s), even though our code was not optimized for CPU specifically. This is likely due to an efficient use of the CPU extended instruction set (SSE, AVX, etc) for the code generated by OpenCL.

BM3D There are many different implementations of BM3D available. We compare our code to two GPU implementations, [35,36] and [53], and two CPU implementations, [40] and OpenCV's. [40] is a reference CPU re-implementation of the original paper and *OpenCV* is the implementation that can be found in OpenCV. We only consider open-source implementations reproducing the original method described in [19].

In Table 7, we show the default parameters of these methods as well as which parameters can be modified. No other methods use the default parameter proposed in the original BM3D publication. Only our method can be configured to have exactly these parameters. It has to be noted, though, that the Hadamard and Haar transforms have been shown to give equivalent results in [19], thus arguably, [40]'s parameters are almost identical to the original, except for the default search region size.

Due to the different sets of 3D filters supported by each implementation, they are not exactly comparable. Thus, we only compare using the same parameters, [40], [35] and our implementation. The patch search region is set to 39×39, the 2D transforms are both set to DCT, and the 1D transforms

Table 7 Default parameters for $\sigma = 20$ for all *BM3D* methods compared. Parameters in gray can be modified easily by changing a value in the code, or passing an argument when calling the method.

Method	patch size	search region size	2D transform	1D transform	patch step	Number of nearest neighbors	Maximum distance for a nearest neighbor
Original [19]	8×8 / 8×8	39×39 / 39×39	Bior/DCT	Haar/Haar	3/3	16/32	2500/400
[40]	8×8 / 8×8	33×33 / 33×33	Bior/DCT	Hadamard/Hadamard	3/3	16/32	2500/400
OpenCV	4×4 / 4×4	16×16 / 16×16	Haar/Haar	Haar/Haar	1/1	8/8	2500/400
[35]	8×8 / 8×8	23×23 / 23×23	DCT/DCT	Hadamard/Hadamard	3/3	16/32	3000/400
[53]	8×8 / 8×8	32×32 / 32×32	DCT/DCT	DCT/DCT	4/4	8/8	3000/400
Ours	8×8 / 8×8	21×21 / 21×21	Bior/DCT	Hadamard/Hadamard	4/4	8/8	2500/400

Table 8 Comparison of denoising performance (average PSNR) of BM3D on several datasets of the compared implementations (noise standard deviation of level 20). *: The default parameters were replaced by the fixed parameters used to compare the three implementations.

Method	BSD68	Kodak	IPOL
[40]*	29.32	30.68	33.06
[35]*	29.26	30.65	33.05
Ours*	29.27	30.66	33.06
[40]	29.50	30.83	33.23
OpenCV	29.04	30.24	31.82
[35]	29.26	30.61	32.94
[53]	27.34	29.60	31.94
Ours	29.34	30.65	32.91

Table 9 Comparison of denoising performance (PSNR) and execution time between for the compared BM3D implementations. CPU methods were run on a single core of a Intel(R) Xeon(R) W-2145 CPU @ 3.70GHz. GPU methods were run on accelerator 3. The running times and the 95% confidence intervals were estimated from 100 runs. Running time for GPU methods corresponds to the sum of time spent running GPU kernels, obtained with nvprof or OpenCL profiling information. Running time for CPU methods corresponds to execution time after the image is loaded and before the result is saved. *: The default parameters were replaced by the fixed parameters used to compare the three implementations.

Method	Lena (PSNR-Time)	CMLA (PSNR-Time)
[40]*	32.98 - 7.68 ±0.06 s	33.39 - 491 ±5 s
[35]*	32.97 - 22.24 ±0.07 ms	33.40 - 986.5 ±0.8 ms
Ours*	32.98 - 16.15 ±0.05 ms	33.40 - 513 ±4 ms
[40]	33.04 - 6.84 ±0.08 s	33.38 - 425 ±3 s
OpenCV	31.87 - 3.36 ±0.07 s	31.71 - 204.6 ±0.5 s
[35]	32.94 - 17.97 ±0.04 ms	33.07 - 760.7 ±0.7 ms
[53]	32.11 - 52 ±4 ms	32.67 - 5.06 ±0.01 s
Ours	32.71 - 2.59 ±0.02 ms	32.89 - 91.9 ±0.5 ms

Table 10 Comparison of the time taken (in ms) on the compared accelerators for the main passes (BM3D with our default parameters) on CMLA. The running times and the 95% confidence intervals were estimated from 100 runs. *: Using long atomics instead of float atomics

Acc.	search	filtering	search	filtering	Total
0	446 ±5	2106 ±9	5230 ±10	2150 ±10	9930 ±30
1	1090 ±10	962 ±4	1081 ±5	713 ±4	3860 ±10
2	17.6 ±0.2	87.2 ±0.2	30.2 ±0.2	105.0 ±0.2	241.9 ±0.2
2*	17.6 ±0.3	35.3 ±0.4	20.2 ±0.3	38.3 ±0.3	124.6 ±0.5
3	6.47 ±0.04	33.34 ±0.07	11.75 ±0.05	39.61 ±0.07	91.8 ±0.2

to Hadamard. The patch step is 3, and the maximum number of nearest neighbors is 16 for the first pass, and 32 for the second pass.

We also compare the performance of all implementations with their default parameters. Similarly as before, we show the denoising performance on standard datasets in Table 8 and denoising performance and running time for two images in Table 9. Details about the running time of our method can be found in Table 10. As we can see, *OpenCV* and [53] underperform compared to other methods. This is likely due to their choice of 2D or 1D transforms.

The three compared methods set to the same fixed parameters had similar denoising performance, which validates the equivalence of these implementations, besides the default parameters. However our implementation was faster than the other GPU implementation [35]. The main optimization compared to [35] is the use of a single kernel to implement all the filtering operations (see Section 4.4.1). When comparing the performance of all the implementations with their default parameters, [40] had the best denoised results, and our implementation was the fastest. Our default parameters were chosen to maximize denoising performance while looking for aggressive speedups. As a result, it ran almost nine times faster than the closest competitor. However as our parameters can easily be changed, our code can also be used to reproduce the results of the best performing [40]. This is done at a cost similar to the one reported in Table 9 as the main difference between the parameters suggested in [19] and the fixed one used for the table is the use of Bior instead of DCT for the first pass.

Table 10, which analyzes the running time of our method on several accelerators, shows that, contrary to NL-means, most of the running time is spent in the filtering of the patch stacks, rather than in the search of neighbors. One limiting factor of this filtering is the writing of the results on an accumulation buffer with emulated float atomics. As said in Section 4.4.3, hardware long atomics can be used if the range of the input is known. We can see in Table 10 that accelerator 2 benefits significantly from hardware long atomics since the total running time is almost divided by two. The hardware of accelerator 3 is able to do hardware float atomics and long

Table 11 Comparison of denoising performance (average PSNR) and average running time per frame of VBM3D and NL-means on the Derf dataset of the compared implementations (noise standard deviation of level 20). Whether the method uses a single frame (SF) or a GPU is indicated.

Method	SF	GPU	PSNR	Time/frame
NLM ours	yes	yes	31.54	0.83 ±0.06 ms
BM3D ours	yes	yes	32.93	4.7 ±0.3 ms
NLM video ours	no	yes	33.53	10.0 ±0.3 ms
VBM3D [26]	no	no	34.21	3.0 ±0.2 s
VBM3D ours	no	yes	34.31	3.4 ±0.2 ms

Table 12 Comparison of the average time per frame taken (in ms) on the compared accelerators for the main passes for our VBM3D implementation (with our default params) on the Derf dataset.

Acc.	search	filtering	search	filtering	Total
0	22.5 ±0.7	31.5 ±0.6	48 ±1	72 ±1	174 ±4
1	28 ±1	20.5 ±0.8	65 ±2	50 ±2	164 ±8
2	1.7 ±0.1	1.13 ±0.04	3.3 ±0.2	2.6 ±0.3	9.0 ±0.5
3	0.54 ±0.07	0.55 ±0.05	0.90 ±0.02	1.1 ±0.1	3.4 ±0.2

atomics (available in CUDA), but these functionalities are not available for the OpenCL driver at the time of writing. Both implementations [35] and [53], written in CUDA, used hardware float atomics. We notice that, like it was for NL-means, our BM3D code running on accelerator 0 (CPU) is competitive with the compared CPU implementations. It runs at an equivalent single-core execution time of 159s, compared to 205s for *OpenCV* and 425s for [40]. One has to be careful though when comparing these results because the parameters are different.

VBM3D and video NL-means In this section, we delve into the performance of the video variations of the previous algorithms.

While NL-means has been previously used for video denoising, there is no standard video version. Thus, for this comparison, we use our code extending the search to a 3D window including the 4 past images and the 4 future images of the current sequence (similarly to VBM3D). In addition, we use 16×16 patches. The use of bigger patches is justified for video denoising as it results in better matches of the same content in the previous frames, which improves the denoising result and the temporal consistency.

We compared with VBM3D using the original parameters of [20] as implemented by [26], and our implementation (which has a different patch size for the second pass, as explained in 2.3). To highlight the improvements of the video algorithms over the image versions, we show the denoising performance of NL-means and BM3D with our default parameters, used separately on each frame (no 3D search window).

Table 11 shows the denoising performance and average running time per frame on the Derf dataset of the compared implementations. The Derf dataset corresponds to the Derf's Test Media collection[2] as processed in [3]: The original videos are RGB of size 1920×1080, and seven grayscale sequences of 100 frames were extracted and down-sampled by a factor two (the resolution is thus 960×540).

Video methods obtain a significantly superior PSNR (a 1.99 db gain for NL-means and 1.28db for BM3D). While NL-means is slower in its video version than its image version, due to the bigger patch search region, VBM3D is faster than our BM3D with default parameters. This is due to the competitive video patch search algorithm of VBM3D, and due to the first pass of the algorithm using a patch step of 6 for VBM3D, while our BM3D uses 4. The fact that our GPU implementation has a slightly better denoising performance (of 0.10 db) compared to the reference CPU implementation is due to the use of 8×8 patches in the second pass. This phenomenon was already observed in Table 2, on the same dataset (with a different noise generation). For VBM3D, Table 12 shows that computation time is quite balanced between all steps even though the search takes slightly more time than the filtering. It's also important to notice that the implementation achieves real-time denoising on these sequences for both accelerators 2 and 3.

6 Conclusion

In this paper, we proposed a GPU implementation of NL-means, BM3D and VBM3D. The technical challenges of this endeavour and of the proposed solutions have been discussed. The PSNR denoising performance was shown to match, and sometimes even to surpass, the denoising performance of the reference CPU implementations while being several orders of magnitude faster. We also proposed compromises for the default parameters to get an even larger speedup at a minor cost for the denoising performance. This work paves the way to new uses of these algorithms in time constrained environments.

Acknowledgments

The authors gratefully thanks Jean-Michel Morel for his valuable feedbacks. Work partly financed by IDEX Paris-Saclay IDI 2016, ANR-11-IDEX-0003-02, Office of Naval research grant N00014-17-1-2552, DGA Astrid project «filmer la Terre» n^{o} ANR-17-ASTR-0013-01, MENRT and Fondation Mathématique Jacques Hadamard.

[2] `https://media.xiph.org/video/derf`

References

1. Ali, R.A., Hardie, R.C.: Recursive non-local means filter for video denoising. EURASIP JIVP (1), 29 (2017)
2. AMD: AMD APP SDK OpenCL^{TM} Optimization Guide (2015)
3. Arias, P., Facciolo, G., Morel, J.M.: A comparison of patch-based models in video denoising. In: IEEE IVMSP, pp. 1–5 (2018)
4. Arias, P., Morel, J.M.: Video denoising via empirical bayesian estimation of space-time patches. JMIV **60**(1), 70–93 (2018)
5. Arias, P., Morel, J.M.: Kalman filtering of patches for frame-recursive video denoising. In: IEEE CVPRW (2019)
6. Aubert, G., Aujol, J.F.: A variational approach to removing multiplicative noise. SIAM SIIMS **68**(4), 925–946 (2008)
7. Aujol, J.F., Aubert, G., Blanc-Féraud, L., Chambolle, A.: Image decomposition application to sar images. In: Springer Scale-Space, pp. 297–312 (2003)
8. Boulanger, J., Kervrann, C., Bouthemy, P., Elbau, P., Sibarita, J.B., Salamero, J.: Patch-based nonlocal functional for denoising fluorescence microscopy image sequences. IEEE TMI **29**(2), 442–454 (2009)
9. Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000)
10. Briand, T., Davy, A.: Optimization of Image B-spline Interpolation for GPU Architectures. IPOL **9**, 183–204 (2019)
11. Brox, T., Kleinschmidt, O., Cremers, D.: Efficient nonlocal means for denoising of textural patterns. IEEE TIP **17**(7), 1083–1092 (2008)
12. Buades, A., Coll, B., Morel, J.M.: A non-local algorithm for image denoising. In: IEEE CVPR, vol. 2, pp. 60–65 (2005)
13. Buades, A., Coll, B., Morel, J.M.: Non-local means denoising. IPOL **1**, 208–212 (2011)
14. Buades, A., Lisani, J.L., Miladinović, M.: Patch-based video denoising with optical flow estimation. IEEE TIP **25**(6), 2573–2586 (2016)
15. Colom, M.: Multiscale noise estimation and removal for digital images. Ph.D. thesis, Universitat de les Illes Balears (2014)
16. Coupé, P., Hellier, P., Kervrann, C., Barillot, C.: Nonlocal means-based speckle filtering for ultrasound images. IEEE TIP **18**(10), 2221–2229 (2009)
17. Coupé, P., Yger, P., Barillot, C.: Fast non local means denoising for 3d mr images. In: International Conference on Medical Image Computing and Computer-Assisted Intervention, pp. 33–40. Springer (2006)
18. Coupé, P., Yger, P., Prima, S., Hellier, P., Kervrann, C., Barillot, C.: An optimized blockwise nonlocal means denoising filter for 3-d magnetic resonance images. IEEE TMI **27**(4), 425–441 (2008)
19. Dabov, K., Foi, A., Egiazarian, K.: Image denoising by sparse 3-d transform-domain collaborative filtering. IEEE TIP **16**(8), 2080–2095 (2007)
20. Dabov, K., Foi, A., Egiazarian, K.: Video denoising by sparse 3d transform-domain collaborative filtering. In: 2007 15th European Signal Processing Conference, pp. 145–149. IEEE (2007)
21. Davy, A., Ehret, T., Facciolo, G., Morel, J., Arias, P.: Non-local video denoising by CNN. CoRR **abs/1811.12758** (2018)
22. Davy, A., Ehret, T., Facciolo, G., Morel, J., Arias, P.: A non-local cnn for video denoising. In: IEEE ICIP, pp. 2409–2413 (2019)
23. De Fontes, F.P.X., Barroso, G.A., Coupé, P., Hellier, P.: Real time ultrasound image denoising. Journal of real-time image processing **6**(1), 15–22 (2011)
24. Duval, V., Aujol, J.F., Gousseau, Y.: On the parameter choice for the non-local means (2010)
25. Ehmann, J., Chu, L.C., Tsai, S.F., Liang, C.K.: Real-time video denoising on mobile phones. In: IEEE ICIP, pp. 505–509 (2018)
26. Ehret, T., Arias, P.: Implementation of the vbm3d video denoising method and some variants (2020)
27. Ehret, T., Arias, P., Morel, J.M.: Global patch search boosts video denoising. In: VISAPP, vol. 5, pp. 124–134 (2017)
28. Ehret, T., Davy, A., Morel, J.M., Facciolo, G., Arias, P.: Model-blind video denoising via frame-to-frame training. In: IEEE CVPR, pp. 11369–11378 (2019)
29. Ehret, T., Morel, J.M., Arias, P.: Non-local kalman: A recursive video denoising algorithm. In: IEEE ICIP, pp. 3204–3208 (2018)
30. Franzen, R.: Kodak lossless true color image suite. source: `http://r0k.us/graphics/kodak` **4** (1999)
31. Frosio, I., Kautz, J.: Statistical nearest neighbors for image denoising. IEEE TIP **28**(2), 723–738 (2018)
32. Gilboa, G., Osher, S.: Nonlocal linear image regularization and supervised segmentation. Multiscale Modeling & Simulation **6**(2), 595–630 (2007)
33. Goossens, B., Luong, H., Aelterman, J., Pižurica, A., Philips, W.: A gpu-accelerated real-time nlmeans algorithm for denoising color video sequences. In: ACIVS, pp. 46–57. Springer (2010)
34. Gu, S., Zhang, L., Zuo, W., Feng, X.: Weighted nuclear norm minimization with application to image denoising. In: IEEE CVPR, pp. 2862–2869 (2014)
35. Honzátko, D., Kruliš, M.: Accelerating block-matching and 3d filtering method for image denoising on gpus. Journal of Real-Time Image Processing **16**(6), 2273–2287 (2019)
36. Honzátko, D., Kruliš, M.: Cuda implementation of bm3d. `https://github.com/DawyD/bm3d-gpu` (2018)
37. Jin, Q., Grama, I., Kervrann, C., Liu, Q.: Nonlocal means and optimal weights for noise removal. SIAM SIIMS **10**(4), 1878–1920 (2017)
38. Junkins, S.: The compute architecture of intel® processor graphics gen9 (2015)
39. Kervrann, C., Boulanger, J., Coupé, P.: Bayesian non-local means filter, image redundancy and adaptive dictionaries for noise removal. In: International Conference on Scale Space and Variational Methods in Computer Vision, pp. 520–532. Springer (2007)
40. Lebrun, M.: An Analysis and Implementation of the BM3D Image Denoising Method. IPOL **2**, 175–213 (2012)
41. Lebrun, M., Buades, A., Morel, J.M.: A nonlocal bayesian image denoising algorithm. SIAM SIIMS **6**(3), 1665–1688 (2013)
42. Lehtinen, J., Munkberg, J., Hasselgren, J., Laine, S., Karras, T., Aittala, M., Aila, T.: Noise2noise: Learning image restoration without clean data. In: International Conference on Machine Learning, pp. 2971–2980 (2018)
43. Ma, K., Duanmu, Z., Wu, Q., Wang, Z., Yong, H., Li, H., Zhang, L.: Waterloo Exploration Database: New challenges for image quality assessment models. IEEE TIP **26**(2), 1004–1016 (2017)
44. Maggioni, M., Boracchi, G., Foi, A., Egiazarian, K.: Video denoising, deblocking, and enhancement through separable 4-D nonlocal spatiotemporal transforms. IEEE TIP **21**(9), 3952–3966 (2012)
45. Mahmoudi, M., Sapiro, G.: Fast image and video denoising via nonlocal means of similar neighborhoods. IEEE SPL **12**(12), 839–842 (2005)
46. Makitalo, M., Foi, A.: Optimal inversion of the generalized anscombe transformation for poisson-gaussian noise. IEEE TIP **22**(1), 91–103 (2012)
47. Márques, A., Pardo, A.: Implementation of non local means filter in gpus. In: Iberoamerican Congress on Pattern Recognition, pp. 407–414. Springer (2013)
48. Martin, D., Fowlkes, C., Tal, D., Malik, J.: A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In: Proc. 8th Int'l Conf. Computer Vision, vol. 2, pp. 416–423 (2001)
49. NVIDIA: NVIDIA OpenCL Best Practices Guide (2009)
50. Pfleger, S.G., Plentz, P.D.M., Rocha, R.C.O., Pereira, A.D., Castro, M.: Real-time video denoising on multicores and gpus with kalman-based and bilateral filters fusion. Journal of Real-Time Image Processing **16**(5), 1629–1642 (2017)
51. Sutour, C., Deledalle, C.A., Aujol, J.F.: Adaptive regularization of the nl-means: Application to image and video denoising. IEEE TIP **23**(8), 3506–3521 (2014)

52. Wang, J., Guo, Y., Ying, Y., Liu, Y., Peng, Q.: Fast non-local algorithm for image denoising. In: IEEE ICIP, pp. 1429–1432 (2006)
53. Wang, T., Sun, Y.: Gpu-accelerated denoising with bm3d. `https://github.com/JeffOwOSun/gpu-bm3d` (2017)
54. Wang, X., Xu, K., Wang, D.: Accelerating block-matching and 3d filtering-based image denoising algorithm on fpgas. In: IEEE ICSP, pp. 235–240. IEEE (2018)
55. Zhang, K., Zuo, W., Chen, Y., Meng, D., Zhang, L.: Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising. IEEE TIP **26**(7), 3142–3155 (2017)
56. Zhang, K., Zuo, W., Zhang, L.: Ffdnet: Toward a fast and flexible solution for cnn-based image denoising. IEEE TIP **27**(9), 4608–4622 (2018)