Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200098

An Implementation of Non-Local Means Algorithm on FPGA

Hayato KOIZUMI and Tsutomu MARUYAMA
Systems and Information Engineering, University of Tsukuba, 1-1-1 Tennoudai, Tsukuba, 305-8573, Japan
Email: maruyama@darwin.esys.tsukuba.ac.jp

Abstract. Non-Local means (NL-means) algorithm is a robust image denoising algorithm. Its computational complexity is, however, higher than other algorithms, and its availability is limited. In this paper, we propose an implementation method of the NL-means algorithm on FPGA. In the NL-means, the cross correlations between the small windows are repeatedly calculated, and a large number of intermediate data have to be held temporarily to reduce the amount of its computation. In our approach, the scan direction of the image is changed in the zigzag way. This zigzag scan increases the computation time because of the recalculation on the scan borders, but the required memory size can be drastically reduced. We have implemented the circuit on a Xilinx FPGA, and showed that with a small size FPGA, its real-time processing is possible.

1. Introduction

Noise reduction is the process of removing noise from an image. Non-Local means (NL-means) algorithm is one of the powerful and robust noise reduction algorithms[1]. A higher noise reduction ratio can be expected than Gaussian filter, Bilateral filter and so on, and it is supported in a major library[2]. However, its computational complexity is much higher than other denoising algorithms.

In NL-means algorithm, a search window is defined centered at the target pixel, and for each pixel in the search window, a template window is considered. Then, using the template windows, the cross-correlations between the target pixel and all pixels in the search window are calculated. These cross-correlations are used to improve the value of the target pixel. These cross-correlations can be efficiently calculated based on the calculation method of the box filter, and high performance can be easily achieved on FPGAs. However, for this efficient calculation method, large size memory is required, which means a large FPGA with large on-chip memory is required, though only a small amount of its logic cells are used.

We have proposed a memory efficient computation method of box filters[3]. This work demonstrated that the cross-correlation of the windows in two images (left and right images in the stereo vision) can be efficiently calculated with much less memory by changing the scan direction. In this approach, the image is scanned in zigzag, not from top-left to bottom-right. In [3], the processing speed was almost half of the top-left to bottom-right scan (it can be controlled by changing the required memory size), but it was still fast enough for real-time processing.

In this paper, we show that this approach works well for the calculation of the cross-correlations in one image using NL-means algorithm as an example. In the NL-means algorithm, the cross-correlations of the pixels in the square search window are calculated, though in the stereo vision, those of the pixels on a line segment along the x axis are calculated. This difference requires more line buffers, and more memory to store the temporary data. However, as shown in this paper, our approach works well also in this case, and the required memory can be reduced to 14% when the processing speed is half of the top-left to bottom-right scan.

2. Non-Local Means Algorithm

In the Non-Local means algorithm, given an image I, the denoised image I^{dn} is given as follow.

$$p^{dn} = \sum_{q \in W_S(p)} w(p,q) \cdot q \tag{1}$$

Here, p is a pixel in the image I, and p^{dn} is the denoised pixel of p. $W_S(p)$ is a window centered at p, called a search window of p, and q is a pixel in $W_S(p)$.

Let $W_T(p)$ be a window centered at p, called a template window of p. Then, $d(p,q)$, the difference between the two template windows, $W_T(p)$ and $W_T(q)$, is defined as follows.

$$d(p,q) = ||W_T(p) - W_T(q)||^2 \tag{2}$$

Using $d(p,q)$, $w(p,q)$ is given as

$$w(p,q) = \frac{1}{N(p)} \exp(-\frac{d(p,q)}{\sigma^2}) \tag{3}$$

where σ is a constant, and

$$N(p) = \sum_{r \in W_S(p)} \exp(-\frac{d(p,r)}{\sigma^2}) \tag{4}$$

$N(p)$ is used to normalize $w(p,q)$.

Let $p = I_{(x,y)}$, $q = I_{(u,v)}$, the size of the search window $(2w_s+1)^2$, and that of the template window $(2w_t+1)^2$. Then, equation (2) can be rewritten as follows.

$$d(I_{(x,y)}, I_{(u,v)}) = \sum_{dx=-w_t}^{w_t} \sum_{dy=-w_t}^{w_t} ||I_{(x+dx,y+dy)} - I_{(u+dx,v+dy)}||^2 \tag{5}$$

And, equation (1) can also be rewritten as

$$I^{dn}_{(x,y)} = \sum_{dx=-w_s}^{w_s} \sum_{dy=-w_s}^{w_s} w(I_{(x,y)}, I_{(x+dx,y+dy)}) \cdot I_{(x+dx,y+dy)} \tag{6}$$

Fig.1 shows the relation of the search window and the template window. In Fig.1, p, the target pixel, is the center of the search window, and for each pixel q in the search window, $d(p,q)$ is calculated using the pixels in their template windows.

The computational complexity of non-local means algorithm is given as follows.

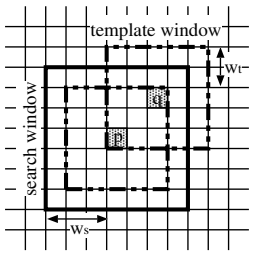

Figure 1. Search window and Template window

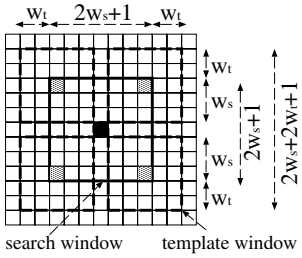

Figure 2. The Range of the Pixels Required for the Calculation

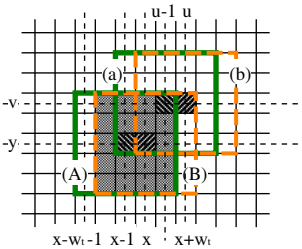

Figure 3. Overlapping of Template windows

1. The number of the pixels in the image is $W \times H$, where W and H are the width and height of the image.
2. For each pixel in the image, equation (6) is calculated. This means that equation (5) is calculated $(2w_s+1)^2$ times.
3. In each calculation of equation (5), $(2w_t+1)^2$ distances are calculated.

Thus, the computational complexity of this algorithm becomes $W \times H \times (2w_s+1)^2 \times (2w_t+1)^2$, which becomes very large for high resolution images.

3. Scan direction and the Performance

In this section, we compare the processing speed and the required memory size by the top-left to bottom-right scan and the zigzag scan[3]. Fig.2 shows how many pixels are necessary for calculating p^{dn} for pixel p. In Fig.2, the black pixel is the target pixel p, and the four gray pixels are the corner pixels of the search window. For calculating $d()$ for all pixels in the search window, $(2w_s+2w_t+1)^2$ pixels are required.

To simplify the discussion, we consider the calculation of only one $d(I_{(x,y)}, I_{(u,v)})$ though it is calculated for $(2w_s+1)^2$ pixels in the search window in the NL-means algorithm.

3.1. Top-left to bottom-right Scan

First, we describe an efficient calculation method of the NL-means algorithm when the image is scanned from top-left to bottom-right. This calculation method is widely used as the one for the box filter.

Suppose that $I^{dn}_{(x-1,y)}$ was calculated, and now $I^{dn}_{(x,y)}$ is going to be calculated. Here, we focus on the calculation of $d(I_{(x,y)}, I_{(u,v)})$ shown in Fig.3. In the calculation of $I^{dn}_{(x-1,y)}$, $d(I_{(x-1,y)}, I_{(u-1,v)})$ was calculated (Fig.3 (A) and (a)), and $d(I_{(x,y)}, I_{(u,v)})$ is going to be calculated for $I^{dn}_{(x,y)}$ (Fig.3 (B) and (b)). The differences of the gray pixels in (A) and (B) in Fig.3 can be shared in these calculations. Therefore, $d(I_{(x,y)}, I_{(u,v)})$ can be calculated from $d(I_{(x-1,y)}, I_{(u-1,v)})$ as follows.

$$\begin{aligned} d(I_{(x,y)}, I_{(u,v)}) =& d(I_{(x-1,y)}, I_{(u-1,v)}) + \\ & d_Y(I_{(x+w_t,y)}, I_{(u+w_t,v)}) - d_Y(I_{(x-w_t-1,y)}, I_{(u-w_t-1,v)}) \end{aligned} \quad (7)$$

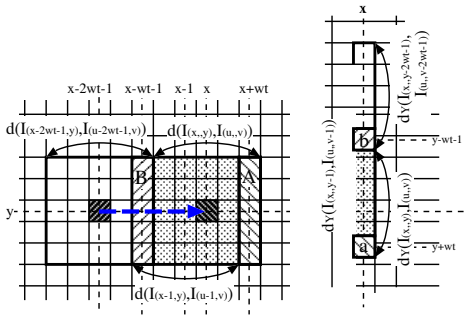

Figure 4. Calculation Method based on Box Filter

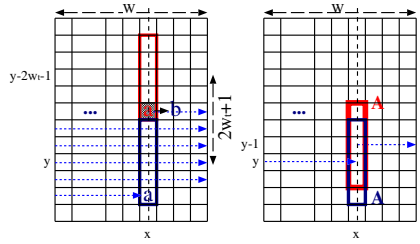

Figure 5. Memory usage in Top-left to Bottom-right Scan

where

$$d_Y(I_{(x,y)},I_{(u,v)}) = \sum_{dy=-w_t}^{w_t} ||I_{(x,y+dy)} - I_{(u,v+dy)}||^2 \tag{8}$$

Fig.4 illustrates this equation. By adding column A to $d(I_{(x-1,y)},I_{(u-1,v)})$, and subtracting column B, $d(I_{(x,y)},I_{(u,v)})$ can be obtained as shown in Fig.4-left. Column B was already calculated as column A of $d(I_{(x-2w_t-1,y)},I_{(u-2w_t-1,v)})$, and by keeping it for $2w_t+1$ steps (blue arrow in Fig.4-left, because the image is scanned from top-left to bottom-right), it can be reused as column B.

In the same way, $d_Y(I_{(x,y)},I_{(u,v)})$ can be calculated using the difference

$$\begin{aligned} d_Y(I_{(x,y)},I_{(u,v)}) =& d_Y(I_{(x,y-1)},I_{(u,v-1)}) + \\ & ||I_{(x,y+w_t)} - I_{(u,v+w_t)}||^2 - ||I_{(x,y-w_t-1)} - I_{(u,v-w_t-1)}||^2 \end{aligned} \tag{9}$$

Fig.4-right illustrates this calculation. By adding

$$a = ||I_{(x,y+w_t)} - I_{(u,v+w_t)}||^2,$$

to $d_Y(I_{(x,y-1)},I_{(u,v-1)})$, and subtracting

$$b = ||I_{(x,y-w_t-1)} - I_{(u,v-w_t-1)}||^2$$

$d_Y(I_{(x,y)},I_{(u,v)})$ can be obtained. Here, b was already calculated as a of $d_Y(I_{(x,y-2w_t-1)}, I_{(u,v-2w_t-1)})$, and by keeping it for $(2w_t+1)\times W$ steps, it is can be reused as b as shown in Fig.5-left. In Fig.5-left, a (the red one) was calculated for $d_Y(I_{(x,y-2w_t-1)},I_{(u,v-2w_t-1)})$, and by waiting $(2w_t+1)\times W$ steps, the focused pixel comes to $(x,y+w_t)$ by the top-left to bottom-right scan (the blue dotted arrows), and the a can be reused as b of $d(I_{(x,y)},I_{(u,v)})$. In this calculation method, $d_Y(I_{(x,y-1)},I_{(u,v-1)})$ (A in Fig.4) has to be also kept for W steps as shown in Fig.5-right.

With this calculation method, $d(I_{(x,y)},I_{(u,v)})$ can be obtained by calculating only a in Fig.4-right, if we can keep the following three values in the memory:

1. $d_Y(I_{(x+w_t,y)},I_{(u+w_t,v)})$ for $2w_t+1$ steps.
2. $b = ||I_{(x,y-2w_t-1)} - I_{(u,v-2w_t-1)}||^2$ for $(2w_t+1)\times W$ steps, and

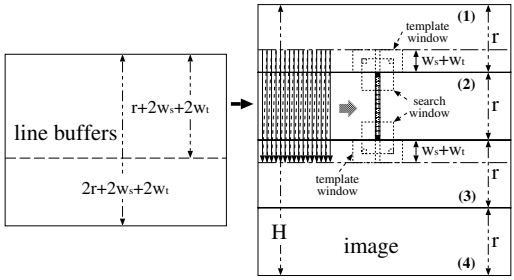

Figure 6. Zigzag scan

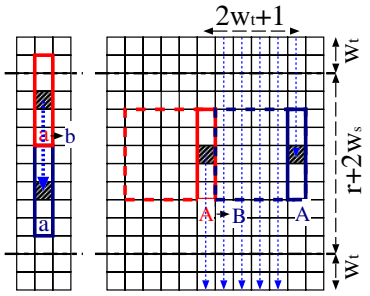

Figure 7. Memory usage in Zigzag scan

3. $d_Y(I_{(x,y-1)}, I_{(u,v-1)})$ for W steps.

In case of NL-means algorithm, these three values have to be kept for all pixels in the search window. Here, we ignore the first one, because its size is much smaller than the last two. The required memory size to keep these values becomes

$$(2w_s+1)^2 \times ((2w_t+1)+1) \times W. \tag{10}$$

In addition to this, $2w_s$ line buffers are necessary to supply $(2w_s+1)^2$ pixels in parallel. The size of these memory is proportional to the image width W, and larger search window (w_s) and template window (w_t) are required for higher resolution images. Therefore, this approach is not feasible for high resolution images, though it gives the minimum computational cost.

This computation method is widely used in many FPGA implementations, and has achieved very high performance, but it is difficult to process high resolution images even with the current largest FPGAs.

3.2. Zigzag scan

Fig.6 shows the outline of our zigzag scan. As shown in Fig.6, the image is divided vertically into blocks ((1),(2),(3) and (4) in Fig.6), the height of each is r. These blocks are processed sequentially from top to bottom. Each block is scanned in zigzag as shown in Fig.6 (in Fig.6, block (2) is being processed). The height of each block is r, but the scan width is $r+2w_s+2w_t$. This is to include all pixels that are necessary for the calculation of all template windows of the top and bottom pixels in the block (two black pixels in Fig.6).

This approach has one advantage, and two disadvantages. First, we describe the two disadvantages. As shown in Fig.6, the scan width for each block is $r+2w_s+2w_t$, but by this scan, $d()$ for only r pixels can be calculated. Thus, the computational efficiency becomes

$$\frac{r}{r+2w_s+2w_t} \tag{11}$$

which is apparently less than 1. Another disadvantage is the large memory size required for the line buffers as shown in Fig.6. To allow the zigzag scan of width $= r+2w_s+2w_t$, $r+2w_s+2w_t$ line buffers are required. In addition to this, r line buffers are necessary

to buffer the data that are necessary for the next block while processing the pixels in the current block. The total number of line buffers becomes $2r+2w_s+2w_t$.

The advantage of our approach is less memory size to hold the temporary data. In this scan method, $d_Y(I_{(x,y)}, I_{(u,v)})$ can also be calculated as shown in equation (9), but in this case, the pixels are scanned vertically, and b in Fig.4-right can be obtained by holding a only for $2w_t+1$ steps as shown in Fig.7-left, though it has to be held for $(2w_t+1)\times W$ steps in case of the top-left to bottom-right scan. To the contrary, for the computation of equation (7) shown in Fig.4-left, A has to be held for $(r+2w_s+2w_t)\times(2w_t+1)$ steps as shown in Fig.7-right, though it has to be held only $2w_t+1$ steps in case of the top-left to bottom-right scan. The main memory usage in this zigzag scan is given by (ignoring the ones used for $d_Y(I_{(x,y)}, I_{(u,v)})$)

$$(2w_s+1)^2\times(r+2w_s+2w_t)\times(2w_t+1). \tag{12}$$

This size is not proportional to W, which means that we can control the memory size by changing r, though the processing speed is also changed.

3.3. Comparison of the Processing Speed and Memory Size

The computational efficiently of the zigzag scan is given by equation (11). Its value can be controlled by changing r, though it also affect the required memory size. However, it is easy to keep this value larger than 0.5.

The ratio of the required memory size is given as follows.

$$\frac{(2w_s+1)^2\cdot(r+2w_s+2w_t)\cdot(2w_t+1)+(2r+2w_s+2w_s)\cdot W}{(2w_s+1)^2\cdot((2w_t+1)+1)\cdot W+2w_s\cdot W}$$

The numerator is the one for the zigzag scan; equation (12) and the line buffers, and denominator is the one for the top-left to bottom-right scan shown in equation (10) and its line buffers.

In our current implementation, $W=640$, $w_t=1$, $w_s=3$, and $r=8$. With these values of the parameters, the processing speed of our approach is 0.5 of the top-left to bottom-right scan, and the memory size ratio is 0.14. For larger W, consequently larger w_s and w_t, the processing speed is kept constant, but the memory size ratio becomes smaller, if r is $c\times w_s$ (c is a constant).

Table 1 compares the required memory size and the processing speed of the zigzag scan to those of the top-left to bottom-right scan when r is changed under the parameters given above. As shown in this table, by changing r, the required memory size can be changed in wide range. This makes it possible to choose the minimum size FPGA for the required processing speed, and also to achieve the maximum performance on the given FPGA by choosing proper r. To the contrary, in the top-left to bottom-right scan, the memory size cannot be reduced for any processing speed.

In both scan methods, by processing n pixels sequentially on the same unit, the logic cells can be reduced to $1/n$. With this sequential approach, the processing speed is also decreased to $1/n$, but the memory size cannot be reduced in both scan methods. However, in the zigzag scan, the required memory size is much smaller, and distributed RAMs can also be used, though only block RAMs can be used in the top-left to bottom-right scan because the memory depth must be W. This flexibility enables to achieve better performance on wide range of FPGAs.

Table 1. Memory usage and computational efficiency

r	4	8	16	32	64
memory size	0.09	0.14	0.23	0.40	0.75
processing speed	0.33	0.50	0.67	0.80	0.80

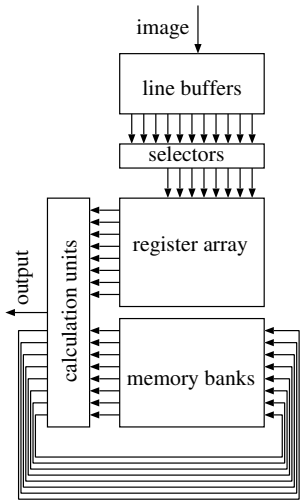

Figure 8. A Block Diagram

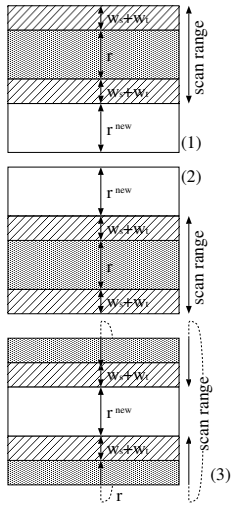

Figure 9. Line Buffers

4. An Implementation on FPGA

In the NL-means algorithm, equation (5) is calculated for all $(w_s+1)^2$ pixels in the search window. The processing speed can be controlled by changing the number of pixels that are processed in parallel. By processing all pixels in parallel, the maximum performance can be achieved. In our implementation, w_s+1 pixels are processed in parallel, and w_s+1 clock cycles are used to process $(w_s+1)^2$ pixels. This approach is taken to reduce the circuit size as much as possible while keeping the processing of 640×480 pixel images faster than 30 fps.

4.1. Block Diagram

Fig.8 shows a block diagram of our system. First, the data are sent from the host computer, and they are once stored in the line buffers for the zigzag scan. Then, the data in the line buffers are read onto the register array through selectors. The equations described in Section 3 are calculated using the values on the register array, and those from the memory banks. In the memory banks, the values discussed in Section 3.2 are stored.

4.2. Line buffers and Register Array

Fig.9 shows the usage of the line buffers in our approach. In Fig.9(1), $w_s+w_t+r+w_s+t+w_t$ line buffers from the top are used for the current zigzag scan, and for the pixels in r lines, NL-means algorithm is applied. This phase takes more than $W\times r$ clock cycles

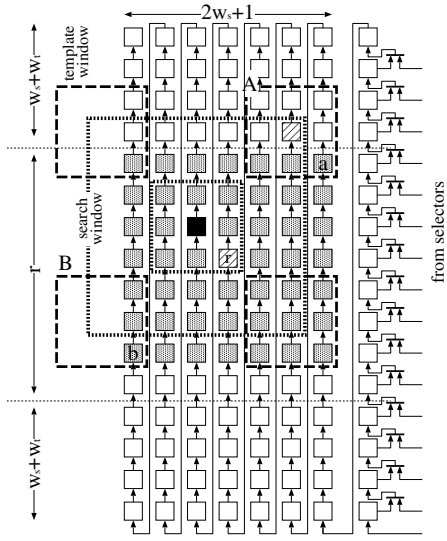

Figure 10. Register Array

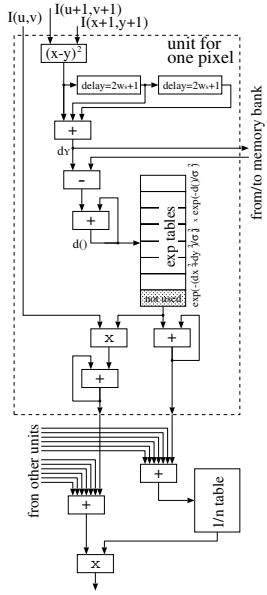

Figure 11. Calculation Units

even if all pixels in the search window are calculated in parallel. During this period, the pixels of the next r lines are read into the line buffers r^{new} for the next zigzag scan. Then, the next $w_s + w_t + r + w_s + t + w_t$ line buffers are used for the next zigzag scan as shown in Fig.9(2), and the next r lines are read into the next line buffers r^{new} . In our current implementation, $r = 8$, $w_s = 3$ and $w_t = 1$. Therefore, as shown in Fig.9, by repeating three phases ((1),(2) and (3)), all lines in the image can be processed.

Fig.10 shows the register array. The data in the line buffers are fetched onto the register array through selectors. As described above, only three phases are necessary for the assignment of the line buffers, which means that only 3-to-1 selectors are required. The register array consists of $(r + 2w_t + 2w_s) \times (2w_t + 1)$ registers. In Fig.10, the black pixel is the center of the search window. Then, the pixel r, the bottom-right corner pixel of its template window, is compared with $(2w_s + 1)^2$ pixels, the gray ones in the figure. As shown in Fig.10, only the values on gray registers are used for the calculations, and other registers are used only for keeping the data. The registers in the right-most column are used to get the data from the line buffers, and to give them to the register array.

4.3. Memory Banks

As described in Section 3.2, two kinds of memory banks are required. The first one is to store a in Fig.7, and the another is to store A. The required depth for the first one is $2w_t + 1$. In our current implementation, $w_t = 1$. Thus, a is held on the registers not in the memory.

The required memory size for the second one is $(2w_s + 1)^2 \times (r + 2w_s + 2w_t) \times (2w_t + 1)$. As described above, in our current implementation, $(2w_s + 1)^2$ pixels in the search window are processed in $2w_s + 1$ clock cycles by processing $2w_s + 1$ pixels in parallel. Thus, the required number of memory banks is $2w_s + 1$. In this case, for each bank, the temporary data for $2w_s + 1$ pixel can be stored. This means that the depth of

the memory banks is $(2w_s+1)\times(r+2w_s+2w_t)\times(2w_t+1)$. This depth becomes 336 under the current parameters. Block RAMs configured as 512×72 can be used as this memory.

4.4. Calculation Units

In our implementation, $2w_s+1$ units are used, and each unit processes $2w_s+1$ pixels sequentially using $2w_s+1$ clock cycles. Fig.11 shows a block diagram of one unit for processing one of $2w_s+1$ pixels. First, $I_{(x+1,y+1)}$ and $I_{(u+1,v+1)}$ are given ($w_t=1$), and their distance is calculated. In Fig.11, the circuit only for one channel is shown, though three channels for R, G and B are processed. Then, it is delayed for $2w_s+1$ clock cycles twice (distributed RAMs are used), and three values are added to calculate d_Y. In the discussion in Section 3, d_Y is calculated using the difference as shown in equation (9), but in the current implementation, $2w_t+1=3$, and the three values are directly added. It is sent to the memory banks (as A in Fig.7), and another one is read back from the memory banks (B in Fig.7). Their difference is added to the register that holds $d()$ to obtain its new value following equation (7). Then, one of the tables that hold $exp(-\frac{dx^2+dy^2}{\sigma_d^2})\cdot exp(-\frac{d()}{\sigma^2})$ is looked up using $d()$ (σ_d is a constant). The first term is a weight considering the distance from the center of the search window. This weight is not shown in equitation (3), but used in our current implementation. $7=2w_s+1$ tables are packed into one block RAM, and one of them is accessed according to the distance from the center pixel. The size of each table is 256, which is large enough to obtain our target PSNR. The output, and the product of the output and $I_{(u,v)}$ are accumulated respectively. Then, those values from $2w_s+1$ units are added. The reciprocal of the sum of the output is obtained by table look-up, and the final output obtained by multiplying them.

5. Experimental Results

We have implemented the circuit on Xilinx FPGA Kintex-7 XC7K160T. For this implementation, 24.6K LUTs (24.3%), 63 block RAMs (19.4%) and 87 DSP slices (14.5%) were used. The size of this circuit is small enough. Its operational frequency is 335.4MHz, and its processing speed is 78.0 fps for 640×480 pixel images. This processing speed is 408X of the software on Core i7-860 2.8GHz.

Fig.12 shows the input image, the image in which noise is added, the output by the original NL-means algorithm, and that by our system. The PSNR by the original algorithm is 31.7dB, and that by our system is 28.0 dB. The PSNR by ours is a bit worse, but it is higher than 25dB, and as reported in [6][7], visually, it is difficult to find the difference. The operation data width is reduced to keep the PSNR higher than 25.0dB, not to achieve the PSNR of the original algorithm. This is to reduce the circuit size as much as possible while keeping the enough quality for human recognition.

6. Conclusions

We have implemented a circuit for Non-Local means algorithm on FPGA. To reduce the memory size, the image is scanned in zigzag. With this scan method, the memory size can be reduced to 14% of the top-left to bottom-right scan. Its processing speed becomes

Figure 12. Comparison of the output images

half of the top-left to bottom-right scan, but it is still 78 fps for 640×480pixel images, which is fast enough for real-time processing.

The design based on this zigzag scan requires more effort than that for the top-left to bottom-right scan. To develop a library to make it easier is one of main future work.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 18K11209, and the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] Buades, Antoni, Bartomeu Coll, and J-M. Morel, A non-local algorithm for image denoising, CVPR, 2005.
[2] https://docs.opencv.org/3.2.0/d5/d69/tutorial_py_non_local_means.html.
[3] W. Sichao, and T. Maruyama, An implementation method of the box filter on FPGA, FPL, 2016.
[4] Jin Wang, Yanwen Guo, Yiting Ying, Yanli Liu and Qunsheng Peng, Fast non-local algorithm for image denoising, International Conference on Image Processing. 2006.
[5] L. L. Gambarra, J. C. Pessoa et. al., Fast non-local image denoising using a hardware implementation, Workshop on Circuits and System Design, 2012.
[6] Thomos, Nikolaos, Nikolaos V. Boulgouris, and Michael G. Strintzis, Optimized transmission of JPEG2000 streams over wireless channels, IEEE Transactions on image processing 15.1, 54-67. 2006.
[7] Li, Xiangjun, and Jianfei Cai, Robust transmission of JPEG2000 encoded images over packet loss channels, IEEE International Conference on Multimedia and Expo, 2007.