Implementation of BM3D Filter on Intel Xeon Phi for Rendering in Blender Cycles

Milan Jaros, Petr Strakos(✉), and Tomas Karasek

IT4Innovations, VSB - Technical University of Ostrava,
17. listopadu 15/2172, 708 33 Ostrava-Poruba, Czech Republic
{milan.jaros,petr.strakos,tomas.karasek}@vsb.cz

Abstract. In this paper parallel implementation of Sparse 3D Transform-Domain Collaborative filter (BM3D) on the Intel Xeon Phi architecture is presented. Efficiency of the implementation in terms of speedup compared to serial implementation of the filter is demonstrated on denoising of rendered images. We also provide comparison with another parallel CPU version and show that ours performs better.

Using the state-of-the-art image filters such as BM3D offers powerful denoising capability in the area of image filtering. To achieve the highest possible quality of the result, the filter has to perform multiple demanding tasks over a single image. Effective implementation of the filter is therefore very important. This is also the case, when filtering is used for image rendering. Rendering times can be significantly decreased by application of powerful time efficient denoising filters. Unfortunately the existing serial implementation of the BM3D filter is time consuming. In this paper we provide efficient parallel implementation of the BM3D filter, and we apply it as a noise reduction technique to the rendered images that reduces the rendering times. We also provide an optimized version of the filter for the Intel Xeon Phi and Intel Xeon architecture.

Keywords: Image denoising · Intel Xeon Phi · Blender cycles
Rendering · Collaborative filtering · High performance computing

1 Introduction

There are plenty of areas where advanced image filtering methods can be employed. They are extensively used in medical imaging, computer vision, or they can be effectively applied in the area of image rendering. Here, very often, path tracing algorithms utilizing the Monte Carlo (MC) method are used [9]. Based on the statistical approach, such rendering systems trace each ray of the light in the scene and calculate its effect on an individual object based on the environmental and material parameters. The results of such renderers bring high level of realism. This is unfortunately paid back by long rendering times. On the other hand, using only a small amount of rendered samples per pixel can reduce the time but also incorporates a high level of noise in the image. To reduce the

T. Kozubek et al. (Eds.): HPCSE 2017, LNCS 11087, pp. 101–114, 2018.
https://doi.org/10.1007/978-3-319-97136-0_8

rendering times while keeping the photo realistic quality, utilization of the image denoising methods has been studied [1,5].

The computationally extensive tasks are usually solved on powerful workstations, or they can use the computational power of supercomputers. Image rendering is one of the areas that generates high computational load. Utilization of supercomputers therefore becomes an interesting idea.

In our contribution, we concentrate on the state-of-the-art image denoising BM3D method and customize it for HPC or multi-core environment. The method has been published in [3] and is one of the best in the area of image denoising [7]. We use the color variant of the method, which has been presented in [2]. Since the method is computationally demanding, its practical potential has been lower. For this reason, we provide its effective parallel implementation. We present implementation that suits two different computer architectures, Intel Xeon and Intel Xeon Phi. Our main focus is put on Intel Xeon Phi with their many integrated core (MIC) architecture. It is nowadays extensively used in supercomputing centres worldwide, apart from the typical CPU architecture. We provide comparison of our parallel version of the filter running on the two architectures with the sequential version of the filter [2]. We also compare our implementation with different parallel implementation from Lebrun [8] that utilizes OpenMP and runs on CPU. We present results in terms of total rendering and filtering times, and we also provide results showing increasing improvement by different optimizations. We also show the positive effect of the filter on the reduction of the rendering time while conserving the final image quality. It is important to mention that we have elaborated the method within our own developed rendering concept called CyclesPhi that can utilize the power of supercomputers, and we make it available within the open-source 3D creation Blender suite.

2 Previous Work

Large amount of algorithms has been studied by researchers in terms of noise reduction in rendering methods based on Monte Carlo. It is possible to divide the algorithms into those trying to modify sampling of the renderer and those using filtering techniques to decrease the residual noise of the renderer.

The recent filtering techniques within the Monte Carlo based rendering systems propose, for example, the iterative approach as in [10]. Here, the authors use a two-step iterative process. First, for the initially rendered image with low samples they use a set of filters in every pixel and select the filter that minimizes the pixel error in terms of mean square error (MSE). Second, for the filter selection they additionally increase the pixel samples based on the filter selection and re-render the image and proceed again through first step. The authors are using discrete set of simple Gaussian filters, which need appropriate amount of samples/pixel (32 samples/pixel used) to work effectively and to obtain satisfactory results.

Another filtering approach presented in [6] considers utilization of the advanced filtering methods in the concept that is primarily intended as a post-processing filtering step. It utilizes the BM3D method as a filter. The authors provide a multilevel denoising algorithm, which first estimates the noise level locally from the close neighbourhood around each image pixel. For such a noise map they reconstruct a histogram of noise levels. Histogram is then divided to several levels defined by the user. To do this, cumulative distribution function (CDF) of the noise map is used. Each discrete level is characterized with the value of standard deviation. For every value of standard deviation filtering is provided. Resulting images are then combined to compute the final image. The provided concept does not depend on used filter, but it performs best with the BM3D method. The authors use the original Matlab C/C++ MEX implementation of BM3D method provided by Dabov et al. in [2,3]. This version does not leverage any parallel programming. Although in context of total rendering time of a single image its runtime is fast, in context of image filtering of larger image series it is slow. This version also can not utilize a specific hardware such as GPU or MIC. In our contribution we provide an optimized parallel version of the color BM3D filter for CPU and MIC architecture.

Concerning the BM3D filter and its available parallel implementations that can serve for general purpose, meaning also for rendering, we have found just two of them [8,11]. The one from Lebrun utilizes multi-core CPUs while the Sarjanoja et al. tackle GPUs. As for the implementation of Sarjanoja et al., they perform better than original Dabov's single threaded version only if they use the so called modified profile. This modified profile unfortunately brings lower filtering quality than the original parameters [4] of the method. Sometimes the difference is negligible sometimes it is not. Within the original profile, beside the lower speed they also run out the memory while filtering UHD (3840×2160) images. An odd thing, which we find in their contribution, is that their testing of Lebrun's parallel version performs worse than the original single threaded version of Dabov's. This is completely opposite to the findings one would expect and it is also opposite to our findings we provide in our contribution within Sect. 7.

3 BM3D Filtering Method

Block-matching and the 3D collaborative filtering method operate over the image trying to minimize the amount of noise based on sparsity of similar image blocks. The filtering method is general with respect to the type of attenuated noise [3], but for ease of explanation the noise is assumed as Additive White Gaussian Noise (AWGN). This can be formulated by the following equation

$$z(x) = y(x) + \eta(x), \tag{1}$$

where z stands for the evaluated image, y is the noiseless image and η is the additive zero-mean Gaussian noise. Variable x stands for the pixel coordinate within the image.

The detailed description of the BM3D method is covered in [3]. The color version of the method is presented in [2]. Here we summarize just the main aspects of the method and accentuate the most computationally extensive parts of the algorithm.

BM3D is a two-step method. In the first step, image is divided into several overlapping areas where similar smaller parts of the image called patches are searched for. Searching for the similar patches is done in a sparse domain provided by wavelet transform of each patch. The found similar patches are stacked in the 3D array and the whole stack is transformed from image to sparse representation by 3D transform (specific combination of transform matrices providing the 3D transform is stated in [3]). Here the filtering operation in the form of the hard thresholding is performed. After this the stack is transformed back to the image domain. This can be symbolically written as

$$\mathbf{Y} = T_{3D}^{-1}(\Upsilon(T_{3D}\mathbf{Z})), \tag{2}$$

where $\mathbf{Y}$ is the stack of filtered patches, T_{3D} is specific 3D transform, Υ represent the filtering operation and $\mathbf{Z}$ is the stack of noisy image patches. This is the core of the method and due to the searching of the similar patches in sparse domain the filtering can be very effective. Each patch from the filtered stack is redistributed back to its position within the image and all overlapping pixels are aggregated and averaged out by weighted average. This can be symbolically written as

$$\widehat{y}(x) = \frac{\sum\sum w \cdot Y(x)}{\sum\sum w \cdot \chi(x)}, \tag{3}$$

where $\widehat{y}$ is the filtered image, w is appropriate weight, Y is a patch from the considered stack of patches, and χ is the patch support. Summation goes through all the patches within one stack and through all the stacks within one image.

In the similar manner, the second step of the method is performed. It uses the results of the preceding step to filter out the noise even further from the noisy input image. Equations 2 and 3 are also used here, differences are only in the way how the searching of similar patches is provided, and how the filtering inside the sparse domain represented by Υ is provided. Both filtering steps of the method are graphically summarized in Fig. 1.

3.1 BM3D - Computationally Extensive Parts

The BM3D method operates in a sparse domain, where all the filtering is performed either by hard-thresholding in Step 1 or by Wiener filtering in Step 2. Conversion to sparse domain is done by matrix multiplication of each selected image patch with transformation matrices. The number of patches that are transformed and then further processed is high. Based on the recommended setting of the method, as elaborated in [4,8], it is around 1300 patches for every single reference patch. The number of the reference patches depends on the image resolution, but generally it is about 1/16th of the number of image pixels. In the case of rendering tasks, image resolution is typically HD (1920×1080 pixels) or

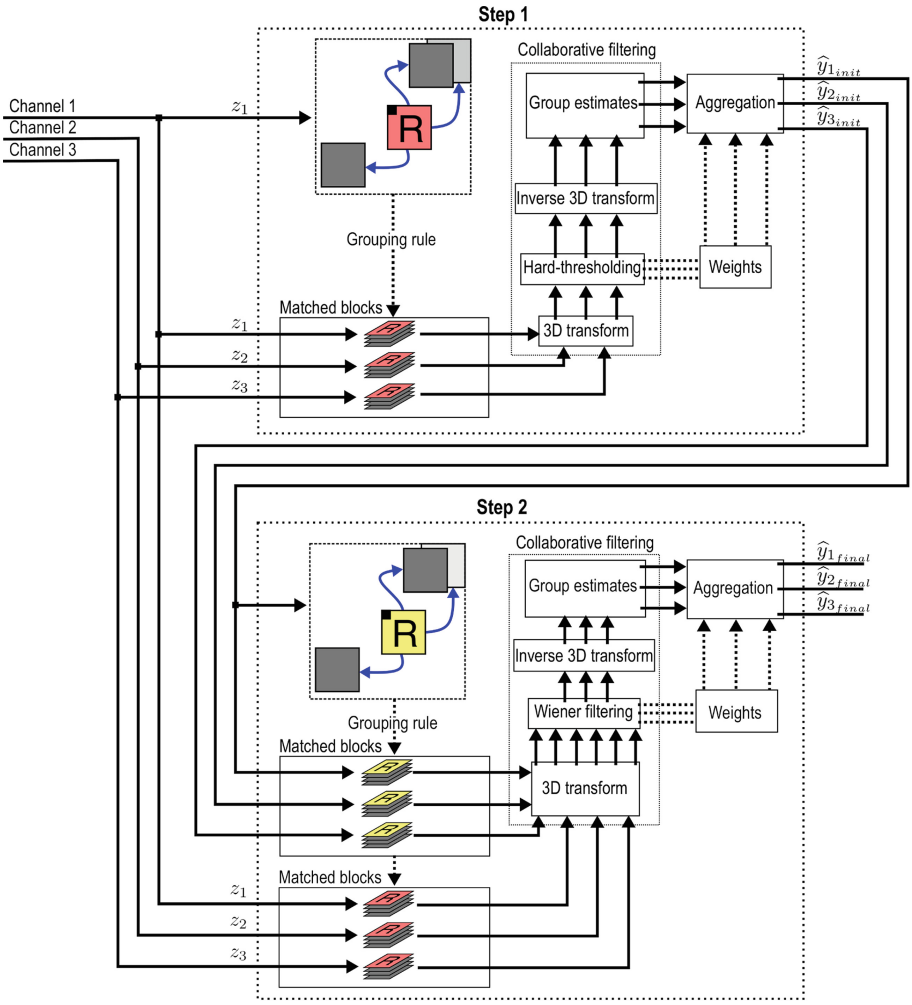

Fig. 1. Workflow of the collaborative filtering method

higher, which gives approximately 130 000 of reference patches at least. Summing it all up, it gives around 340 000 000 of image patches that are being processed just in Step 1. Similar counting holds also for the Step 2. Luckily computations can be parallelized around each reference patch. Limitations are set if one tries for concurrency between Step 1 and Step 2. It is not possible since the results of Step 1 are used right at the start of Step 2.

Another issue arises with pixel aggregation denoted by Eq. 3, where assignment to individual pixel positions is performed. This appears in both of the filtering steps. If one uses parallelization concept where area around each reference patch is solved by individual thread of multiple threads, then one needs

to assign to each pixel position sequentially, because the areas are overlapping and therefore multiple access to same memory location can easily occur if not carefully treated. This is an huge bottleneck and speed barrier if solved trivially by sequential code. If we want to parallelize the aggregation process we have to ensure that at one time instant memory area at specific address is accessed just by one thread and meanwhile non-blocking any other communication from the remaining threads.

Our solution to the mentioned issues using parallelization is described in Sect. 6.

4 Denoising Capabilities of the Collaborative Filtering Method on Rendered Images

As it is experimentally tested in [7], BM3D as a state-of-the-art filtering method outperforms many of the actual image denoising methods. We show its application on two distinct rendered scenes. They differ in the depth of field (DOF). First scene of Tatra car has large DOF, while scene with the worm has small DOF to highlight the worm body from the background. This is to accentuate the typical filtering problem if one wants to preserve the edges and also blur out the noise in the background. Each of the scenes has been rendered for an increasing set of samples per pixel (spp) as shown in Fig. 2, for Tatra and Fig. 3, for the worm respectively. The number of rendered samples/pixel is proportional to the level of noise that remains in the image after the rendering. The higher the number of samples the lower the amount of the residual noise. Filtered images up to 512 samples/pixel are shown in Figs. 4 and 5. For higher sampling filtering looses its effect because the level of noise after rendering is low. A complete list of the computed results is presented in Tables 1 and 2. In case of Tatra scene, it makes sense to use filtering up to 512 samples/pixel and up to 2048 samples/pixel in case of the worm. It is possible to use 2 or 4 times lower number of samples/pixel if we use filtering to obtain the same visual quality of the rendered scene.

5 Sequential Implementation of BM3D Filter

The sequential version of the algorithm is summarized in the following pseudocode. As stated in Sect. 3.1, computationally extensive parts, printed in bold, are concentrated in 3(b)i–3(b)iii and in both aggregation parts 4 and 6.

1. Load image
2. Set parameters and create transformation matrices for Step 1, Step 2 (see [3,8])
3. Step 1 - Calculate the group estimates
   (a) Set positions of reference blocks
   (b) **for** $i = 1 \div R_1$ number of reference blocks in the image
      i. **Do grouping by matching on Channel 1**
      ii. Use created matching rule on Channel 2, 3

Fig. 2. Rendered scene of Tatra car **without** filtering; from left to right, up to bottom - 8196 samples/pixel, 256 samples/pixel, 64 samples/pixel, 1 sample/pixel

Fig. 3. Rendered scene of the worm **without** filtering; from left to right, up to bottom - 8196 samples/pixel, 256 samples/pixel, 64 samples/pixel, 1 sample/pixel

iii. **Perform collaborative filtering**
iv. Compute the weight value for the composed group of blocks

(c) **end**

4. Step 1 - **Aggregate the group estimates, weights and compute the initial image estimate**
5. Step 2 - Calculate the group estimates
   (a) Set positions of reference blocks
   (b) **for** $i = 1 \div R_2$ number of reference blocks in the image
      i. **Do grouping by matching on Channel 1 of the initial image estimate**
      ii. Use created matching rule on Channel 2, 3 of the initial image estimate

Fig. 4. Rendered scene of Tatra car **with** filtering; from left to right, up to bottom - 512 samples/pixel, 256 samples/pixel, 64 samples/pixel, 1 sample/pixel

Fig. 5. Rendered scene of the worm **with** filtering; from left to right, up to bottom - 512 samples/pixel, 256 samples/pixel, 64 samples/pixel, 1 sample/pixel

iii. Collect image blocks from noisy channels using the matching rule
iv. Compute the Wiener coefficients
v. Compute the weight value corresponding to the filtered group of blocks
vi. **Perform the collaborative Wiener filtering**

(c) **end**

6. Step 2 - **Aggregate the group estimates, weights and compute the final image**

Table 1. Computed values for Tatra scene; 8-bit/channel range; FHD (1920 × 1080) resolution; MSE - Mean Square Error, SSIM - Structural Similarity Index, PSNR - Peak Signal-to-Noise Ratio

Tatra	Raw			Filtered			
Samples/pixel	MSE	SSIM	PSNR	MSE	SSIM	PSNR	σ
1	4581.158	0.065	11.521	1358.037	0.647	16.802	59.525
2	2524.610	0.095	14.109	509.139	0.754	21.062	46.469
4	1436.970	0.136	16.556	200.482	0.878	25.110	36.123
8	772.817	0.193	19.250	83.286	0.900	28.925	26.842
64	94.142	0.541	28.393	12.235	0.960	37.255	9.509
128	35.356	0.735	32.646	8.703	0.965	38.734	5.851
256	19.637	0.818	35.200	5.740	0.974	40.542	4.354
512	10.262	0.888	38.019	4.285	0.980	41.812	3.146
1024	3.389	0.959	42.831	3.514	0.983	42.673	1.819
8192	0.000	1.000	-	0.000	1.000	-	0.000

Table 2. Computed values for the worm scene; 8-bit/channel range; FHD (1920×1080) resolution

Worm	Raw			Filtered			
Samples/pixel	MSE	SSIM	PSNR	MSE	SSIM	PSNR	σ
1	6496.412	0.053	10.004	3410.300	0.496	12.803	65.007
2	4880.197	0.071	11.246	1938.514	0.588	15.256	60.648
4	3346.443	0.097	12.885	958.296	0.679	18.316	52.962
8	2015.827	0.131	15.086	408.117	0.763	22.023	42.771
64	352.891	0.350	22.654	47.618	0.836	31.353	18.600
128	173.270	0.477	25.744	22.810	0.894	34.550	13.088
256	89.716	0.599	28.602	13.377	0.928	36.867	9.438
512	40.439	0.743	32.063	8.617	0.951	38.777	6.346
1024	18.643	0.849	35.426	5.661	0.963	40.602	4.313
2048	8.049	0.923	39.073	5.965	0.969	40.374	2.834
4096	2.697	0.971	43.822	6.337	0.973	40.112	1.641
8192	0.000	1.000	-	0.000	1.000	-	0.000

6 Parallel Implementation of BM3D Filter

To achieve the best possible implementation in terms of the algorithm speed, we have implemented it in C++ (Intel Compiler 2017.1) and integrated it under our CyclesPhi rendering engine. We have used OpenMP standard with its *#pragma* directives for parallel programming and *SIMD* directives for vectorization.

We use our own vectorized code for operations with matrices (summation, subtraction, multiplication, L_2 norm) because it performs better than Eigen or MKL libraries on small matrices that are used within the filter implementation. We work mainly with matrices of size 8×8. Due to such small size of matrices filter has to perform hundreds of millions of operations. We solve this issue by *#pragma omp simd* vectorization of the aforementioned operations. By using in-lined functions for the operations we further shorten the computation time. Advantage of using *#pragma omp simd* is that it accommodates to the architecture that is being used (AVX2 and KNC in our case).

Although this implementation is already quite efficient, large bottleneck in matrix multiplications still persisted. This was solved by direct assembler implementation, which helps especially on MIC architecture, as can be seen in results. The assembler code was generated by library for small matrix multiplication (LIBXSMM v1.8). We have generated the assembler code for the commonly used sizes of multiplied matrices.

Another issue to deal with was the memory allocation for the small matrices. This problem is more concerning MIC than CPU. While using Eigen library the time necessary for memory allocation and de-allocation was too high. This was another reason for moving to our own solution. Beside own code for operations with matrices we have defined own classes for the matrices and handle the memory allocation within them. At the beginning, we allocate all the memory necessary for the computations and during the computations we dynamically assign the memory. This way we significantly accelerate the computation and also the initialization of variables that are needed for the task.

For parallelization of serial code the obvious step is to parallelize the operations around each reference patch using *#pragma omp parallel*. It means the for loop 3(b)–3(c), (5(b)–5(c)) in pseudo-code of Sect. 5 was parallelized this way. Another computationally extensive operation in serial code is aggregation of the group estimates (4 and 6. in pseudo-code). Here the parallelization has to be done more carefully, because we are aggregating the results from different memory locations to shared memory area for all threads. Multiple writes to the same part of the memory can occur. To prevent this, while retaining parallelization with its speedup, we efficiently re-ordered the vector of indexes that localize reference patches within the image. Re-ordering prevents patch overlapping between concurrently solved tasks around reference patches. The situation is documented in Fig. 6.

We have implemented the code with focus on MIC architecture, but all the enhancements of the code are fully compatible with CPU architecture. Gradual improvements on both CPU and MIC architecture are documented in the results section.

We have also performed a comparison between our parallel implementation of the BM3D and the one provided by Lebrun [8]. Lebrun implements CPU version of the BM3D filter that can leverage parallelization using OpenMP. We have compared our MIC and CPU versions with Lebrun's in terms of speed and memory demands. For the reference also Dabov's single threaded version is stated [2]. More details can be found in Sect. 7.

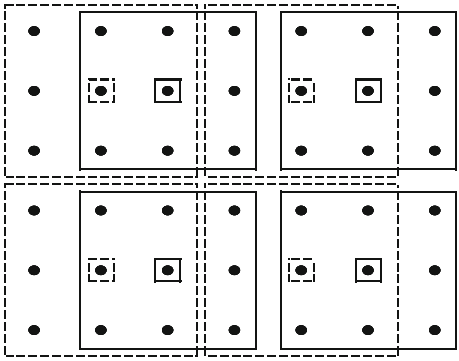

Fig. 6. Re-ordering of reference patches for preventing the area overlapping. Areas outlined by same line type (continuous, dashed) are solved concurrently in parallel. First all areas outlined by dashed line are solved by available thread, before areas outlined by continuous lines are being processed. In this way multiple writes are eliminated

7 Results

Tatra and worm scenes were used as examples for the tests of filter processing time. For a reference we also show the rendering times for different samples per pixel, see Table 3. We have tested the filter for the set of sampling used in Sect. 4. Specific sampling does not influence the filter runtime directly, but it affect the value of standard deviation of noise σ. There is a change in parameters of the filtering method (size of the patch, step between reference patches, etc.) based on the level of noise represented by the σ as recommended in [4,8]. Parameters are different if $\sigma > 40$ or $\sigma \leq 40$. In case of Tatra, sampling from 1 to 2 samples/pixel has $\sigma > 40$, other sampling has lower σ, see Table 1. In case of the worm, up to 8 samples/pixel it is $\sigma > 40$, rest of the sampling has lower σ, see Table 2. This distinguishes the measured filtering runtimes. We have thus computed their mean values from all the measurements in each scene based on the value of σ. Results can be seen in Tables 4 and 5. There are also shown measurements on different architectures (CPU, MIC) with specific optimizations (AVX2, AVX2+SIMD, AVX2+xsmm+SIMD, KNC-offload, KNC-offload+SIMD, KNC-offload+xsmm+SIMD). In our tests KNC-offload stands for Xeon Phi (brand name Knights Corner) with offload programming model. All of these results are compared with the results obtained by running the filter implementation of Dabov et al. [2].

Our speed improvements by different optimization of the filtering algorithm are in graphical form represented in Fig. 7.

Our parallel implementation of the filter was further compared with parallel version by Lebrun. Comparison was made in terms of speed and memory utilization while filtering Tatra images of increasing resolution. For reference we have also compared it with single threaded Dabov's version. Results can be seen in Table 6. Filters are using the recommended parameter setting as stated in [4] under the normal profile. Our solution is much more efficient in terms of memory

Table 3. Rendering times for Tatra and the worm scene for specific samples per pixel; FHD resolution

Rendering time	Time [h:m:s.ms]				
Samples/pixel	Tatra	Worm	Samples/pixel	Tatra	Worm
1	00:04.34	00:32.27	128	00:56.01	07:41.19
2	00:04.68	00:35.73	256	01:48.05	14:57.93
4	00:05.50	00:42.56	512	03:31.49	29:12.96
8	00:07.16	00:55.90	1024	06:58.27	58:13.70
16	00:10.31	01:22.07	2048	14:03.66	01:56:14.17
32	00:16.75	02:16.01	4096	28:02.70	03:52:25.31
64	00:29.68	04:05.13	8192	55:16.09	07:44:07.64

Table 4. Tatra - BM3D runtime for different optimizations and architectures compared with original version of the filter by Dabov et al.; FHD resolution

Filter runtime	Ours [s]		Dabov et al. [s]	
Arch., Inst. set	$\sigma > 40$	$\sigma \leq 40$	$\sigma > 40$	$\sigma \leq 40$
CPU, AVX2	76.9	139.5	107.4	66.4
CPU, AVX2+SIMD	23.0	24.7		
CPU, AVX2+xsmm+SIMD	**12.7**	**15.0**		
MIC, KNC-offload	369.6	598.8		
MIC, KNC-offload+SIMD	81.6	142.3		
MIC, KNC-offload+xsmm+SIMD	19.8	38.1		

Table 5. Worm - BM3D runtime for different optimizations and architectures compared with original version of the filter by Dabov et al.; FHD resolution

Filter runtime	Ours [s]		Dabov et al. [s]	
Arch., Inst. set	$\sigma > 40$	$\sigma \leq 40$	$\sigma > 40$	$\sigma \leq 40$
CPU, AVX2	77.9	141.4	109.0	62.9
CPU, AVX2+SIMD	23.3	25.2		
CPU, AVX2+xsmm+SIMD	**12.8**	**15.1**		
MIC, KNC-offload	370.2	597.9		
MIC, KNC-offload+SIMD	81.6	142.5		
MIC, KNC+xsmm-offload+SIMD	19.9	38.5		

utilization especially if compared with Lebrun, which utilizes incredible 100 GB of memory compared to ours 2 GB for image in FUHD resolution. In terms of speed our solution performs better or is the same up to FHD resolution of the image. On higher resolutions our implementation starts to lag behind Lebrun. Single threaded version performs always worse than others.

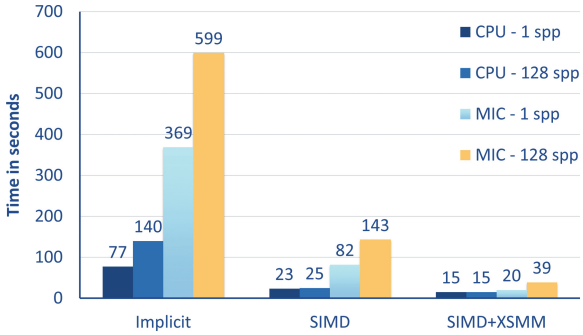

Fig. 7. BM3D runtime for different optimizations and architectures

Table 6. Tatra - our BM3D implementation compared to Lebrun's and Dabov's; images with 128 samples per pixel; $\sigma = 5$ as a filtering parameter; Normal profile from [4] used for BM3D setting

Resolution	CPU		MIC		Dabov		Lebrun	
	[s]	[GB]	[s]	[GB]	[s]	[GB]	[s]	[GB]
768 × 576	**3.1**	**0.14**	7.7	1.51	14.9	0.68	4.4	2.60
1280 × 720	**6.7**	**0.17**	16.2	1.55	31.9	0.76	7.7	4.50
1920 × 1080	15.0	**0.24**	38.1	1.65	73.4	0.92	**14.9**	8.60
2560 × 1440	30.9	**0.33**	67.7	1.76	131.3	1.11	**24.9**	13.50
3840 × 2160	62.2	**0.60**	151.6	2.15	278.9	1.63	**58.6**	28.00
7680 × 4320	295.9	**2.10**	623.8	3.84	955.7	4.20	**204.5**	101.00

All the tests were performed on one computing node of Salomon supercomputer. Specifically on 2x Intel Xeon E5-2680v3, 2.5 GHz for CPU tests and Dabov et al. tests and 1x Intel Xeon Phi 7120P, 61cores for MIC tests.

From the results it can be seen, how our parallelization concept can help in speeding up the algorithm runtime. Although the speedup is not ideally proportional in terms of utilized cores, we can bring up to 8× faster implementation on CPU (24 cores, 24 threads) and up to 5× faster implementation on MIC (61 cores, 244 threads) compared to one core solution of Dabov's.

8 Conclusion

In our contribution we have presented optimized parallel version of the stat-of-the-art filtering technique BM3D. Final version of the algorithm can run on two different architectures (CPU, MIC) and it can be efficiently used on supercomputers. Since compute nodes are often equipped with CPU+MIC our implementation could completely utilize those computing nodes and thus it can be extremely suitable in computationally extensive areas such as image rendering.

Provided implementation is faster than originally presented algorithm. The highest speed-up reaches up to 8× in case of CPU and up to 5× in case of MIC. We perform better also against different parallel implementation of the filter from Lebrun. Our solution is faster on smaller resolutions and utilizes memory much more efficiently. Although Lebrun's version is faster on high resolutions it becomes impractical due to extreme memory demands.

Acknowledgements. This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project "IT4Innovations National Supercomputing Center - LM2015070".

References

1. Bauszat, P., Eisemann, M., Magnor, M.: Guided image filtering for interactive high-quality global illumination. Comput. Graph. Forum **30**(4), 1361–1368 (2011)
2. Dabov, K., Foi, A., Katkovnik, V., Egiazarian, K.: Color image denoising via sparse 3D collaborative filtering with grouping constraint in luminance-chrominance space. In: Proceedings - International Conference on Image Processing, ICIP, vol. 1, pp. I313–I316 (2006)
3. Dabov, K., Foi, A., Katkovnik, V., Egiazarian, K.: Image denoising with block-matching and 3D filtering. In: Proceedings of SPIE - The International Society for Optical Engineering, vol. 6064 (2006)
4. Dabov, K., Foi, A., Katkovnik, V., Egiazarian, K.: Image denoising by sparse 3-D transform-domain collaborative filtering. IEEE Trans. Image Process. **16**(8), 2080–2095 (2007)
5. Dammertz, H., Sewtz, D., Hanika, J., Lensch, H.P.A.: Edge-avoiding a-trous wavelet transform for fast global illumination filtering. In: Doggett, M., Laine, S., Hunt, W. (eds.) High Performance Graphics. The Eurographics Association (2010)
6. Kalantari, N.K., Sen, P.: Removing the noise in Monte Carlo rendering with general image denoising algorithms. Comput. Graph. Forum **32**(2 Part 1), 93–102 (2013)
7. Katkovnik, V., Foi, A., Egiazarian, K., Astola, J.: From local kernel to nonlocal multiple-model image denoising. Int. J. Comput. Vis. **86**(1), 1–32 (2010)
8. Lebrun, M.: An analysis and implementation of the BM3D image denoising method. Image Process. On Line **2**, 175–213 (2012)
9. Pharr, M., Jakob, W., Humphreys, G.: Physically Based Rendering: From Theory to Implementation, 3rd edn, pp. 1–1233 (2016)
10. Rousselle, F., Knaus, C., Zwicker, M.: Adaptive sampling and reconstruction using greedy error minimization. ACM Trans. Graph. **30**(6), 159:1–159:12 (2011)
11. Sarjanoja, S., Boutellier, J., Hannuksela, J.: BM3D image denoising using heterogeneous computing platforms. In: Conference on Design and Architectures for Signal and Image Processing, DASIP, vol. 2015, December 2015