Journal of Real-Time Image Processing
https://doi.org/10.1007/s11554-017-0737-9

ORIGINAL RESEARCH PAPER

Accelerating block-matching and 3D filtering method for image denoising on GPUs

David Honzátko[1] · Martin Kruliš[1]

Received: 10 April 2017 / Accepted: 19 November 2017

Abstract
Denoising photographs and video recordings is an important task in the domain of image processing. In this paper, we focus on block-matching and 3D filtering (BM3D) algorithm, which uses self-similarity of image blocks to improve the noise-filtering process. Even though this method has achieved quite impressive results in the terms of denoising quality, it is not being widely used. One of the reasons is a fact that the method is extremely computationally demanding. In this paper, we present a CUDA-accelerated implementation which increased the image processing speed significantly and brings the BM3D method much closer to real applications. The GPU implementation of the BM3D algorithm is not as straightforward as the implementation of simpler image processing methods, and we believe that some parts (especially the block-matching) can be utilized separately or provide guidelines for similar algorithms.

Keywords Image denoising · Block matching · Filtering · GPU · CUDA

1 Introduction

Images acquired by optical sensors such as CCD or CMOS chips have always been devalued slightly by noise. The noise is a result of the imperfection inherent to every empirical measurement. Even though it cannot be completely eliminated, both hardware and software methods for noise reduction were devised.

Our focus lies on the random noise caused by thermal variances. It is particularly noticeable when a camera increases the sensitivity (ISO) of the chip to compensate for insufficient illumination of the scene for instance. Human observer can perceive this noise in an image as random speckles on an otherwise smooth surface, and it can be approximated by *additive white Gaussian noise* (AWGN), which is an independent random noise that follows normal distribution.

There are various software denoising methods. The presented block-matching and 3D filtering (BM3D) is one of the most advanced algorithms for image denoising [4]. According to all papers cited in Sect. 2, the BM3D is still considered as a state-of-the-art denoising method despite the fact it has been published in 2006. It effectively combines two successful approaches—non-local filtering and transform-based filtering. Unfortunately, the algorithm is very computationally demanding, so it has not been widely employed in regular applications, not to mention in real-time processing.

In this work, we present a novel implementation of the algorithm which uses CUDA framework to harness raw computational power of modern GPUs. Despite the fact that the GPUs were originally designed for graphical tasks and that they have been used to accelerate various image processing methods, the parallelization of BM3D algorithm is neither trivial nor straightforward. The block-matching part is very data intensive, and thus it requires a solution that utilizes caches and shared memory of the GPU to their full potential. Furthermore, direct implementation would contain many redundant computations; thus, a more sophisticated algorithm can improve the processing times further.

✉ Martin Kruliš
krulis@ksi.mff.cuni.cz

David Honzátko
honzatko@ksi.mff.cuni.cz

1 Parallel Architectures/Algorithms/Applications Research Group, Faculty of Mathematics and Physics, Charles University in Prague, Malostranské nám. 25, Prague, Czech Republic

The paper is organized as follows. Section 2 summarizes related work, and Sect. 3 reviews the fundamentals of current GPU architectures. The BM3D algorithm is presented in Sect. 4, and the details of our parallel implementation are in Sect. 5. Empirical evaluation and comparison of our solution to CPU baseline and existing OpenCL implementation are provided in Sect. 6. Section 7 concludes the paper.

2 Related work

Image denoising is quite a common task at present. Since it is by definition data-parallel, various denoising algorithms have been accelerated using GPUs. For instance, a smoothing kernel where each pixel is computed as weighted average of its neighbors is almost a model example of a problem suited for GPU architectures.

Among the more advanced denoising methods, those based on non-local similarities as *Non-Local Means* (NL-means) [1] are perhaps the most attractive and the most often studied from the perspective of GPU acceleration. Due to their high denoising effectivity and related high computational costs, several algorithms were developed [8, 11, 20]. At present, algorithms based on GPU-accelerated NL-means are being widely used and they are available in libraries such as OpenCV.[1] Although some of these algorithms provide good denoising performance in general, neither of them offers better denoising qualities than BM3D.

Recent research on image denoising brought up several sophisticated methods that outperform BM3D in terms of denoising quality.

Some of those methods are based on non-local self-similarities just as BM3D, but they employ rather different denoising filters. Among those, we can list at least learned simultaneous sparse coding (LSSC) [10] and weighted nuclear norm minimization (WNNM) [7]. To our best knowledge, there is no GPU implementation of these methods so far, but there is a possibility that parts of our GPU implementation of BM3D could be also reused for acceleration of these algorithms.

Most of the recently developed methods rely on models that were pre-trained on some large dataset of natural images. We can list at least: LSSC [10], expected patch log likelihood (EPLL) [21], optimized markov random fields (opt-MRF) [2], trained reaction diffusion (TRD) [3], and cascade of shrinkage fields (CSF) [19]. However, due to the recent breakthroughs in the field of neural networks, further research and possible innovations of these denoising methods are expected. The last three of the mentioned methods have GPU-accelerated implementations, and it has the potential to outperform BM3D not only in the denoising quality, but also in the processing time. However, only CPU implementation of these methods was compared so far; thus, there is a possibility that our implementation can be better in the terms of processing time.

Except for CSF [19] and TRD [3], the presented methods were not evaluated on larger images that are common in the customer segment. Our implementation of BM3D primarily targets this type of images.

There was already one attempt to implement BM3D on GPU [18]; however, presented solution used rather simple parallelization that may be outperformed by algorithm proposed in this paper. In fact, our experimental data strongly support this assumption (Sect. 6.2).

3 GPU fundamentals

In this section, we review the GPU architecture fundamentals with particular emphasis on aspects which have great importance in light of the studied problem. We will focus mainly on the NVIDIA Kepler [12], Maxwell [13], and Pascal [14] since these architectures were used in our experiments. However, most stated facts are generally valid for all GPU architectures.

3.1 GPU device

A GPU card is a peripheral device connected to the host system via the PCI-Express (PCIe) bus. The GPU is quite independent, since it has its own memory and processing unit, so the host may offload computations to the GPU while the CPU performs other tasks. On the other hand, the GPU cannot access the host memory directly, so all the data must be transferred to/from the GPU internal memory.

The GPU processor consists of several *streaming multiprocessors* (SMPs), which can be roughly related to the CPU cores. The SMPs share the main memory bus and the L2 cache, but otherwise they are almost independent. Each SMP consists mainly of multiple *GPU cores* (e.g., 192 on Kepler or 128 on Maxwell), instruction decoder and scheduler, L1 cache, and shared memory. The GPU cores are tightly coupled and they execute the same code simultaneously, but each core has its own arithmetic units and registers.

3.2 Thread execution

The GPUs employ a parallel paradigm called *data parallelism*, in which the concurrency is achieved by processing multiple data items concurrently by the same routine (called *kernel* in the case of GPUs). When a kernel is

[1] http://opencv.org/.

invoked, the caller specifies how many threads are spawned for this kernel. Each thread executes the kernel code, but it has an unique thread ID which identifies its portion of data (work).

The threads are grouped together into blocks of the same size, and blocks are scheduled on available SMPs. Threads within one block may synchronize their work using barriers and closely cooperate using internal resources of the SMP (especially the shared memory). Furthermore, threads in a block are divided into subgroups called *warps*. The number of threads in a warp is fixed for each architecture (32 in all current architectures). Threads in a warp are executed in a *lock-step*, which means they are all issued the same instruction at a time. When a warp is forced to wait (e.g., when transferring data from the memory), SMP simply schedules instructions of another warp to keep GPU cores occupied.

The lock-step execution suffers from branching problems. When threads in a warp diverge in their code execution—e.g., in an '`if`' or '`while`' statement—all code branches must be executed by the whole warp. Thread masks instruction execution according to its local conditions to ensure correct results, so heavily branched code could lead to underutilization of the GPU cores.

3.3 Memory organization

The GPU memory hierarchy (Fig. 1) is more complex than in case of CPU.

The *host memory* is the operational memory of the computer. It is directly accessible by the CPU, but it cannot be accessed directly by any peripheral coprocessors such as the GPU. Input data need to be transferred from the host memory (RAM) to the graphic device global memory (VRAM), and the results need to be transferred back when the kernel execution finishes.

The *global memory* can be accessed from the GPU cores, so the input data and the results computed by a kernel are stored here. The global memory bus has both high latency and high bandwidth. Data are transferred in larger aligned blocks, so the threads are encouraged to access the memory in coalesced manner.

The *shared memory* is shared among threads within one thread block. It is rather small (tens of kB), but almost as fast as the GPU registers. The shared memory can play the role of a program-managed cache for the global memory, or it can be used to exchange intermediate results by the threads in the block. The memory is divided into several banks (usually 16 or 32), so that subsequent 4-byte words are stored in subsequent banks (modulo the number of banks). When multiple threads access the same bank (except if they read the same address), the memory operations

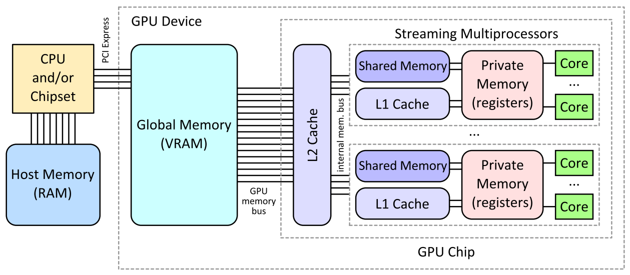


Fig. 1 Host and GPU memory organization scheme

are serialized which create undesirable delay for all threads in the warp due to the lock-step execution.

The *private memory* belongs exclusively to a single thread and corresponds to the GPU core registers. Private memory size is very limited (tens to hundreds of words); therefore, it is suitable just for a few local variables.

Finally, there are two levels of cache present. The L2 cache is shared by all SMPs, and it transparently caches all access to global memory. The L1 cache is private to each SMP, and it caches data from global memory selectively (using specialized loading instruction, which can be either written explicitly by programmer or generated by compiler).

In summary, we can infer the following guidelines for our parallel code.

- Inputs, outputs, and intermediate data of the BM3D algorithm must fit the global memory. Even though the image data of photographs are typically measured in megabytes, the intermediate results of block-matching algorithm may be very large.
- The workload has to be designed to promote coalesced data access pattern.
- Shared memory should be utilized wisely, and the data should be organized so that the banking conflicts are avoided.

These issues have to be reflected in our algorithm design, and the outlined limitations have to be observed.

3.4 Thread cooperation and data exchange

One of the critical aspects of complex GPU algorithms is the thread information exchange (i.e., how closely the threads cooperate). Any data exchange requires some level of implicit or explicit synchronization, and thus it may limit the performance of any parallel algorithm. These limitations may be crucial, and thus the division of the workload among the threads can affect the overall performance significantly.

In our case, the main concern is the thread cooperation and data exchange within a thread block (i.e., among the threads that share the SMP). The threads within one block can cooperate via shared memory. This memory can be read and written by any thread, and the synchronization may be achieved either implicitly by mutual exclusion and barriers, or explicitly by atomic instructions. Even though this type of data exchange is quite efficient, the shared memory read-after-write latency is approximately 24 cycles [15], so we need to plan these data exchanges accordingly. Furthermore, allocating larger amounts of shared memory limits the ability of the scheduler to assign multiple thread blocks to one SMP, which may lead to underutilization of the CUDA cores.

Threads within one warp may communicate more tightly without the use of shared memory. For this purpose, NVIDIA GPUs implement specialized warp instructions. For instance, a `ballot` voting instruction which is used to broadcast one bit of information from each thread to all threads in the warp. All NVIDIA architectures since Kepler offer more sophisticated shuffling functions, where each thread sends a 32-bit value and receives another value from another thread according to a given shuffling pattern.

These methods of cooperation have to be employed with extra care in order to achieve optimal performance, especially in case of data-intensive algorithms such as the block-matching step of BM3D denoising.

4 Block-matching and 3D filtering algorithm

The BM3D algorithm is based on the idea that images (photographs) have much more sparse representation than noise in transform domain, and thus the noise can be easily attenuated.

The BM3D algorithm benefits from the fact that pictures capturing real-world scenes often exhibit repetitive patterns such as geometric shapes or textures. The algorithm identifies these similar patches and uses them to improve sliding window transform-domain denoising.

We have implemented and described only the algorithm for noise of standard deviation $\sigma < 40$, which is more common in camera-produced photographs. With higher values of σ a significant drop of denoising performance can be expected. However, this paper focuses on the efficiency aspects of the method only and the execution time is not directly affected by the σ value. We assume only grayscale images where each pixel is represented by an integer value from $\langle 0, 255 \rangle$, but extending our method to denoise color images as well [5] is quite straightforward.

For better orientation, we will use a notation which is as close as possible to the notation introduced in the analysis of Lebrun [9]. All differences are duly emphasized.

BM3D algorithm consists of two main *phases*.[2] The first phase produces a *basic estimate* of the denoised image. This phase is based on hard thresholding, and it exhibits relatively good denoising performance on its own, so it can be used even separately. The second phase uses both the basic estimate and the noisy image to produce the final denoised image estimate. It is based on empirical Wiener filtering, and it improves the denoising performance of the first phase further.

4.1 The first (basic estimate) phase

The first phase of the algorithm can be divided into three essential steps:

1. *Grouping* employs the block-matching concept to identify similar blocks—square areas of fixed size. Since the term 'block' has also other meanings (e.g., the CUDA thread block), we will denote these square areas *patches*. Similar patches are grouped together into 3D groups (1D stacks of 2D patches).
2. *Collaborative Filtering* represents the main denoising part of this phase. It performs 3D decorrelating transformation of the stacks produced by previous step, hard thresholding, and corresponding inverse 3D transformation.
3. *Aggregation* combines the results of the filtered overlapping patches into the final denoised image by computing weighted average for each pixel.

4.1.1 Grouping (block-matching)

The first step performs the block-matching (BM) procedure to identify similar parts of the input image. The algorithm operates on *patches* of $k^{\mathbf{hard}} \times k^{\mathbf{hard}}$ pixels, where $k^{\mathbf{hard}}$ is a configuration parameter.[3] The block-matching part takes every patch from the image as a *reference patch* R and searches its neighborhood of $n^{\mathbf{hard}} \times n^{\mathbf{hard}}$ pixels[4] for similar patches P. An example of block-matching results for a selected reference patch is depicted in Fig. 2.

The similarity of the patches is defined as an inversion of their distance d. In our implementation, we have used normalized Euclidean (L_2) metric as a distance function:

$$d(P, Q) = \frac{L_2^2(P, Q)}{(k^{\mathbf{hard}})^2}$$

Two patches are considered similar if their distance is lower than threshold $\tau^{\mathbf{hard}}$. Each group of reference patch and its similar patches are stored in a 3D array $\mathcal{P}(R) =$

[2] The two main phases were originally denoted *steps* [9]. We have decided to change the original terminology to avoid ambiguity of the term 'step' as it would become rather overused in the detailed description.

[3] The patch size is typically relatively small. We use $k^{\mathbf{hard}} = 8$ in our experiments.

[4] We use $n^{\mathbf{hard}} = 39$ in our experiments.

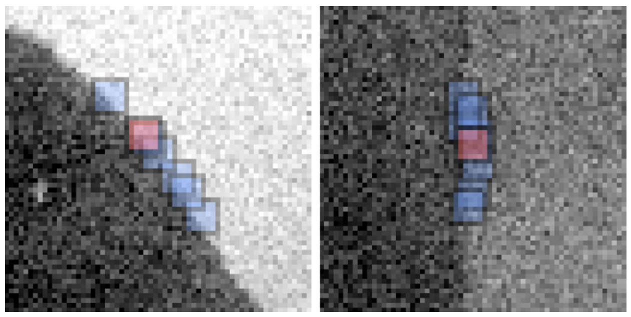


Fig. 2 Example of block-matching results

$\{P : d(R,P) \leq \tau^{\mathbf{hard}}\}$ and passed on to the subsequent steps of the algorithm.[5] For performance reasons, the number of patches is limited to a fixed constant $|\mathcal{P}(R)| \leq N^{\mathbf{hard}}$. In our experiments, $N^{\mathbf{hard}} = 16$ and the patches with the lowest distances are selected into the group if the limit is reached. Furthermore, the exact number of patches in the stack is rounded to the nearest lower power of 2, since it is required by our implementation of collaborative filtering.

The BM algorithm is very computationally demanding, since there are $(W - k^{\mathbf{hard}} + 1) \times (H - k^{\mathbf{hard}} + 1)$ reference patches (W and H being the width and the height of the image, respectively) and each reference patch requires $\Theta((n^{\mathbf{hard}})^2 (k^{\mathbf{hard}})^2 \log N^{\mathbf{hard}})$ time to find similar patches. One of the possible optimization techniques is to select reference patches more sparsely. We introduce parameter $p^{\mathbf{hard}}$ $(1 \leq p^{\mathbf{hard}} \leq k^{\mathbf{hard}})$, which defines the pixel distance between adjacent reference patches (both horizontally and vertically).[6] To ensure the coverage of the entire image, patches adjacent to the border are taken as reference patches regardless of the parameter $p^{\mathbf{hard}}$.

4.1.2 Collaborative 3D filtering

The filtering step employs transform-based denoising, which consists of three straightforward sub-steps:

1. Transformation of the filtered sample using decorrelating unitary function $\tau_{3D}^{\mathbf{hard}}$ (e.g., Fourier, Cosine, Bior, or Walsh–Hadamard transform).[7]
2. Filtering the coefficients using hard thresholding.
3. Inverse transformation $\tau_{3D}^{\mathbf{hard}^{-1}}$ to the transformation used in the first sub-step.

Traditional 2D filtering performs the transformation on the image (or its portions) directly. Collaborative filtering employs 3D transformation on a stack of patches produced by the grouping step. Due to the similarity of the patches, the results of 3D transformation have even sparser representation than the individual 2D transformations, while the noise remains uniformly distributed (with constant absolute value equal to its standard deviation σ). Hence, more details of the original image are preserved after thresholding.

The procedure can be expressed by the following formula:

$$\mathbb{P}^{\mathbf{hard}}(R) = \tau_{3D}^{\mathbf{hard}^{-1}}\left(\gamma\left(\tau_{3D}^{\mathbf{hard}}(\mathcal{P}(R))\right)\right)$$

where $\gamma(x)$ is a hard thresholding operator which returns zero if $|x| \leq \lambda_{3D}^{\mathbf{hard}}\sigma$, otherwise it returns x. The constant $\lambda_{3D}^{\mathbf{hard}}$ is a crucial configuration parameter.[8]

4.1.3 Aggregation

The aggregation step combines the filtering results to produce a basic estimate of the denoised image. A patch can be assigned to multiple 3D groups by the block-matching process, and the patches may overlap. Hence, each pixel value is determined as weighted average from the values of all contributing patches. The coverage of the whole image is ensured by the fact that each pixel belongs to at least one reference patch and a reference patch is always member of its own 3D group.

Homogeneous patches are prioritized over patches containing edges and corners in order to avoid their distortion. Homogeneous patches have much sparser representation; thus, more of their coefficients are filtered by the hard thresholding. Therefore, we use the number of retained (nonzero) coefficients to measure homogeneity.

The weight $w_P^{\mathbf{hard}}$ of a patch P is determined as an inverse value of the number of retained coefficients. Let us denote $N_R^{\mathbf{hard}}$ the number of nonzero coefficients of the 3D group $\mathbb{P}^{\mathbf{hard}}(R)$. The weights of all patches $P \in \mathbb{P}^{\mathbf{hard}}(R)$ are:

$$w_P^{\mathbf{hard}} = \begin{cases} 1/N_R^{\mathbf{hard}} & \text{if} \quad \mathrm{N}_\mathrm{R}^{\mathbf{hard}} > 0 \\ 1 & \text{if} \quad \mathrm{N}_\mathrm{R}^{\mathbf{hard}} = 0 \end{cases}$$

To simplify the aggregation equation, we additionally define indicator function $X(P, x)$ which has value 1 when pixel x belongs to patch P and 0 otherwise. Furthermore, we denote $u_{P,R}(x)$ the value of pixel x from the patch P produced by the denoised stack $\mathbb{P}^{\mathbf{hard}}(R)$. The value

[5] We use $\tau^{\mathbf{hard}} = 2500$ in our experiments.

[6] We use $p^{\mathbf{hard}} = 3$ in our experiments.

[7] τ_{3D} usually comprises a 2D transform applied to each patch and 1D transform applied to the 3rd dimension of the 3D group (across the patches), while different transforms can be combined. 2D Cosine transform and 1D Walsh–Hadamard transform are used in our work.

[8] We use $\lambda_{3D}^{\mathbf{hard}} = 2.7$ in our experiments.

Table 1 Overview of the optimal parameter values for both phases

Parameter	Hard	Wien
k (patch size)	8	8
n (neighborhood size)	39	39
N (max. patches in group)	16	32
p (patch dist. in pixels)	3	3
τ (distance threshold)	2500	400
λ_{3D} (hard thresholding limit)	2.7	–

$u^{\mathbf{basic}}(x)$ of pixel x of the basic image estimate is subsequently computed as

$$u^{\mathbf{basic}}(x) = \frac{\sum_R w_R \sum_{P \in \mathbb{P}(R)} X(P,x)\, u_{P,R}(x)}{\sum_R w_R \sum_{P \in \mathbb{P}(R)} X(P,x)}$$

where the $\sum_R$ is a sum over all reference patches R of the whole image.

4.2 The second (Wiener) phase

The second phase of the algorithm uses empirical Wiener filtering of the noisy image with the basic estimate from the first phase as an oracle. It is empirically proven that this phase improves the denoising performance.

This phase follows the same outline as the first phase; only the individual steps are slightly modified. All parameters used in this phase will be indexed $^{\mathbf{wien}}$ to distinguish them from the parameters of the first phase. Optimal values of these parameters might be slightly different in each phase. Table 1 presents the overview of these values, and we will also point out the differences in this section.

4.2.1 Grouping

To improve the matching accuracy, the block-matching is computed using the basic estimate rather than the original noisy image.

In this step, two 3D arrays for each reference patch are constructed:

- $\mathbb{P}^{\mathbf{basic}}(R) = \{P : d(R,P) \leq \tau^{\mathbf{wien}}\}$ where R is a reference patch in the basic image estimate and P denotes an arbitrary patch in the neighborhood of R ($\tau^{\mathbf{wien}}$ is 400 in our experiments)
- $\mathbb{P}(R)$ that contains patches of the noisy image located at the same positions as the patches in $\mathbb{P}^{basic}(R)$.

Analogically to the first phase, only $N^{\mathbf{wien}}$ the most similar patches are kept in the stack due to performance reasons.[9]

[9] We use $N^{\mathbf{wien}} = 32$ in our experiments.

4.2.2 Collaborative filtering

The nonlinear hard thresholding filter is replaced by linear Wiener filter in this step. Instead of setting small coefficients to zero, Wiener filtering attenuates each noisy coefficient in the transform domain to achieve the lowest mean square error (MSE) between denoised and original signal.

For an arbitrary signal, coefficients of the Wiener filter are computed from the estimates of the original signal power $|S(x)|^2$ and noise power $|N(x)|^2$.

$$\omega(x) = \frac{|S(x)|^2}{|S(x)|^2 + |N(x)|^2}$$

In our case, the power of the original signal is estimated from 3D groups of the basic image estimate and the noise power is the expected noise variance σ^2. Thus, the coefficients of the Wiener filter are computed as follows[10]:

$$\omega_R(x) = \frac{\left|\tau_{3D}^{\mathbf{wien}}(\mathbb{P}^{\mathbf{basic}}(R))(x)\right|^2}{\left|\tau_{3D}^{\mathbf{wien}}(\mathbb{P}^{\mathbf{basic}}(R))(x)\right|^2 + \sigma^2}$$

Filtering itself is performed as element-wise multiplication of transformed noisy 3D groups $\tau_{3D}^{\mathbf{wien}}(\mathbb{P}(R))$. with the filter coefficients ω_R.

$$\mathbb{P}^{\mathbf{wien}}(R) = \tau_{3D}^{\mathbf{wien}^{-1}}\left(\omega_R \cdot \tau_{3D}^{\mathbf{wien}}(\mathbb{P}(R))\right).$$

4.2.3 Aggregation

Given the filtered 3D arrays $\mathbb{P}^{\mathbf{wien}}(P)$, the aggregation step produces the final estimate of the denoised image. As in the first phase, aggregation computes weighted average of all overlapping pixel estimates where the weight prefers pixel estimates belonging to the more homogeneous patches. Since $\tau_{3D}^{\mathbf{wien}}$ is a decorrelating transform, inverse squared L_2 norm of the filter coefficients has proven to be adequate weight for each stack:

$$w_P^{\mathbf{wien}} = ||\omega_P||_2^{-2}$$

The pixel value of the final image estimate $u^{\mathbf{final}}(x)$ is therefore computed as:

$$u^{\mathbf{final}}(x) = \frac{\sum_R w_R^{\mathbf{wien}} \sum_{P \in \mathbb{P}^{\mathbf{wien}}(R)} X(P,x)\, u_{P,R}^{\mathbf{wien}}(x)}{\sum_R w_R^{\mathbf{wien}} \sum_{P \in \mathbb{P}^{\mathbf{wien}}(R)} X(P,x)}$$

where $u_{P,R}^{\mathbf{wien}}(x)$ is the value of pixel x from the patch P in the denoised stack $\mathbb{P}^{\mathbf{wien}}(R)$.

[10] In our experiments, we compose $\tau_{3D}^{\mathbf{wien}}$ from the same transformations as in the first phase (i.e., 2D Cosine transform and 1D Walsch–Hadamard transform).

5 Parallel implementation

Our implementation is designed to accelerate denoising of a single image. Denoising multiple images can introduce another level of parallelism; however, even a single denoising task is very computationally demanding and able to saturate the most powerful GPUs of the day. The input image is loaded entirely to the GPU global memory, and all intermediate results are kept in the global memory as well. Hence, no host-GPU data transfers are required except for the input image and the denoised result.

We will explain the parallel implementation on the first phase since the second phase reuses most of its methods. The extension to the second phase is straightforward; therefore, it is mentioned only briefly at the end of this section. Since most of the kernels can be used in both phases, we omit the $^{\mathbf{hard}}$ and $^{\mathbf{wien}}$ upper scripts in the notation.

For implementation reasons, we have divided the individual steps of the first phase into multiple kernels which are executed as a sequence. Figure 3 summarizes the kernel execution while pointing out the lifetime of the data buffers and the data flow between the kernels.

The block-matching kernel constructs a 3D group for each reference patch. Since preserving the copies of the patches is extremely memory demanding and since the grouping process needs to select *N* the most similar patches, the block-matching kernel keeps only the coordinates of the patches within the original image. Furthermore, since the number of patches in groups may differ from 1 to *N*, the buffers are allocated for the maximal possible value (*N*) and the exact numbers of patches are kept in a separate buffer.

The collaborative filtering step is divided into four kernels. This modularity allowed us to experiment with different combinations of decorrelating functions. For instance, we can replace our custom 2D transformation kernels with existing CUDA libraries like cuFFT (CUDA Fast Fourier transform [16]). Furthermore, the 3D transformation has to be decomposed to 2D and 1D transformation, since the 3rd dimension (the number of patches) differs for each group. This decomposition may be suboptimal from the global memory data transfers perspective; however, this is not a serious issue since the computational time significantly dominates the data transfer time. The presented solution uses 2D discrete cosine transform and 1D Fast Walsh–Hadamard transform.

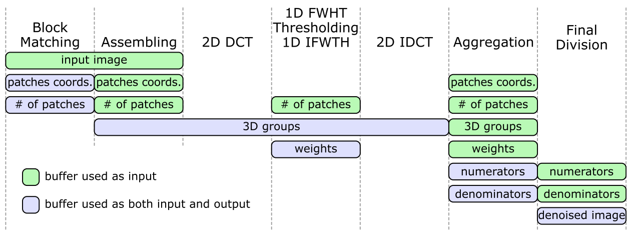


Fig. 3 Schema of CUDA kernels and the CUDA buffers data flow

The aggregation process constructs the numerator buffer and the denominator buffer. These buffers correspond to the values of the pixel estimates in the aggregation formula described in Sect. 4.1.3. We need to keep these values in two buffers since the formula is a fraction. Once all 3D groups are aggregated, the last kernel computes the pixel estimates by dividing corresponding numerator and denominator values.

When denoising a large image, the amount of memory required by intermediate data (especially 3D groups) could very well exceed the size of the global memory of contemporary GPUs. Furthermore, on desktop computers (where the GPU is also used to display the user interface) CUDA imposes an execution time limit for each kernel which can be easily exceeded by our solution. Therefore, the algorithm processes the image in batches that are areas[11] from which the reference patches are selected. However, since some patches similar to the reference ones can originate outside of the batch area, the kernels work with the area enlarged by the neighborhood of the reference patches. Because the enlarged areas overlap, the numerator and denominator buffers are shared by all batches and hold the values for the entire image. All the other auxiliary buffers are being reused by the batches. Final division of numerator and denominator buffers is launched once all the batches are processed.

5.1 Block matching

The block-matching algorithm offers many opportunities for employing data parallelism. Multiple reference patches can be processed concurrently, distances between reference patch and patches in its neighborhood (the $n \times n$ window) can be computed concurrently, and even the L_2 distance function itself can be parallelized internally. However, the construction of a 3D group (the selection of the most similar *N* patches) has to be synchronized if the distances are computed concurrently and the L_2 distance computation would require synchronization in the final reduction. Furthermore, naïve parallelization would lead to poor utilization of memory caches and the shared memory and it may not be suitable for the lock-step execution model of the GPUs.

We have experimented with various approaches to divide the work among the threads. An approach which seems to provide the best ratio between data caching and core occupancy (so it exhibits the best performance results)

[11] Bath area was empirically selected as 256×128 pixels.

was selected. One batch is processed by one execution of block-matching kernel. Each thread block process W_{size} (size of the warp—i.e., 32) reference patches, which are consecutive in the horizontal direction. In other words, the block performs matching on patches with coordinates $x = x_{\text{block}} + i \cdot p$ and $y = y_{\text{block}}$ where $i = 0\ldots31$ and $(x_{\text{block}}, y_{\text{block}})$ are the basic coordinates (i.e., the top-left coordinate of the first patch) for the block.

The exact number of threads[12] in the block is determined by the hardware capabilities of the GPU such as number of registers or size of the shared memory. We denote T_{idx} the index of a thread T within its block. Additionally, we denote T_{line} the index of a thread T within its warp ($T_{\text{idx}} \bmod W_{\text{size}}$) and T_{warp} the index of the warp where the thread belongs to ($\lfloor T_{\text{idx}}/W_{\text{size}} \rfloor$). Let us emphasize that the actual number of threads is always divisible by the number warp size W_{size} and the number of warps in a block is denoted W_{count}.

Threads with the same T_{line} cooperate to process one reference patch—i.e., each thread identifies its reference patch horizontal coordinate as $x = x_{\text{block}} + T_{\text{line}} \cdot p$. The threads iteratively compute distances to all patches in the n times n window determined by their corresponding reference patch where each thread is responsible for a different set of patches. Once a distance is computed, the thread compares it with τ and inserts it into the corresponding 3D group if it passes the threshold.

5.1.1 Cooperative distance calculations

Since the patches are overlapping, parts of the L_2 distance computations overlap as well. In order to decrease the amount of redundant work and to improve caching, the distances are computed cooperatively. The cooperation uses the fact that threads in one warp process different (yet overlapping) reference patches, but they share the relative displacement of the patches to which the distances are being computed. A visualization of this overlapping is depicted in Fig. 4.

The overlapping reference patches of a thread block cover an area denoted R_{area}. Size of this area is determined by the space between two adjacent reference patches[13] p, by the number of these patches in a warp W_{size} and by the size of each patch k. Since the reference patches in a warp are consecutive in the horizontal direction, the R_{area} is $(W_{\text{size}} - 1)p + k$ pixels wide and k pixels high.

Threads of a single warp simultaneously compute the distance between reference patches of R_{area} and patches in

[12] Using default parameters, the selected number of threads on presented GPU architectures is 640 in the first phase and 320 in the second phase.

[13] Let us remember that $1 \leq p \leq k$ holds.

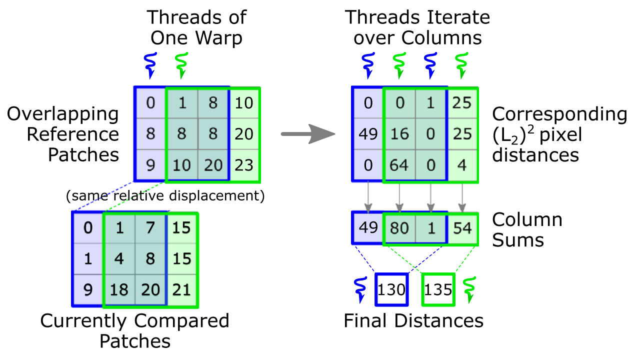

Threads of One Warp
Threads Iterate over Columns
Overlapping Reference Patches
Corresponding $(L_2)^2$ pixel distances
(same relative displacement)
Column Sums
Currently Compared Patches
Final Distances
0 1 8 10
8 8 8 20
9 10 20 23
0 0 1 25
49 16 0 25
0 64 0 4
0 1 7 15
1 4 8 15
9 18 20 21
49 80 1 54
130
135

Fig. 4 Cooperative distance computation of two overlapping pairs of patches

their neighborhood that share the relative displacement. These patches form an area denoted P_{area} of the same size as R_{area}. Since there are multiple warps in a block, each warp has a pre-defined set of displacements and corresponding P_{area}s to process.

Threads in a warp compute the L_2 distances between pixels in their R_{area} and P_{area} so that each column is processed by one thread and the threads iterate over the columns (each thread computes columns $i \cdot W_{\text{size}} + T_{\text{line}}$ for $i = 0, 1, \ldots$). The computed column sums are stored into the shared memory. Once they are all computed, each thread in the warp adds up the column sums corresponding to its own reference patch and the whole warp proceeds to the next P_{area}. Each warp has its pre-defined set of displacements between R_{area} and P_{area} to process. This procedure is repeated until all valid displacements between the reference patches and the patches in their neighborhood are covered. The complete schema of the work decomposition is depicted in Fig. 5.

We cache the R_{area} manually in the shared memory since it is used by all threads in a block. Caching of P_{area} blocks for each warp is left to hardware caches. The P_{area} blocks are overlapping horizontally like the patches, so we expect high level of cache hits. Furthermore, individual warps in the block are not synchronized so it would be tedious to cache these values manually.

5.1.2 Selecting the most similar patches

Once a distance is computed by a thread, it is compared with λ_{3D} constant. If it passes this threshold, the patch coordinates and the distance itself are stored in the corresponding list of the top-N patches. Since the N value is typically small (in our case 16 or 32), simple sorted list is more suitable than more elaborate data structures, like regular heap. Each thread has its private copy of the list in the shared memory, so no data synchronization is required.

If a list contains less than N patches, the new patch reference is simply added. Otherwise, the new patch

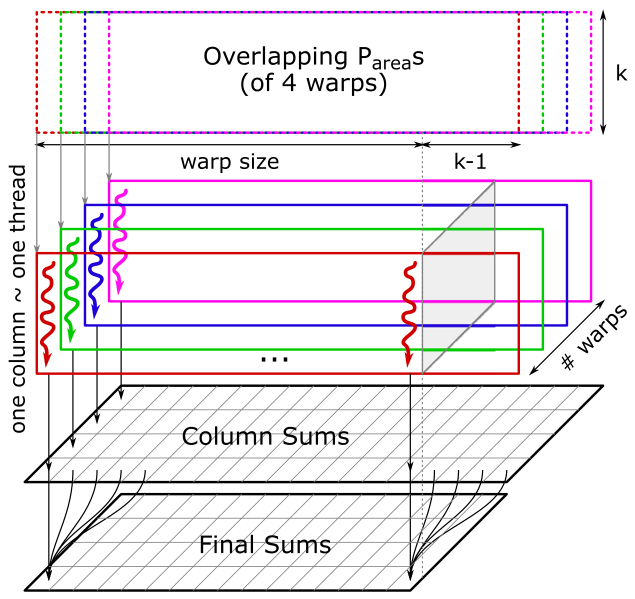


Fig. 5 Schema of work division among threads in a block when computing distances (in case step $p = 1$)

distance is compared to a distance of the least similar patch and replaced if it is smaller. After new reference is added or the last reference is updated, the list order is fixed using one step of insertion sort algorithm. Keeping the list sorted this way may be suboptimal from the algorithmic point of view; however, inserting sort has very small overhead especially on short lists.

When the entire window of patches is processed, the shared memory contains W_{count} copies (one for each warp) of W_{size} patch stacks (one of each line). The first warp aggregates the private stack copies, while each thread operates on its own instances (Fig. 6). This algorithm performs the merging of individual privatized copies serially which may lead to significant code divergence in the lock-step execution of the first warp. On the other hand, this solution is very easy to implement and its performance is more than sufficient since the L_2 distance computations represent the dominant workload.

Even though the privatization of the lists depicted in Fig. 6 leads to synchronization-free updates of these lists, it significantly increases the shared memory requirements of a thread block. It may even lead to suboptimal core occupancy as the memory demand would prevent sufficient number of warps to be executed simultaneously, especially on the Kepler architecture. In order to counteract this problem, the lists were reduced to their minimal size.

The list items comprise a patch reference (relative 2D offset) and the computed distance. The relative offset is limited by the size of the $n \times n$ window in which the patches are located. In our case, the coordinates are rather small since $n = 39$. The unnormalized distance is also an integer as the pixel values are between 0 and 255. Its maximal value is determined as $\tau \times k^2$, which in our case is $2500 \times 64 = 160{,}000$ so the distance should not take more than 18 bits. Therefore, we can compose both the distance and the relative offset of the patch into 32-bit integer[14] and if the distance occupies the highest bits of the integer, the values can be compared directly in the sorting process.

5.2 Collaborative filtering and aggregation

The assembling kernel takes the patch coordinates computed by the block-matching step and prepares the 3D groups for filtering by copying the patches from the image into separate buffers. This part is quite memory demanding since each group takes $O(k^2N)$ memory. However, thanks to the iterative processing of batches the total memory consumption is about hundreds of megabytes for a typical selection of parameters, which is well within the current GPU memory sizes.

The main work is performed in the kernel which combines 1D transformation, hard thresholding, and inverse 1D transformation. Since the transformation is performed independently on $k \times k$ columns of the 3D group, each one is computed by a single thread of a block and the data are kept in shared memory. This kernel is also responsible for computing the number of retained (nonzero) coefficients after the hard thresholding. Each thread counts the retained coefficients in its own column, and the threads in the block employ parallel reduction algorithm which uses warp shuffle functions to speed up the process.

Once the 3D groups are filtered, their patches are multiplied with their respective weights and added to correct locations within the numerator buffer. Analogically, the weights are added to corresponding locations in the denominator buffer. Since both buffers are shared by all threads, the operation is performed by `atomic_add` instruction. The threads from one block perform the updates of one patch, which means that they do not create memory conflicts among themselves and they promote the utilization of caches where the atomic updates are actually resolved.

The utilization of atomics creates possible bottleneck. However, we do not believe that a more elaborate solution for avoiding the atomics collisions would perform better since the buffers are as large as the image (or at least as large as the part of the image processed by current batch). Any attempt to privatize portions of these buffers would have an extensive memory footprint which will lead to

[14] In the implementation, we have decided to use 16 bits for distance and 2×8 bits for offsets, thus reducing the precision of the distance. One of the reasons is that in the future the distance could be saved as floating-point with half precision instead of 16-bit integer.

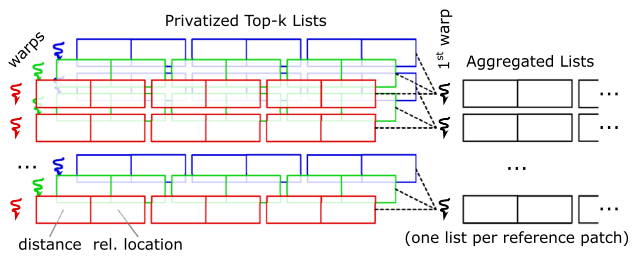


Fig. 6 Schema of the privatized lists of the most similar patches

serious decrease in core occupancy (thus limiting the parallelism much more than the collisions of atomic updates). The privatization effort would also require subsequent reduction step which would have to be in a separate kernel.

The collisions of atomic operations occur only when two thread blocks simultaneously update the same patch or patches that are overlapping in a certain way. Since the patches are rather small, even collisions in the same window are rare. Furthermore, the probability of a collision could be decreased even more by assigning more distant reference patches to simultaneously running thread blocks. If the reference patches are further apart, the intersection of their respective windows is smaller, so it is less likely that their patches will overlap. However, this approach exhibited decrease in performance, which is likely to be caused by decrease in L2 cache hits.

5.3 Extension to the second phase

The second phase follows almost the same workflow as the first phase. There is only one extra copy of the 3D groups buffer for calculating the Wiener coefficients, and the modification is illustrated in Fig. 7. Except for the Wiener filtering (and 1D transformation) kernel, all kernels are reused from the first phase with no changes.

At the beginning of the second phase, block matching is launched on the basic image estimate. Produced coordinates are subsequently used for assembling basic 3D groups from the basic image estimate as well as 3D groups from the original noisy image. Both the 3D groups are subsequently transformed by 2D transform kernel. These two transformed groups are then entering the Wiener filtering kernel that performs the 1D transformations, computation of the Wiener coefficients, and the filtering itself. The remaining steps of the second phase (i.e., the inverse 2D transformation, aggregation, and final division) are exactly the same as in the first phase.

5.3.1 Wiener filtering

Although the Wiener filtering is based on a different concept than the hard thresholding, the architecture of the kernel is very similar. Each thread block works with one reference patch and its respective 3D groups.

Wiener filtering kernel combines 1D transformation of both 3D groups, calculation of the Wiener coefficients, the filtering itself (which is basically multiplication), and the inverse 1D transform of the filtered 3D group. Once again the computation is done in $k \times k$ columns of the 3D group where each thread processes one column. All the intermediate results (transformed group, Wiener coefficients) are traditionally stored in the shared memory.

This kernel is also responsible for computing weights of the 3D group. Each thread counts the squared Euclidean norm (L_2^2) in its own column, and the whole block employs the parallel reduction algorithm for the final summation.

6 Experiments

We compared our **CUDA** solution with a baseline open source **CPU**-only implementation of M. Lebrun [9] which is written in C++ and uses OpenMP for parallelization and with **OpenCL** implementation written by Sarjanoja [17].

The OpenMP implementation allocates buffers for the processing of entire image at once which makes it extremely memory demanding for large images. On the other hand, it employs an efficient method of integral images [6] for block matching as well as an efficient FFTW library for 2D transformations.

The OpenCL implementation is a simple straightforward attempt to employ GPU in BM3D computation. The block-matching kernel is implemented so that each thread finds similar patches to a single reference patch without any cooperation with other threads. In the filtering and aggregation steps, each thread is assigned some percentage of output pixels to process. Naturally, it suffers from many redundant computations as well as from memory inefficiency.

The proposed method deals with the problems of the existing implementations: it processes the image in batches which reduces the memory consumption; it has significantly fewer redundant operations; and it is designed to efficiently utilize memory hierarchy of the GPU. The only cost is the reduced precision in distance computation which has a little influence on denoising quality as it is presented at the end of this chapter.

Besides the implementation comparison in terms of execution time and denoising performance, we also analyzed individual kernels of our CUDA implementation through profiling.

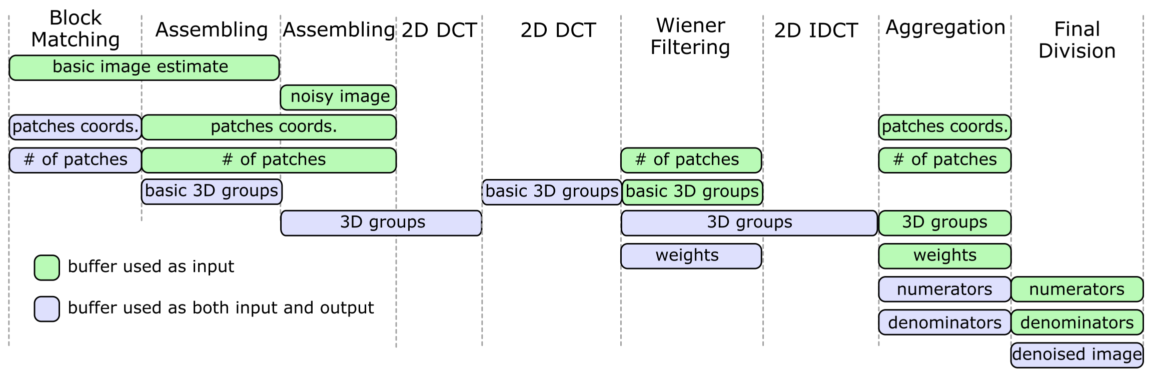


Fig. 7 Schema of CUDA kernels and the CUDA buffers data flow in the second phase

6.1 Hardware and methodology

We have tested and compared the implementations on three hardware platforms:

- A **Server** with NVIDIA Tesla K20m (2496 cores, Kepler) GPU, two Intel Xeon CPUs (2×6 cores @2.3 GHz), and 28 GB of RAM.
- A high-end **PC** with NVIDIA GeForce GTX 980 (2048 cores, Maxwell) GPU, two Intel Xeon CPUs (2×6 cores @1.6 GHz), and 96 GB of RAM.
- A regular desktop **PC2** with NVIDIA GTX 1080 (2560 cores, Pascal) GPU, Intel Core i7 CPU (4 cores @3.60 GHz), and 16 GB of RAM.

The experiments used three ordinary camera-produced images. The pictures were scaled to different (lower) resolutions and corrupted by white noise with standard deviation $\sigma = 25$.[15] Every single noisy image was denoised by all implementations, and the measured execution time included memory allocations and data transfers between the host and the GPU.

All implementations were tested using the same parameters, except of the 1D transform in OpenCL implementation. However, due to the complexity of the algorithm we assume that the choice of 1D transform would not have significant influence on execution time.

6.2 Performance results

Average execution times of the BM3D method are presented in Fig. 8. The implementations scale almost linearly with the image size, but the CPU version benefits from larger images as they can better saturate all available cores and reduce the overhead of thread initialization and synchronization. Let us emphasize that the OpenCL implementation is slower than CPU implementation on the Server, since it has older GPU (K20m), but it has two powerful Xeon CPUs. Furthermore, we have limited the *y*-axis range so the slowest solution scales out, but on the other hand results of our implementation could be read our more precisely.

It is also important to note that our CUDA implementation is very conservative in the memory demand since it processes the image in batches and thus keeps less 3D stacks in device memory simultaneously. For instance, while the OpenMP implementation requires 53 GB of RAM to process 14 Mpix image, our implementation requires only 0.7 GB of device memory and 0.3 GB of RAM.

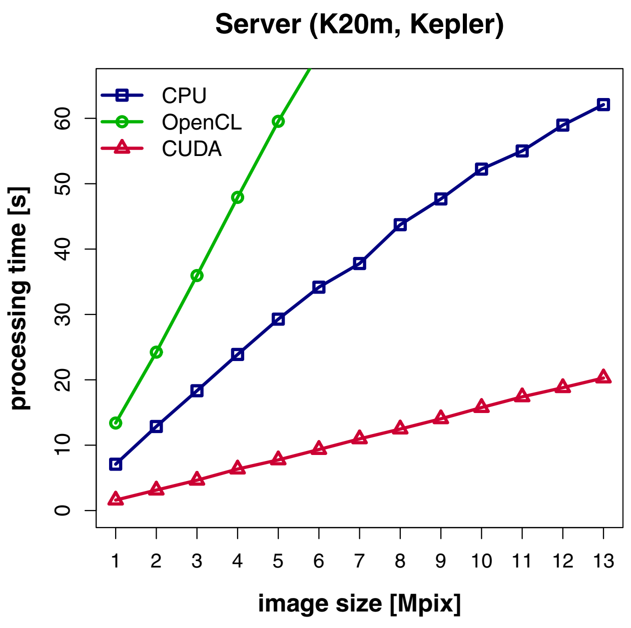
We were unable to test CPU and OpenCL versions on the regular desktop (PC2); however, we could measure the performance of the CUDA implementation, and we can compare it with the performance of other GPUs. Figure 9 presents the CUDA results on all three tested platforms.

The relative speedup of our solution with respect to both CPU version and OpenCL version is presented in Fig. 10. Despite the older GPU and powerful CPUs on the Server, the CUDA version outperforms the CPU more than $3\times$. On the high-end desktop (PC), the GPU outperforms the CPU approximately $30\times$ and the existing OpenCL implementation about $10\times$. Since we could not measure the CPU nor OpenCL implementations on the second desktop (PC2), we have at least compared the CUDA results with the CPU results of the high-end desktop. Even though these results were measured on different machines, the CUDA implementation is almost completely dependent only on the characteristics of the GPU, so the presented speedup is still relevant. The GTX 1080 outperformed two Intel Xeon CPUs almost by the factor of 50.

6.2.1 Performance profiling

The distribution of execution time among all kernels is presented in Figs. 11 and 12 presents aggregated times from both phases. Acronym *BM* stands for block matching,

[15] Unlike the image size, choice of σ has very little influence on the execution time.



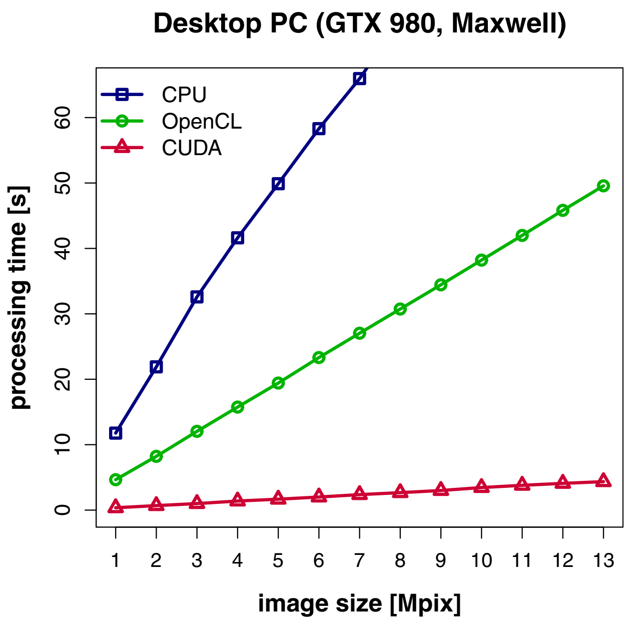


Fig. 8 Average real processing times of images of a different size

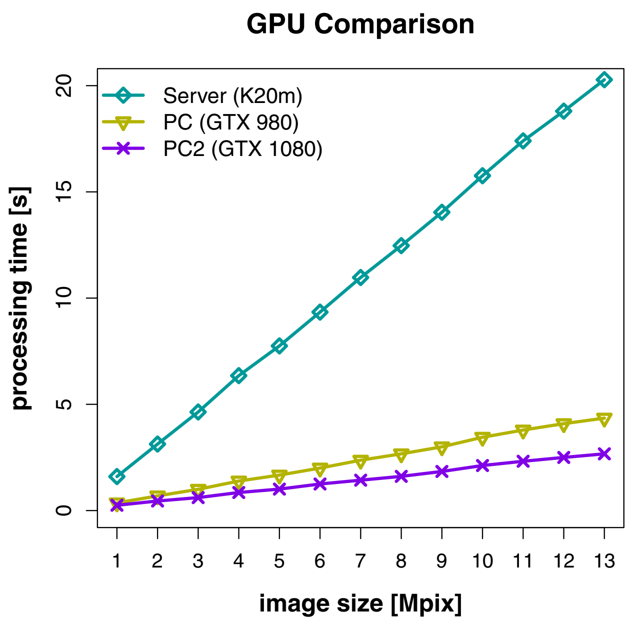


Fig. 9 Comparing real processing times of different GPU architectures

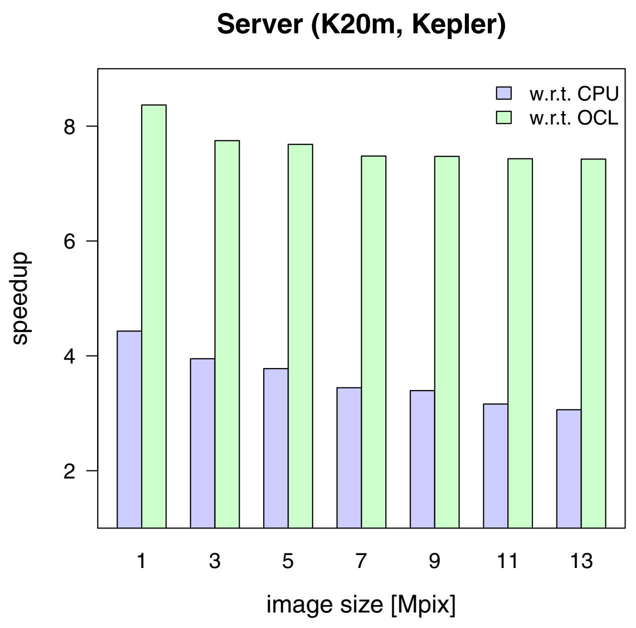
Asm is assembling, *DCT* and *IDCT* is forward and inverse 2D DCT transform, respectively, *1D* is the 1D transform combined with thresholding or Wiener filtering, and *Aggr* is the final aggregation. Let us note that all kernels take more time in the second phase, as they use larger 3D groups ($N^{\mathrm{wien}} = 32$) and some of them are performed both on the noisy image and on the image denoised by the first phase. The aggregated kernel times (Fig. 12) also denote how many times a kernel is executed (e.g., *Asm* $3\times$ means that the assembling kernel was used once on the noisy image in the first phase, once on the noisy image in the second phase, and once on the basic image estimate in the second phase).

On all GPU architectures, the block-matching kernel dominates the execution time; therefore, any further attempts for optimization should be focused here first. The second most time demanding step is the 1D transform, especially when applied with Wiener filtering. On the other hand, both these steps take less relative time on newer architectures. This is mainly caused by the memory demands of the steps in question and since newer architectures provide more shared memory and cache size per core, the core occupancy as well as cache hits increase.

To assess the performance details further, the code was subjected to profiling that determined core occupancy, branch divergence, cache hits, and other important characteristics. The first phase is quite optimized, but some issues remained:

- The *block-matching* kernel suffers from slightly higher code divergence, which is caused by the updates in the top-*N* lists of the most similar patches (using Insert sorting step).
- The *1D transformation with hard thresholding* is limited by shared memory requirements per thread. Current shared memory size prevents to launch enough warps to saturate all cores of a multiprocessor.
- The *aggregation* step exhibits lower throughput of L2 cache, which is likely to be caused by collisions of atomic updates.

We were unable to improve these issues any further as any attempt to solve them lead to more complicated code,



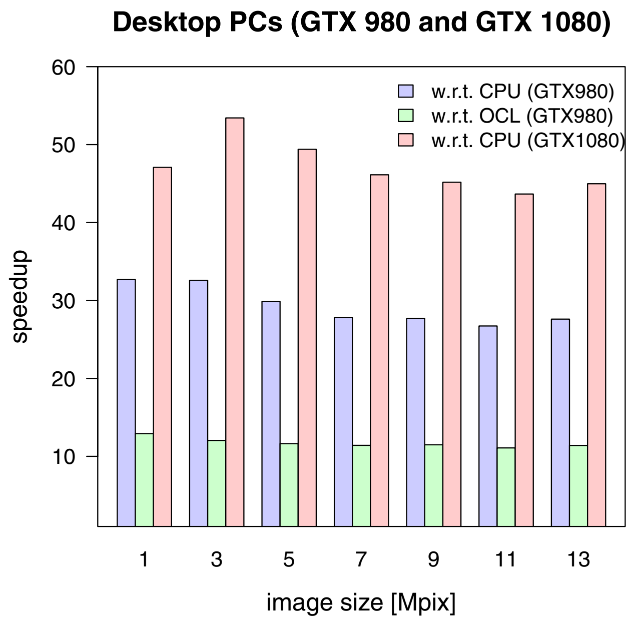


Fig. 10 Speedups achieved with respect to CPU and OpenCL implementations

redundant work, or other problems which at the end caused a decrease in overall performance. Furthermore, the presented issues do not affect overall performance seriously and even if they were solved completely the improvement in performance would be tens of percents at most.

The second phase re-utilizes substantial part of the first phase algorithms; however, there are some differences. The *block-matching* step has reduced core occupancy in the second phase, as it requires almost twice as much shared memory. Similarly, the *Wiener filtering* is quite memory demanding and the utilization of shared memory limits the occupancy. On the contemporary architectures, there is little we can do to improve these issues. However, both issues may resolve naturally if the amount of shared memory is increased in the future generations of GPUs. It may be also possible to keep the data in question in global memory instead of shared memory and still process them efficiently if the global memory is faster (e.g., when new HBM is present on the chip) or if the L2 cache size increases.

6.3 Denoising performance

We measure the denoising performance of the tested images corrupted by white noise with variance $\sigma = 25$ using *peak signal-to-noise ratio* (PSNR) which is in our case (using 8 bits per pixel) defined as:

$$\mathrm{PSNR} = 10 \cdot \log_{10} \frac{255^2}{\mathrm{MSE}}$$

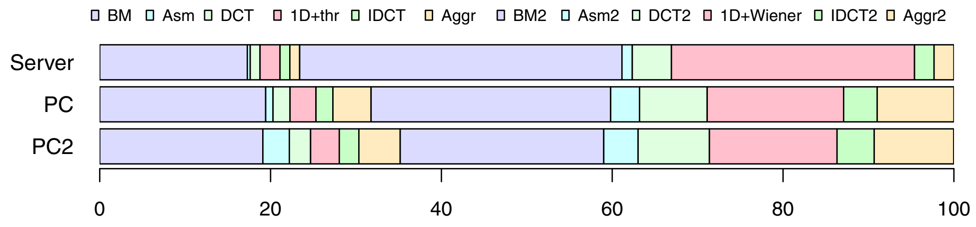
The *MSE* stands for *Mean Squared Error* between original (noiseless) image and final image estimate.

On the tested images, our implementation reaches PSNR values that are on average lower by 0.08 dB compared to the OpenMP implementation. It is probably caused by the rounding errors in aggregation step when we store the basic image estimate as 8-bit image, whereas the OpenMP implementation stores it in floating-point representation. However, for a subjective perception of a real photograph, such difference is completely negligible.

On the other hand, if we compare the OpenCL and OpenMP implementations, the PSNR of the images denoised by OpenCL is on average lower by 0.34 dB. The difference varies with the image being denoised which is presumably caused by different 1D transformations used in these implementations.

More elaborate experiments regarding the denoising performance are beyond the scope of this paper. The measured results indicate that the GPU parallelization has

Fig. 11 Relative time spent in individual kernels



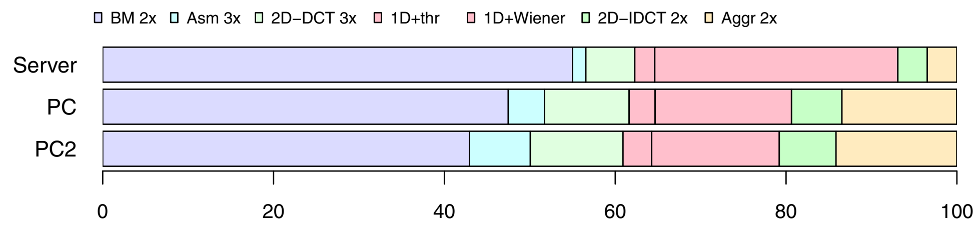


Fig. 12 Relative time spent in individual types of kernels (sum from both phases)

not affected the denoising performance in a significant way.

7 Conclusions

We have presented a very optimized parallel solution of the block-matching 3D filtering algorithm for image denoising which harness the raw computational power of modern GPUs via CUDA framework. Despite the fact that the BM3D presents many opportunities for parallelism, it is also very data intensive and the parallelism needs to be expressed in a way that promotes both lock-step execution model and memory cache utilization. On the desktop computers, our method is up to 50× faster with regular photographs than OpenMP implementation of Lebrun and it has much lower memory footprint. Furthermore, we have outperformed existing OpenCL implementation by an order of magnitude on regular desktop PCs.

This improvement allows us to reduce processing time of regular photographs from the order of minutes to the order of seconds. For instance, a 6 Mpix image could be denoised in a little over one second instead of one minute. Such difference may be the key to bridge the gap of BM3D applicability since regular users may be willing to wait seconds when editing photographs, but they certainly do not wish to wait minutes.

Although the proposed design is targeted to grayscale images, the presented method remains valid for color images as well. The color BM3D algorithm (CBM3D) [5] employs the block matching solely on the luminance channel, while the collaborative filtering and the aggregation are performed for each channel separately. Therefore, the extension of the proposed implementation to color images would be straightforward and it would utilize all the proposed kernels almost as is.

In the future work, it might be interesting to compare various transformations used in collaborative filtering and various algorithm parameters from the perspective of denoising quality and performance to establish the best trade-offs. Furthermore, our algorithm is designed to run on a single GPU, but it could be easily extended to utilize multiple GPUs in order to achieve peak performance on high-end computers and on Servers.

Acknowledgements This paper was supported by Czech Science Foundation (GAČR), Project Number P103-14-14292P, and by Specific Research Project SVV-2017-260451.

References

1. Buades, A., Coll, B., Morel, J.-M.: A non-local algorithm for image denoising. In: IEEE Conference on Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on, vol. 2, pp. 60–65. IEEE (2005)
2. Chen, Y., Pock, T., Ranftl, R., Bischof, H.: Revisiting loss-specific training of filter-based mrfs for image restoration. In: German Conference on Pattern Recognition, pp. 271–281. Springer (2013)
3. Chen, Y., Yu, W., Pock, T.: On learning optimized reaction diffusion processes for effective image restoration. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 5261–5269 (2015)
4. Dabov, K., Foi, A., Katkovnik, V., Egiazarian, K.: Image denoising with block-matching and 3D filtering. In: Proceeding SPIE, Image Processing: Algorithms and Systems, Neural Networks, and Machine Learning Electronic Imaging, p. 606414. International Society for Optics and Photonics (2006)
5. Dabov, K., Foi, A., Katkovnik, V., Egiazarian, K.: Color image denoising via sparse 3D collaborative filtering with grouping constraint in luminance–chrominance space. Image Process. **1**, I-313 (2007)
6. Facciolo, G., Limare, N., Meinhardt-Llopis, E.: Integral images for block matching. Image Process. Line **4**, 344–369 (2014). https://doi.org/10.5201/ipol.2014.57
7. Gu, S., Zhang, L., Zuo, W., Feng, X.: Weighted nuclear norm minimization with application to image denoising. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2862–2869 (2014)
8. Huang, K., Zhang, D., Wang, K.: Non-local means denoising algorithm accelerated by gpu. In: Sixth International Symposium on Multispectral Image Processing and Pattern Recognition, p. 749711. International Society for Optics and Photonics (2009)
9. Lebrun, M.: An analysis and implementation of the bm3d image denoising method. Image Proces. Line **2**, 175 (2012)
10. Mairal, J., Bach, F., Ponce, J., Sapiro, G., Zisserman, A.: Non-local sparse models for image restoration. In: 2009 IEEE 12th International Conference on Computer Vision, pp. 2272–2279. IEEE (2009)
11. Márques, A., Pardo, A.: Implementation of non local means filter in gpus. In: Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications, pp. 407–414. Springer (2013)
12. NVIDIA.: Kepler GPU Architecture. http://www.nvidia.com/object/nvidia-kepler.html (2017). Accessed 10 Nov 2017
13. NVIDIA.: Maxwell GPU Architecture. http://developer.nvidia.com/maxwell-compute-architecture (2017). Accessed 10 Nov 2017

14. NVIDIA.: Pascal GPU Architecture. https://developer.nvidia.com/pascal (2017). Accessed 10 Nov 2017
15. NVIDIA.: CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/ (2017). Accessed 10 Nov 2017
16. CUDA Nvidia. CUFFT library. https://developer.nvidia.com/cufft (2010). Accessed 27 Nov 2017
17. Sarjanoja, S.: Opencl implementation of bm3d image denoising algorithm. https://github.com/Sampas/bm3dcl (2015). Accessed 10 Nov 2017
18. Sarjanoja, S., Boutellier, J., Hannuksela, J.: Bm3d image denoising using heterogeneous computing platforms. In: 2015 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 1–8. IEEE (2015)
19. Schmidt, U., Roth, S.: Shrinkage fields for effective image restoration. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2774–2781 (2014)
20. Zheng, Z., Xu, W., Mueller, K.: Performance tuning for cuda-accelerated neighborhood denoising filters. In: Workshop on High Performance Image Reconstruction (HPIR) (2011)
21. Zoran, D., Weiss, Y.: From learning models of natural image patches to whole image restoration. In: 2011 International Conference on Computer Vision, pp. 479–486. IEEE (2011)

David Honzátko received a bachelor's degree from General Computer Science at Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic, and he is currently a master student in Artificial Intelligence at the same institution. His research interests include image processing, parallel computing, and machine learning.

Martin Kruliš received a doctoral degree from software systems at the Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic, and he is currently working as an assistant professor at the same institution. He is a member of parallel architectures/applications/algorithms research group and similarity retrieval research group, and his research interests include parallel programming, high performance computing, and concurrency in (multimedia) database systems.