On-Orbit Real-Time Variational Image Destriping: FPGA Architecture and Implementation

Liqun Chen, Yi Chang, *Member, IEEE*, and Luxin Yan, *Member, IEEE*

***Abstract*— On-orbit real-time image processing is of increasing demands due to requirements for quick-response missions. Image destriping is usually an important pre-processing step to improve image quality in practice. The unidirectional variational models have shown impressive destriping performance. However, they are not easy for real-time implementation for high-computation complexity and there are few attempts for hardware implementation. This article is the first hardware implementation of variational image destriping algorithm, which achieves high throughput for large-swath remote-sensing images. In this article, a fully pipelined hardware architecture is proposed. First, the involved iteration loop is unrolled and a coarse-grained parallelism is obtained. Second, for each deployed iteration computation blocks (ICBs), a dedicated timing arrangement is designed to alleviate the bottleneck caused by the data dependency within each ICB, obtaining a fine-grained parallelism. Moreover, to further optimize the critical path, an approximate simplification scheme is proposed, saving the resource usage and reducing computing delay. The proposed architecture is implemented and verified on a XILINX 6vcx240t field programmable gate array (FPGA); it achieves a maximum frame rate up to 41.9 frames/s with delay of only tens of row cycles for 8-bit 2048 × 2048 images. It performs all the processing on the pixels in raster scan order on-the-fly as they are being transmitted from camera payload, which significantly facilitates on-orbit real-time processing of large-swath remote-sensing images with high data rate.**

***Index Terms*— Field programmable gate array (FPGA) implementation, fully pipelined architecture, image destriping, on orbit, split Bregman method.**

I. Introduction

NOWADAYS, the significant technological progresses in satellite imaging instruments lead to a continual increasing of the spatial, spectral, temporal, and radiometric resolutions of the remote-sensing images. Therefore, the limited downlink capability makes the timely missions, such as maritime search, disaster monitoring, and military surveillance [1]–[8], should be performed on orbit in real time. In practice, the stripe noise commonly exists in remote-sensing images and severely degrades the image quality [9]–[12], limiting the performance of the further processing, so it is critical to suppress the stripes before subsequent processings.

Manuscript received August 18, 2021; revised December 10, 2021 and December 31, 2021; accepted December 31, 2021. Date of publication January 5, 2022; date of current version March 8, 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 61971460. *(Corresponding author: Luxin Yan.)*

The authors are with the National Key Laboratory of Science and Technology on Multispectral Information Processing, School of Automation, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: yanluxin@gmail.com).

Digital Object Identifier 10.1109/TGRS.2022.3140428

In past decades, many efforts have been made for remote-sensing image destriping. Most of them are devoted to destriping algorithms, but related works on real-time image destriping are rarely reported. This work mainly focuses on implementation of high-performance on-orbit real-time image destriping, more specifically, it is the first hardware implementation for variational model-based method by far.

The image destriping methods can be roughly classified into four categories: digital filtering technique-based methods [13]–[16], matching-based methods [17]–[20], variational model-based methods [21]–[31], and deep learning-based methods [32]–[36]. The first two categories are of relatively low-computation complexity and easy for hardware implementation, but their destriping performances are very limited. The third category of methods has obtained excellent destriping performances, outperforming the former two categories in both qualitative and quantitative evaluations, but at the cost of higher computation load. The fourth category of methods could obtain competitive destriping performances with the help of big data learning. Since the available on-orbit resources are constrained in terms of size, weight, and power, deep learning-based methods at present are not appropriate options for their high overhead of hardware resources and power. In this work, considering the trade-off between the destriping performance and computation load, we implement a variational model-based method to cope with on-orbit single-image destriping task, not including hyperspectral or multispectral images.

However, the variational model-based methods usually need repetitive iterations to converge to the optimal solution, which are of high-computational complexity. For instance, the gradient descent optimization was used to obtain the solution in [23], and split Bregman iteration method was used in [25] to achieve faster convergence to solve the minimization problem. Running C++ simulation experiments with the method in [25] on a 3.2 GHz Intel Core i7-8700 with six cores and 16 GB of RAM to destripe a MODIS image in size of 512 × 512, the runtime is about 2.4 s. It indicates that the variational methods are performed far from real time even using a modern high-performance general-purpose CPU on the ground, let alone the resource-limited computing platform on orbit. This is because these variational methods need numbers of repetitive iterations, and exhibit strong data dependencies within the computation procedures. With modern multi-core CPUs, the iterative algorithms can only be implemented in coarse-grained parallelism, each core executes operations serially, and necessary data interaction between cores will cause high latency, degrading the performance. Some works [37], [38] were devoted to exploit the potential parallelism in iterative algorithms, and designed parallel hardware architectures on field programmable gate array (FPGA) platforms by taking advantages of its abundant parallel resources. Meanwhile, FPGA platforms consume much less power than CPU or GPU platforms, and are widely applied on orbit. Therefore, we are motivated to exploit the intrinsic parallelism of the variational model-based methods, and propose a high-throughput, low-delay, and fully pipelined hardware architecture for real-time image destriping with a unidirectional variational model. The main ideas and contributions are threefold.

1) To the best of our knowledge, it is the first high-performance hardware implementation of variational image destriping method for large-swath remote-sensing images with high data rate. It executes all the processings on the pixels in raster scan order on-the-fly as they are being transmitted from camera payload, achieving a throughput up to 176 MPixels/s with only tens of row cycles delay in XILINX 6vcx240t FPGA platforms, which significantly facilitates on-orbit real-time applications.
2) A fully pipelined iteration unrolled hardware architecture with multiple cascaded iteration computation blocks (ICBs) for split Bregman method is proposed, obtaining both coarse- and fine-grained parallelisms, and alleviating the inherent data dependencies between and within iterations. Consequently, it greatly improves the overall throughput, eliminates huge frame buffers usage and thus decreases processing delay (PD).
3) A dedicated timing arrangement and an approximate simplification scheme for one single ICB are proposed for pipelined implementation, which jointly improve the processing speed as well as decreasing the resource consumptions. As a result, for the proposed iteration unrolled architecture with multiple cascaded ICBs, the overall throughput is further improved and the resource consumptions are reduced in practice.

The rest of this article is organized as follows. Section II introduces the unidirectional destriping method and the parallelism analysis. Section III discusses the parameters determination and the finite precision effects. The hardware architecture is presented in Section IV. The experimental results of the proposed design are provided in Section V. Finally, the conclusion is given in Section VI.

II. Unidirectional Variational Destriping Method and Parallelism Analysis

A. Unidirectional Variational Destriping Method

Different from the random noise, stripe noise exhibits noteworthy directional characteristic. Unidirectional variational model effectively exploits this characteristic and achieves impressive destriping results [25], [26]. In this work, we consider vertical stripes and a destriping model with unidirectional variational as

$$\min_{\boldsymbol{u}} \frac{1}{2}\|\boldsymbol{u}-\boldsymbol{I}_s\|_2^2 + \lambda_1\left\|\nabla_y \boldsymbol{u}\right\|_1 + \lambda_2\|\nabla_x(\boldsymbol{u}-\boldsymbol{I}_s)\|_1 \tag{1}$$

where $\boldsymbol{u}$ denotes the desired clear image, $\boldsymbol{I}_s$ stands for the striped image, and the symbols ∇_y and ∇_x are the derivative operators across and along the stripes, respectively. The second and third terms are the unidirectional variational regularization terms, penalizing the ℓ_1-norm of the gradient across the stripes, and constraining the ℓ_1-norm of the difference between the gradients along the stripes of the clear and striped images. The parameters λ_1 and λ_2 are the regularization coefficients. The model (1) is an improved version of the model in [23], in which the first data fidelity term is introduced to preserve the image intensities. Moreover, the model (1) is a simplified version of our previous model in [25]. In comparison, the framelet regularization term is withdrawn for lowering the computational complexity at the cost of slight performance decrease (about 1–2 dB loss in PSNR [25]).

B. Numerical Algorithm

As with the work in [25], we use split Bregman method [39] to solve ℓ_1-norm regularizations due to its two advantages. First, it avoids the nondifferentiable points especially for the ℓ_1-norm cases. Second, it can converge quickly with less memory usage, which makes it attractive for large-swath remote-sensing images.

The main idea of split Bregman method is to convert the unconstrained minimization problem on $\boldsymbol{u}$ by introducing two auxiliary variables $\boldsymbol{d}_y = \nabla_y\boldsymbol{u}$ and $\boldsymbol{d}_x = \nabla_x(\boldsymbol{u}-\boldsymbol{I}_s)$. The minimization of (1) is equivalent to the constrained problem

$$\begin{aligned}&\min_{\boldsymbol{u}} \frac{1}{2}\|\boldsymbol{u}-\boldsymbol{I}_s\|_2^2 + \lambda_1\left\|\nabla_y \boldsymbol{u}\right\|_1 + \lambda_2\|\nabla_x(\boldsymbol{u}-\boldsymbol{I}_s)\|_1\\ &\text{s.t. } \boldsymbol{d}_y = \nabla_y\boldsymbol{u}, \quad \boldsymbol{d}_x = \nabla_x(\boldsymbol{u}-\boldsymbol{I}_s).\end{aligned} \tag{2}$$

By applying Bregman iteration, the problem (2) can be further transformed into an unconstrained minimization

$$\begin{aligned}\min_{\boldsymbol{u},\boldsymbol{d}_x,\boldsymbol{d}_y} &\frac{1}{2}\|\boldsymbol{u}-\boldsymbol{I}_s\|_2^2 + \lambda_1\left\|\boldsymbol{d}_y\right\|_1 + \lambda_2\|\boldsymbol{d}_x\|_1\\ &+\frac{\alpha}{2}\left\|\boldsymbol{d}_y-\nabla_y\boldsymbol{u}-\boldsymbol{b}_y\right\|_2^2 + \frac{\beta}{2}\|\boldsymbol{d}_x-\nabla_x(\boldsymbol{u}-\boldsymbol{I}_s)-\boldsymbol{b}_x\|_2^2\end{aligned} \tag{3}$$

where α and β are Bregman penalization parameters, and the variable $\boldsymbol{b}_x$, $\boldsymbol{b}_y$ are determined via Bregman iteration. Obviously, the minimization problem (3) can be further converted to three subproblems.

1) The $\boldsymbol{u}$-related subproblem is

$$\begin{aligned}\min_{\boldsymbol{u}} &\frac{1}{2}\|\boldsymbol{u}-\boldsymbol{I}_s\|_2^2 + \frac{\alpha}{2}\left\|\boldsymbol{d}_y-\nabla_y\boldsymbol{u}-\boldsymbol{b}_y\right\|_2^2\\ &+\frac{\beta}{2}\|\boldsymbol{d}_x-\nabla_x(\boldsymbol{u}-\boldsymbol{I}_s)-\boldsymbol{b}_x\|_2^2.\end{aligned} \tag{4}$$

Equation (4) is a convex function with three ℓ_2-norm terms, thus we can directly employ the differentiation with respect to $\boldsymbol{u}$. Then the minimization leads to the following equation:

$$\begin{aligned}(\boldsymbol{u}-\boldsymbol{I}_s) &+ \alpha\left(\nabla_y^T\nabla_y\boldsymbol{u} - \nabla_y^T\left(\boldsymbol{d}_y-\boldsymbol{b}_y\right)\right)\\ &+\beta\left(\nabla_x^T\nabla_x\boldsymbol{u} - \nabla_x^T(\boldsymbol{d}_x-\boldsymbol{b}_x+\nabla_x\boldsymbol{I}_s)\right) = 0\end{aligned} \tag{5}$$

then it can be transformed into

$$\begin{aligned}&\left(\boldsymbol{I}+\alpha\nabla_y^T\nabla_y+\beta\nabla_x^T\nabla_x\right)\boldsymbol{u}\\ &\quad= \boldsymbol{I}_s + \alpha\nabla_y^T\left(\boldsymbol{d}_y-\boldsymbol{b}_y\right) + \beta\nabla_x^T(\boldsymbol{d}_x-\boldsymbol{b}_x+\nabla_x\boldsymbol{I}_s)\end{aligned} \tag{6}$$

where ∇_y^T and ∇_x^T are the transposition operators of ∇_y and ∇_x, respectively. $\boldsymbol{I}$ denotes an identity matrix with the same size as $\boldsymbol{u}$. Equation (6) can be solved with a closed-form solution by the fast Fourier transform (FFT), but it requires all the elements of an image to be involved in the transforming computation, leading to huge memory usage and computation delay. Therefore, FFT is not appropriate for hardware real-time implementation. Here we choose direct discretization to solve $\boldsymbol{u}$-related subproblem. Assuming an image to be a 2-D vector of size $M \times N$ and stripes as vertical, the gradient computation can be given as backward difference by

$$\nabla_x \boldsymbol{u}(i,j) = \begin{cases} \boldsymbol{u}(i,j) - \boldsymbol{u}(i-1,j), & i = 2,3,\ldots,M \\ 0, & i = 1 \end{cases} \tag{7}$$

$$\nabla_y \boldsymbol{u}(i,j) = \begin{cases} u(i,j) - \boldsymbol{u}(i,j-1), & j = 2,3,\ldots,N \\ 0, & j = 1. \end{cases} \tag{8}$$

And their corresponding transposition operators ∇_y^T and ∇_x^T can be given as forward difference by

$$\nabla_x^T \boldsymbol{u}(i,j) = \begin{cases} \boldsymbol{u}(i,j) - \boldsymbol{u}(i+1,j), & i = 1,2,\ldots,M-1 \\ 0, & i = M \end{cases} \tag{9}$$

$$\nabla_y^T \boldsymbol{u}(i,j) = \begin{cases} \boldsymbol{u}(i,j) - \boldsymbol{u}(i,j+1), & j = 1,2,\ldots,N-1 \\ 0, & j = N. \end{cases} \tag{10}$$

The second-order difference operators $\nabla_x^T \nabla_x$ and $\nabla_y^T \nabla_y$ can be given by

$$\nabla_x^T \nabla_x \boldsymbol{u}(i,j) = \begin{cases} 2\boldsymbol{u}(i,j) - \boldsymbol{u}(i+1,j) - \boldsymbol{u}(i-1,j), \\ \quad i = 2,\ldots,M-1 \\ 0, \quad i = 1,M \end{cases} \tag{11}$$

$$\nabla_y^T \nabla_y \boldsymbol{u}(i,j) = \begin{cases} 2\boldsymbol{u}(i,j) - \boldsymbol{u}(i,j+1) - \boldsymbol{u}(i,j-1), \\ \quad i = 2,\ldots,N-1 \\ 0, \quad i = 1,N. \end{cases} \tag{12}$$

Finally, with (7)–(12), (6) can be discretized to the formulation as

$$\begin{aligned} &\boldsymbol{u}(i,j)(1+2\alpha+2\beta) \\ &= \alpha\big(\boldsymbol{u}(i,j-1) + \boldsymbol{u}(i,j+1) + \boldsymbol{d}_y(i,j) \\ &\quad - \boldsymbol{d}_y(i,j+1) - \boldsymbol{b}_y(i,j) + \boldsymbol{b}_y(i,j+1)\big) + \beta(\boldsymbol{u}(i-1,j) \\ &\quad + \boldsymbol{u}(i+1,j) + \boldsymbol{d}_x(i,j) - \boldsymbol{d}_x(i+1,j) - \boldsymbol{b}_x(i,j) + \boldsymbol{b}_x(i+1,j) \\ &\quad - \boldsymbol{I}_s(i-1,j) - \boldsymbol{I}_s(i+1,j)) + \boldsymbol{I}_s(i,j)(1+2\beta). \end{aligned} \tag{13}$$

2) The $\boldsymbol{d}_x$-related subproblem is

$$\min_{\boldsymbol{d}_x} \lambda_2 \|\boldsymbol{d}_x\|_1 + \frac{\beta}{2}\|\boldsymbol{d}_x - \nabla_x(\boldsymbol{u} - \boldsymbol{I}_s) - \boldsymbol{b}_x\|_2^2 \tag{14}$$

which can be solved by a shrinkage operator as

$$\boldsymbol{d}_x = \text{shrink}\left(\nabla_x(\boldsymbol{u} - \boldsymbol{I}_s) + \boldsymbol{b}_x, \frac{\lambda_2}{\beta}\right) \tag{15}$$

where

$$\text{shrink}(r,\xi) = \frac{r}{|r|} * \max(r - \xi, 0). \tag{16}$$

Algorithm 1 Unidirectional Variational Image Destriping

```
Input data $\boldsymbol{I}_s$, parameters $\lambda_1$,$\lambda_2$,$\alpha$,and $\beta$.
Initialize $\boldsymbol{u}_0 = \boldsymbol{I}_s$, $\boldsymbol{d}_x = 0$, $\boldsymbol{d}_y = 0$, $\varepsilon = 10^{-3}$.
  while ($\|\boldsymbol{u}^k - \boldsymbol{u}^{k-1}\| / \|\boldsymbol{u}^k\| > \varepsilon$ and $k < N_{\max}$) do
    1. Solve $\boldsymbol{u}^k$ by (13)
    2. Update $\boldsymbol{d}_x^k$, $\boldsymbol{d}_y^k$ by (15) and (17)
    3. Compute $\boldsymbol{b}_x^k$, $\boldsymbol{b}_y^k$ by (18)
  end while
Output Destriped Image=$\boldsymbol{u}^k$
```

The shrinkage operator is a soft threshold method proposed by [40], which is fast and needs only a few operations during the update of $\boldsymbol{d}_x$. Similarily, the $\boldsymbol{d}_y$-related subproblem can also be addressed by a shrinkage operator

$$\boldsymbol{d}_y = \text{shrink}\left(\nabla_y \boldsymbol{u} + \boldsymbol{b}_y, \frac{\lambda_1}{\alpha}\right). \tag{17}$$

Finally, the Bregman variables can be updated in the following way:

$$\begin{cases} \boldsymbol{b}_x^k = \boldsymbol{b}_x^{k-1} + \left(\nabla_x\left(\boldsymbol{u}^k - \boldsymbol{I}_s\right) - \boldsymbol{d}_x^k\right) \\ \boldsymbol{b}_y^k = \boldsymbol{b}_y^{k-1} + \left(\nabla_y \boldsymbol{u}^k - \boldsymbol{d}_y^k\right). \end{cases} \tag{18}$$

With the equations above, the complete destriping procedure can be summarized as listed in section entitled Algorithm 1, in which the three subproblems are solved alternately. Using Gauss–Seidel iteration, faster convergence can be achieved to solve the three subproblems when compared with Jacobi iterative method [26], [41]–[43].

C. Data Dependencies and Parallelism Analysis

To exploit the intrinsic parallelism of the destriping method illustrated in Algorithm 1, we have comprehensively analyzed every procedure throughout the numerical solution, and two levels of iterations have been explored: 1) the outermost loop which repeatedly performs a series of computations on the input/intermediate matrices and 2) the dedicated computations aiming at each single pixel and traversing every element in the input/intermediate matrix. Therefore, we can fully exploit the spatial parallelism potential inside the employed destriping approach with the two characteristics above. First, it is noticeable that the input and output of each iteration in the outermost loop are all matrices of same size, and each iteration receives the output matrix from the previous iteration as their input, thus we can unroll all these iterations and instantiate them in cascade to obtain a coarse-grained parallelism. Second, in each iteration, the three steps of the split Bregman method all require only a portion of the input data matrix, moreover, the relative positions of the elements involved in the computation of any (x,y) element do not change across the input data matrix. Thus, we can instantiate the three computational steps in each iteration only for once to compute the elements of the input matrix sequentially. Thirdly, the dependency relations among the three computational steps indicate that, the step of solving $\boldsymbol{d}_x^k$, $\boldsymbol{d}_y^k$ relies on the updated result of the step of solving $\boldsymbol{u}^k$, and the step of solving $\boldsymbol{b}_x^k$, $\boldsymbol{b}_y^k$ also requires the updated results from the step of solving $\boldsymbol{d}_x^k$, $\boldsymbol{d}_y^k$. Thus, this feature suggests an implementation in fully pipelined manner to achieve a fine-grained parallelism.

III. Parameters, Iterations, and Finite Precision Effects

A. Parameter Choice

In the numerical algorithm above, there are four parameters λ_1, λ_2, α, and β to be predefined. Obtaining optimal values for them is not straightforward. We select the appropriate values for parameters through simulated experiments with respect to 8-bit images. To obtain simulated striped images, a set of 256×256 clear original remote-sensing images were chosen and added with different levels of stripe lines. Three discontinuous stripe lines per ten lines with magnitude ranging from 2 to 7 were added and treated as weak stripes, five successive stripe lines per ten lines ranging from 2 to 13 were treated as medium stripes, and eight successive strip lines per ten lines from 2 to 25 as severe stripes. For assessing the quality of destriped images, peak signal-to-noise (PSNR) is used

$$\mathrm{PSNR} = 10\log_{10}\left(\frac{\mathrm{MAX}^2}{\|\boldsymbol{u} - \boldsymbol{I}_s\|^2}\right) \tag{19}$$

where MAX is the maximum value of the image pixels. Higher value of PSNR means better image quality.

The experiments with respect to PSNR varying λ_1, λ_2, α, and β on images with different stripe levels have been carried out, in which the value of one parameter was changed while other three were fixed. The experiment results are shown in Fig. 1, in the case of weak stripes, the PSNR stabilizes when $\lambda_1 \in [10, 30]$, $\lambda_2 \in [245, 700]$, $\alpha \in [0, 30]$, and $\beta \in [100, 200]$, so optimal values can be selected from these intervals. In the case of medium stripes, the range of optimal parameters are chosen as $\lambda_1 \in [30, 90]$, $\lambda_2 \in [500, 1400]$, $\alpha \in [0, 15]$, and $\beta \in [80, 200]$. In the case of severe stripes, the optimal intervals are $\lambda_1 \in [120, 180]$, $\lambda_2 \in [960, 1600]$, $\alpha \in [5, 20]$ and $\beta \in [110, 200]$. To reduce the computational complexity and overhead in the hardware implementation, it is natural to select the parameters λ_1, λ_2, α, and β as power of 2. So we set regularization parameters $\lambda_1 = 16$, $\lambda_2 = 256$ for weak-striped images, $\lambda_1 = 64$, $\lambda_2 = 512$ for medium-striped images, and $\lambda_1 = 128$, $\lambda_2 = 1024$ for severe-striped images. As for Bregman penalization parameters, we set $\alpha = 16$ and $\beta = 128$ for all three levels of stripes.

Benefited from the usage of split Bregman method and Gauss–Seidel iteration, a faster convergence to solution can be achieved, making it easier to unroll the iterations. For hardware implementation, the number of iterations should be fixed. Therefore, we have carried out the experiments on the number of iterations versus the mean square error (MSE), which is the mean square error between the images obtained from two adjacent iterations. The results are shown in Fig. 2(a). For the weak and medium stripe cases, the value of MSE tends to stabilize after about 10 and 15 iterations, respectively. As for the severe stripe cases, the minimum optimal number of iterations indicates 20.

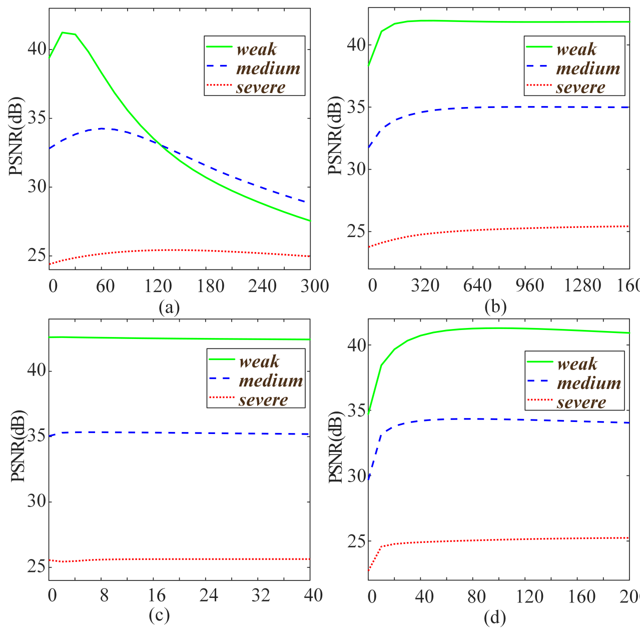


Fig. 1. (a) PSNR versus parameter λ_1. (b) PSNR versus parameter λ_1. (c) PSNR versus parameter α. (d) PSNR versus parameter β.

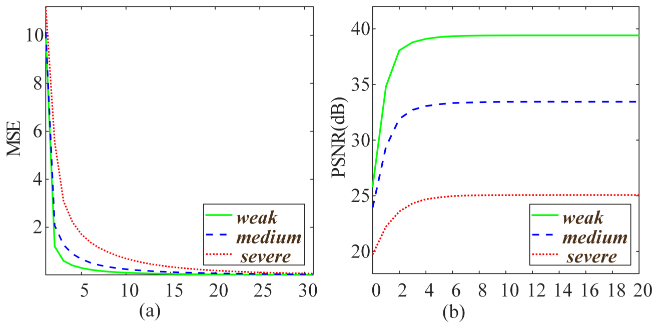


Fig. 2. (a) MSE versus number of iterations. (b) PSNR difference between the floating-point reference model and finite precision model versus number of bits for the fractional part (m).

To verify the performance of our method with fixed parameters and iteration numbers, we have carried out real experiments on actual images, including Terra MODIS level 1 B data, the Hyperspectral Digital Imagery Collection Experiment (HYDICE) image of the Washington DC Mall, real infrared image, and HYDICE Urban band 1 data. The experiment results are shown in Fig. 3, we can see that different levels of stripes images are all well removed.

B. Finite Precision Effect

On the one hand, the variables in the software implementation are represented as double-precision float points, which is not economic for hardware implementation given the high consumption of the hardware resources. On the other hand, the number of bits representing the variables has great influence on the quality of destriped images. Therefore, a tradeoff between the stripe removal performance and the hardware resource expense should be achieved. For 8-bit images, the integer part

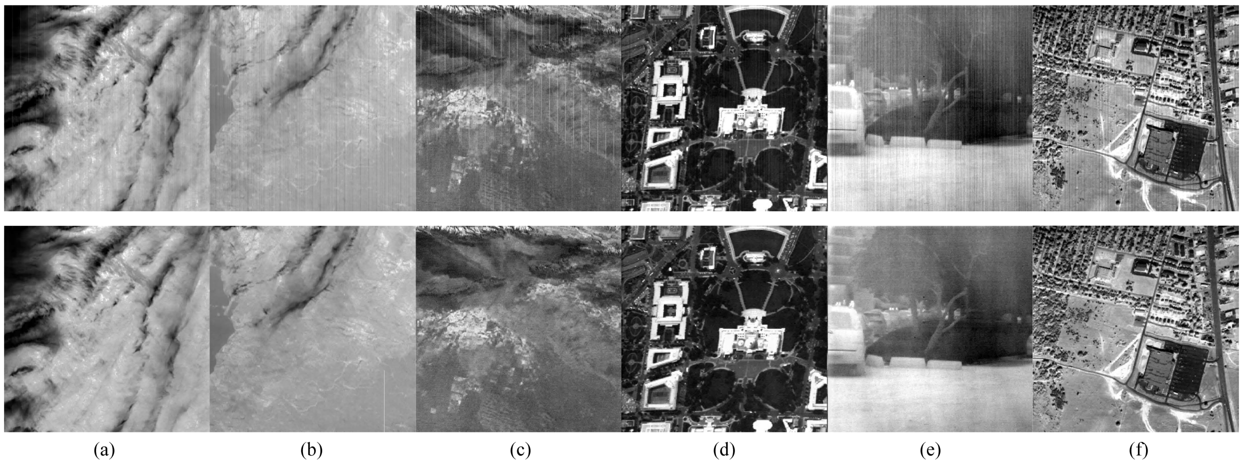


Fig. 3. Experiments on real images. The top row are the actual striped images, the bottom row are the corresponding destriped images. (a) MODIS band 24. (b) MODIS band 30. (c) MODIS band 21. (d) Washington DC band 1. (e) Real infrared image. (f) Urban band 1.

of the key intermediate variables during the processing such as $\boldsymbol{d}_x(i, j)$, $\boldsymbol{d}_y(i, j)$, $\boldsymbol{b}_x(i, j)$, $\boldsymbol{b}_y(i, j)$ and $\boldsymbol{u}(i, j)$ are set as 9 bits, in which the top digit represents the sign of the variables. The fractional parts of these variables are set as m bits. For assessing the finite precision effects of our proposed method, the MATLAB implementation is set as the reference. The obtained PSNR difference between the floating point reference model and finite precision model varying the precision of the fractional part is detailed in Fig. 2(b). The curves indicate that the difference of PSNR saturates after m reaches 8, and such phenomenon holds true for all the three cases of stripe level. Therefore, we set $m = 8$ to represent the fractional parts of the variables $\boldsymbol{d}_x(i, j)$, $\boldsymbol{d}_y(i, j)$, $\boldsymbol{b}_x(i, j)$, $\boldsymbol{b}_y(i, j)$ and $\boldsymbol{u}(i, j)$ in a dedicated hardware architecture for 8-bit images, to reduce hardware resources overhead while preserving good stripe-removal performance.

IV. FPGA Architecture and Implementation

A. Fully Pipelined Iteration Unrolled Hardware Architecture

The overview of the proposed fully pipelined iteration unrolled hardware architecture is depicted in Fig. 4. The main feature is that it mainly consists of n unrolled ICBs, which are spread out in cascade to form a coarse-grained row-level parallelism, and alleviate the data dependency between iterations. Each ICB executes a single iteration task, and contains three sub-blocks $\boldsymbol{u}$ updater, $\boldsymbol{d}$ updater, and $\boldsymbol{b}$ updater, corresponding to the three computation steps of the split Bregman method. The three sub-blocks are arranged in pipeline, leading to a fine-grained pixel-level parallelism. In addition, the parameters λ_1, λ_2, α, and β can be set manually according to the stripe intensity. With the pipeline characteristic, the throughput for coping with the image frame sequences is considerably improved, and the PD is greatly reduced, because the upcoming images need not to be buffered for long time to wait for the finish of destriping the current image.

The image input component and stripped image output component mainly consist of some simple controlling logic and buffers, and they will not be detailed in this article. In the following, we mainly introduce the technical details of the ICB. For convenience, we will refer to the neighbors of the current position as north, south, west, and east: $x(i-1, j) = x_n$, $x(i+1, j) = x_s$, $x(i, j-1) = x_w$, $x(i, j+1) = x_e$, where x represents variables like $\boldsymbol{u}$, $\boldsymbol{d}_x$, $\boldsymbol{d}_y$, $\boldsymbol{b}_x$, $\boldsymbol{b}_y$ and $\boldsymbol{I}_s$, besides, the element in the current position can be indicated as $x(i, j) = x_c$.

B. $\boldsymbol{u}$ Updater

The $\boldsymbol{u}$ updater is devoted to updating the variable $\boldsymbol{u}$ in equation (13). With the usage of Gauss-Seidel iteration method, the variables involved in the computation are partly from the current iteration, and the others are from the previous one. The variables context for $\boldsymbol{u}$ update computation are shown in Fig. 5(a). For the present ICB (assume the present ICB as the kth one), only $\boldsymbol{u}_n^k$ and $\boldsymbol{u}_w^k$ can be used, which are calculated and buffered in the current ICB, and variables such as $\boldsymbol{u}_s^{k-1}$ and $\boldsymbol{u}_e^{k-1}$ are from the previous ICB. Likewise, the variables $\boldsymbol{d}_x$, $\boldsymbol{d}_y$, $\boldsymbol{b}_x$, $\boldsymbol{b}_y$ and $\boldsymbol{I}_s$ are also the update results from the previous ICB. As shown in Fig. 5(b), two FIFOs and several registers can be used to provide the neighboring variables. FIFO1 and FIFO2 buffer the $\boldsymbol{u}$ values which are just updated by the computation logic in the current and the previous ICB, respectively. Similarly, three FIFOs with several registers are also used to form the computational context with respect to $\boldsymbol{d}_x$, $\boldsymbol{d}_y$, $\boldsymbol{b}_x$, $\boldsymbol{b}_y$, and $\boldsymbol{I}_s$. In the following, we choose $\boldsymbol{I}$ to represent the original image instead of $\boldsymbol{I}_s$ to avoid confusion.

1) Bottleneck for $\boldsymbol{u}$ Updater Implementation: The critical data dependency occurs because of the need for $\boldsymbol{u}_w^k$, which is the result of processing the previous pixel. In other words, $\boldsymbol{u}_w^k$ will not be obtained until the processing of previous pixel is finished. Squeezing all the update computations into one clock cycle is an option, but it will result in a too long data path and severely restrict the overall throughput. Therefore, the complex computations have to be divided into several phases to shorten the critical data path. Meanwhile, stalling the input pixels for a few clock cycles to wait for the $\boldsymbol{u}_w^k$ is also a natural choice, but it will inevitably cause extra delay for every pixel, and the whole PD of one image frame will be rather large. In conclusion, an efficient way

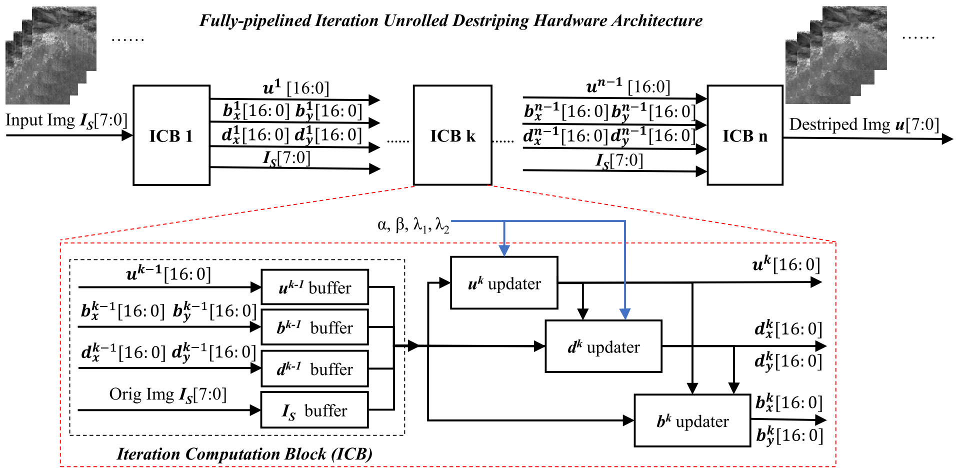


Fig. 4. Block diagram of the proposed fully pipelined iteration unrolled hardware architecture. A coarse-grained parallelism and a fine-grained parallelism are both achieved in the proposed architecture, leading to high-throughput and low-delay performance.

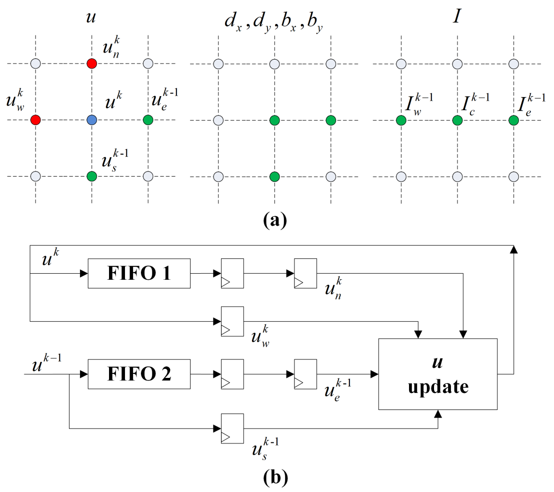


Fig. 5. (a) Context for $\boldsymbol{u}$ update computation. Red dots represent the values from previous iteration, green dots represent the values from current iteration. (b) Block scheme of partial context construction.

of obtaining and using $\boldsymbol{u}_w^k$ is the key of breaking through the bottleneck and realizing a high-throughput pipelined architecture.

2) *Timing Arrangement:* To alleviate the bottleneck, we propose a dedicated timing arrangement for the $\boldsymbol{u}$ updater: a special data path for $\boldsymbol{u}_w^k$ is designed, and the other variables involved are arranged to be processed in advance, as depicted in Fig. 6. The pipelined computation is arranged in five clock cycles, the computations not related to $\boldsymbol{u}_w^k$ start for four clock cycles ahead and are divided into three parts: y-direction, x-direction, and others. These three parts of calculations are performed simultaneously in three clock cycles, and the sum of the three intermediate results will not be obtained until the fourth cycle. In the fifth clock cycle, the newly obtained $\boldsymbol{u}_w^k$ participates in the final computation phase in time, as indicated by the red data path in Fig. 6. Therefore, the update of each $\boldsymbol{u}$ will not be postponed and the sequential pixels will be processed consecutively at each clock cycle.

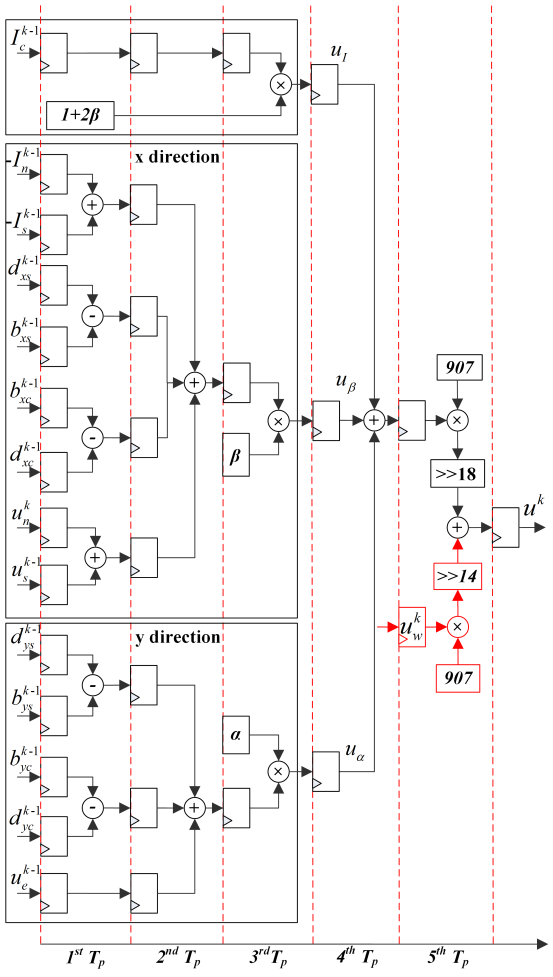


Fig. 6. Block diagram of the destriped image $\boldsymbol{u}$ updater. The entire process costs 5 pixel clock cycles. The red part in the fifth cycle represents the dedicated data path for $\boldsymbol{u}_w^k$, which needs optimization for acceleration.

3) *Optimization for Critical Path:* With the proposed timing arrangement, the processing speed of $\boldsymbol{u}$ updater is greatly improved, but a long data path still exists, i.e., the division

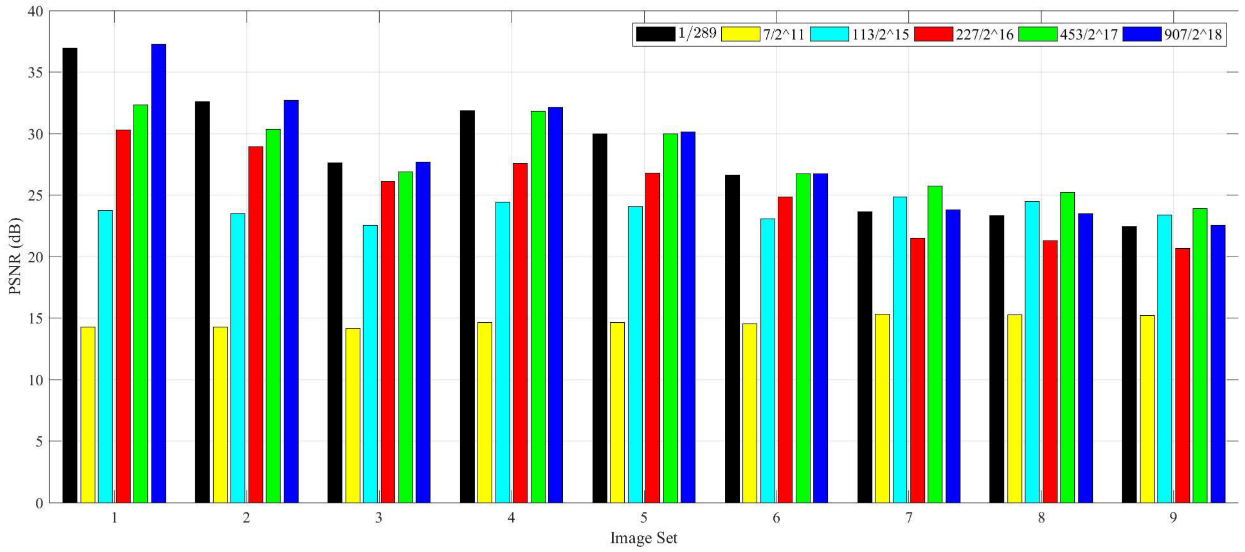


Fig. 7. PSNR values of different approximate schemes in terms of the critical path $u_w^k \times (1/(1+2\alpha+2\beta)$. The approximate schemes are in the form of $\times(\mathrm{N}/2^{\mathrm{K}})$, the black bar indicates the original division, the yellow, cyan, red, green, and blue bars represent N = 7 & K = 11, N = 113 & K = 15, N = 227 & K = 16, N = 453 & K = 17, N = 907 & K = 18, respectively. The blue bar shows the closet PSNR value to the black bar.

operation of $1 + 2\alpha + 2\beta$ in the final phase, which is considerably both time and area consuming. Thus, we propose a approximated simplification scheme to speed up the critical path. We replace the division with multipliers and bit shift operations as $\times(\mathrm{N}/2^{\mathrm{K}})$, and we have investigated the effects of different selection of N and K, the comparative results with respect to PSNR are depicted in Fig. 7. When more precise results are required, the bigger value of both N and K should be chosen, however, it will incur more computational delay and hardware resource expense. Thus, we should choose appropriate values of N and K for the balance between precision and time and resource overhead. The bar graph in Fig. 7 shows that, when compared with the original division, the approximation $907/2^{18}$ achieves the closest PSNR values among all the methods. Thus, the division can be simplified as a multiplier $\times 907$ with a bit shift operator $\gg 18$. As for the data path of $\boldsymbol{u}_w^k$, the original computation $\boldsymbol{u}_w^k \times (\alpha/(1+2\alpha+2\beta))$ can be directly transformed to $(\boldsymbol{u}_w^k \times 907) \gg 14$ while parameter α selects 16, as shown in Fig. 6. Furthermore, the multiplier $\times 907$ can be replaced with several shift registers and adders as $A \times 907 = A \ll 10 + A \ll 9 + A \ll 8 + A \ll 4 + A \ll 2 + A$, which will further speed up the computation. As a result, the division is purely simplified to a few shift registers and adders, and hardware resource overhead are considerably reduced. The result and performance will be discussed in Section V.

C. $\boldsymbol{d}$ and $\boldsymbol{b}$ Updater

Analogous to the $\boldsymbol{u}$ updater, Gauss–Seidel iteration makes use of the variables $\boldsymbol{u}_c^k$, $\boldsymbol{u}_w^k$, $\boldsymbol{u}_n^k$ from the $\boldsymbol{u}$ updater block in the present ICB, and the variables from the previous ICB $\boldsymbol{I}_c^{k-1}$, $\boldsymbol{I}_w^{k-1}$, $\boldsymbol{b}_{xc}^{k-1}$, and $\boldsymbol{b}_{yc}^{k-1}$. Data dependencies do not exist between pixels in the same iteration, so it is very easy to implement the computations in four clock cycles in pipeline, as shown in Fig. 8(a) and (b). Taking the $\boldsymbol{d}_x$ updater for example, in the first clock cycle, two subtractors and an adder are used to compute $\nabla_{\mathrm{x}}(\boldsymbol{u}-\boldsymbol{I}_s)+\boldsymbol{b}_x$, in the next clock cycle, the absolute value is obtained, and in the third clock cycle, the result of $|\nabla_{\mathrm{x}}(\boldsymbol{u}-\boldsymbol{I}_s)+\boldsymbol{b}_x| - (\lambda_2/\beta)$ is computed through a subtractor. Finally, in the fourth clock cycle, after comparing with 0 and instructed by the sign of $\nabla_{\mathrm{x}}(\boldsymbol{u}-\boldsymbol{I}_s)+\boldsymbol{b}_x$, $\boldsymbol{d}_x^k$ is obtained. The pipelined computations of $\boldsymbol{d}_x$ and $\boldsymbol{d}_y$ updaters are performed simultaneously.

As for $\boldsymbol{b}_x$, $\boldsymbol{b}_y$ updater, $\boldsymbol{I}_c^{k-1}$, $\boldsymbol{I}_w^{k-1}$, and $\boldsymbol{b}_{xc}^{k-1}$ are from the previous iteration, and $\boldsymbol{d}_{xc}^k$ is from the $\boldsymbol{d}_x$ updater in the current ICB, $\boldsymbol{u}_c^k$ and $\boldsymbol{u}_w^k$ are from the $\boldsymbol{u}$ updater in the current iteration. The computations of (18) are simply arranged in two clock cycles and performed simultaneously, as depicted in Fig. 8(c) and (d).

D. Data Flow of the Architecture

When each ICB receives the input variables, the $\boldsymbol{u}$ updater, $\boldsymbol{d}$ updater and $\boldsymbol{b}$ updater start to function sequentially after buffering the corresponding variables $\boldsymbol{u}$, $\boldsymbol{d}_x$, $\boldsymbol{d}_y$, $\boldsymbol{b}_x$, and $\boldsymbol{b}_y$. As shown in Fig. 9(a), for each pixel, the total computations only take 11 clock cycles, besides, each pixel can be processed continuously without stalling, leading to a fine-grained parallelism with high throughput and low PD.

Owing to the row buffers used in each ICB, there is about one row cycle interval between two adjacent ICBs, for example, only when the present ICB finishes processing one row of data, the corresponding outputs can be just streamed into the next ICB. This feature leads to a coarse-grained parallelism. The processing flow between ICBs is illustrated in Fig. 9(b), taking the architecture with 20 ICBs for example, when the update computations for the first row of data have been finished in the endmost ICB, the 21st row just reaches in the first ICB to start the first update computations. For an image frame, after the input of the first pixel, it only takes about the transmitting time of 20 rows of pixels to generate

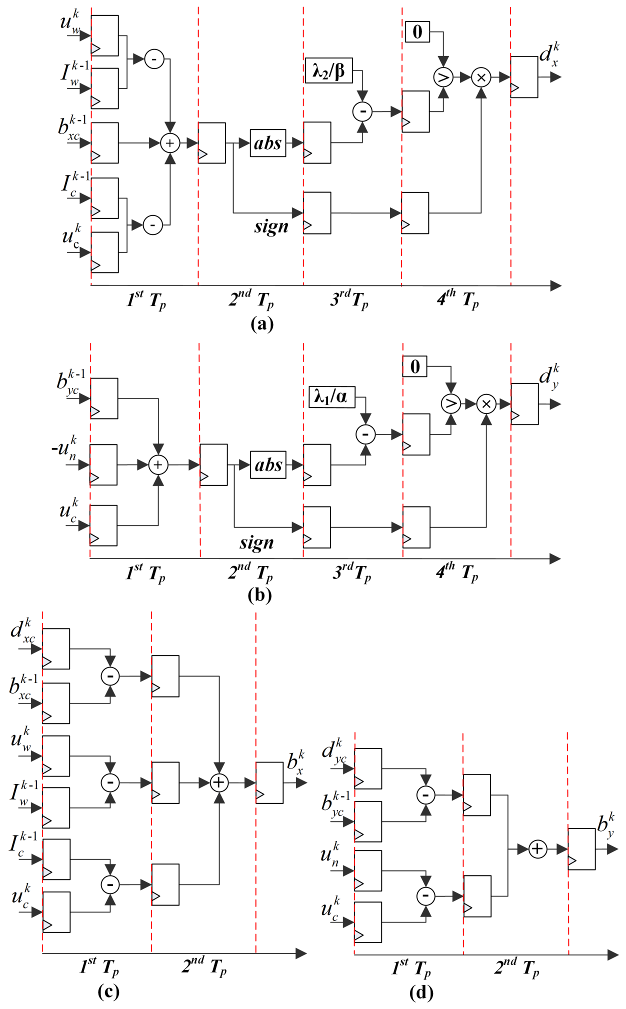


Fig. 8. Block diagrams of (a) $\boldsymbol{d}_x$ updater, (b) $\boldsymbol{d}_y$ updater, (c) $\boldsymbol{b}_x$ updater, (d) $\boldsymbol{b}_y$ updater. $\boldsymbol{d}_x$ and $\boldsymbol{d}_y$ updater take 4 pixel clock cycles, $\boldsymbol{b}_x$ and $\boldsymbol{b}_y$ updater take 2 pixel clock cycles.

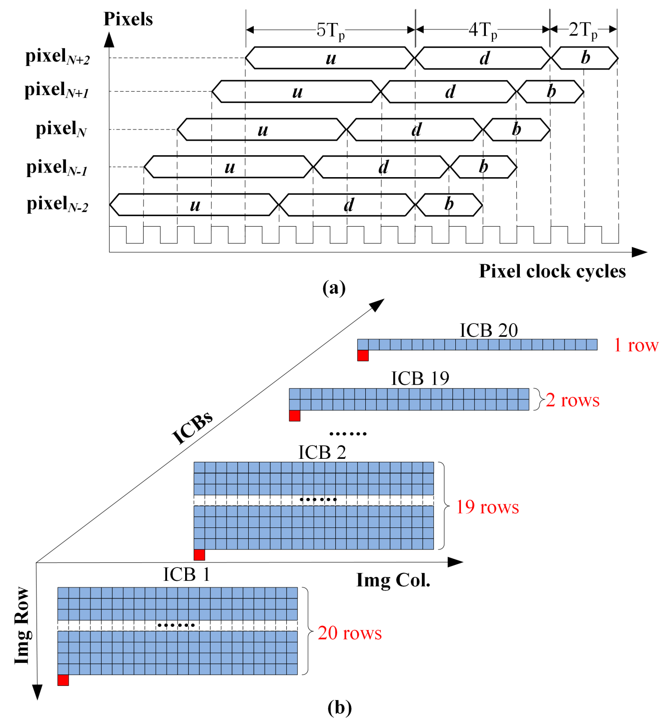


Fig. 9. Processing flow. (a) and (b) Processing flow of pixels within one iteration and between different iterations, respectively. The red dots represent the pixels being processed at present iterations.

the first pixel of the final destriped image, which shows very low PD.

V. Result and Discussion

A. FPGA Implementation Result

1) Throughput and Processing Delay: The proposed architecture is implemented on a single Xilinx xc6vlx240t FPGA in Verilog-HDL and is synthesized, placed, and routed with ISE 14.7. The maximum clock frequency of each updater block and the entire architecture is depicted in Table I. Since the critical path of the entire architecture exists in the $\boldsymbol{u}$ updater block, the overall maximum processing speed is exactly the maximum processing speed of the $\boldsymbol{u}$ updater itself. Owing to more consumption of the hardware resources, the implementation result with respect to maximum speed of the entire architecture is slightly lower than that of only implementing the $\boldsymbol{u}$ updater block, which is caused by the different place and route consequences. The maximum throughput of the architecture with 20 ICBs can reach up to 176.0 MPixels/s. There is no extra buffering logic inside the three updater blocks, thus the PD for them are 5, 4, and 2 clock cycles, respectively. The buffers are configured in each ICB, and each data buffer only stores a row of the input data, bringing a delay of about c clock cycles (Let r indicate image row and c indicate image column). In total, the PD in one ICB for an input pixel is the sum of computation delay and buffer delay, i.e., $11 + c$ clock cycles. The PD is positively associated with the number of deployed ICBs and the image column size, more specifically, with n ICBs deployed, the corresponding total delay is $n \times (11 + c)$ clock cycles, as shown in Table I. As a result, when 20 ICBs are deployed, the proposed architecture can destripe $10\,742 \times 8192$ images at 2 frames/s with PD of only 0.932 ms.

2) Advantage of Proposed Timing Arrangement and Approximate Simplification Method: To verify the proposed timing arrangement and approximate simplification method, an primitive implementation for $\boldsymbol{u}$ updater without the proposed optimization methods was used for comparison in terms of the resource usage and maximum working frequency. The comparison results are shown in Table II, Primitive indicates the $\boldsymbol{u}$ updater implemented without the proposed timing arrangement and approximate simplification method for the division, and all the update computations are performed in one clock cycle, resulting in a low maximum frequency of 28.3 MHz. Scheme1 is the first hardware implementation version, which transforms the original computation $\boldsymbol{u}_w^k \times (\alpha/(1 + 2\alpha + 2\beta))$ to $(\boldsymbol{u}_w^k \times 907) \gg 14$. The computation involving multiplication and division is simplified to multiplication with shifting, and the maximum frequency is promoted to 138.1 MHz. Afterward, we have further optimized the multiplication $\times 907$ to the form of $A \times 907 = A \ll 10 + A \ll 9 + A \ll 8 + A \ll 4 + A \ll 2 + A$, which is the Scheme2 in Table II. The

TABLE I
MAXIMUM FREQUENCY AND PD OF THE PROPOSED FPGA IMPLEMENTATION (T_p IS THE PIXEL CLOCK CYCLE, c IS THE COLUMN SIZE OF THE IMAGE, AND n STANDS FOR THE NUMBER OF ITERATIONS DEPLOYED IN THE ARCHITECTURE)

		Max Frequency(MHz)	Computation Delay	Buffer Delay	Total Delay
Sub-blocks	***u*** updater	190.0	$5T_p$	0	$5T_p$
	d updater	374.4	$4T_p$	0	$4T_p$
	b updater	578.4	$2T_p$	0	$2T_p$
Overall Architecture	n=1	176.9	$11T_p$	cT_p	$(11+c)T_p$
	n=10	176.0	$110T_p$	$10cT_p$	$(110+11c)T_p$
	n=15	175.8	$165T_p$	$15cT_p$	$(165+15c)T_p$
	n=20	176.0	$220T_p$	$20cT_p$	$(220+20c)T_p$

TABLE II
MAXIMUM SPEED AND RESOURCE EXPENSE WITH AND WITHOUT THE OPTIMIZATIONS

Scheme	$f_{\max}$ (MHz)	Resource Usage of ***u*** Updater		
		LUT	Flip-Flop	DSP48E1
Primitive	28.3	2628	307	0
Scheme 1	138.1	397	306	6
Scheme 2	**190.0(↑571%)**	**612(↓76.71%)**	**306**	**0**

TABLE III
RESOURCE USAGE OF THE PROPOSED ARCHITECTURE IN FPGA (xc6vlx240t)

resources	LUT	register	DSP48E1
u updater	612	306	0
d updater	319	143	0
b updater	135	14	0
1 ICB	1845(1.2%)	1305(0.4%)	0
10 ICBs	17863(11.8%)	12598(4.2%)	0
15 ICBs	27326(18.1%)	19270(6.4%)	0
20 ICBs	36640(24.3%)	25880(8.6%)	0

TABLE IV
FPGA (xc6vlx240t) MEMORY USAGE (Mbits) UNDER DIFFERENT ICBS AND DIFFERENT IMAGE SIZE CASES

image size	512 × 512	1024 × 1024	2048 × 2048
10 ICBs	1.25	2.50	5.00
15 ICBs	1.97	3.75	7.50
20 ICBs	2.50	5.00	10.00
1 ICB	32	128.00	512.00

TABLE V
FPGA (xc6vlx240t) ON-CHIP POWER CONSUMPTION (W) UNDER DIFFERENT ICBS AND DIFFERENT IMAGE SIZE CASES

image size	512 × 512	1024 × 1024	2048 × 2048
10 ICBs	3.637	3.673	3.759
20 ICBs	4.562	4.648	4.857

multiplication is simplified to several shift registers and adders, and the maximum speed is further improved to 190 MHz. Compared with Scheme1, the LUT usage has increased a few, but the special resource DSP48E1 usage is reduced to 0. Such feature can make our architecture transplantable in different FPGA platforms from different vendors, and as an application-specified integrated circuit (ASIC) eventually. In consequence, the implemented ***u*** updater with the proposed optimization methods has achieved a speedup of 571% and reduction of 76.71% in terms of LUT resource usage, helping accelerate the overall throughput and save the resource consumptions. Even if many ICBs are deployed, the resource expenses are much less than purely unrolling multiple ICBs without the proposed optimizations.

3) Resource Usage: As with the hardware resource usage, the overhead of FPGA logic elements such as LUT, register, and DSP48E1 are listed in Table III, and they are only related with the number of deployed ICBs. Each ICB utilizes only 1.2% LUTs and 0.4% registers, the unrolling of multiple iterations leads to the multiplication of the utilized logic elements. In addition, image input component, stripped image output component, and the buffering logic for variables $\boldsymbol{u}$, $\boldsymbol{d}_x$, $\boldsymbol{d}_y$, $\boldsymbol{b}_x$, $\boldsymbol{b}_y$, and $\boldsymbol{I}_s$ have introduced additional registers and LUTs. With 20 ICBs deployed, the usage of LUTs is 24.3% of the total amount in xc6vlx240t FPGA, meanwhile, 8.6% of the registers and none of DSP48E1 are used.

As for the architecture with fixed ICBs, its memory usage only varies with the size of the input images, especially the column size. As listed in Table IV, when processing images in size of 2048 × 2048 with 20 ICBs, the total memory usage is 10.00 Mbits, which is less than the total volume of the available ON-chip memory in xc6vlx240t FPGA (14.6 Mbits). Nevertheless, it is noteworthy that, if the swath of the image is too big, the ON-chip memory will be not enough, thus we would choose OFF-chip memory for data buffering.

4) Power Consumption: Owing to the low resource utilization, the ON-chip power consumption is low, as listed in Table V. The ON-chip power consumption results are generated by the XPower Analyzer of ISE 14.7. It can be observed that, the ON-chip power consumption mainly varies with the computational resource utilization, and it is slightly influenced by the utilized ON-chip BRAMs. For the implementation coping with 2048 × 2048 images with 20 ICBs deployed, its maximum ON-chip power consumption is only 4.857 W. Moreover, as is known to all, the usage of OFF-chip memories, such as SDRAM/DDR, is one of the main causes of the power consumption for the entire embedded system [44]. Due to the advantage of low utilization rate of memory of the proposed architecture, OFF-chip memory is not necessary for the embedded system, thus the main power consumption of the system comes from the FPGA, promoting the overall power efficiency of the system. In conclusion, along with low-resource-overhead characteristic, the low-power-consumption feature makes our proposed architecture suitable for power and resource-limiting ON-orbit applications.

TABLE VI
PD, TOTAL PROCESSING TIME (TT), MAXIMUM FRAME RATE (FR) OF PROPOSED, AND BUFFER-LOOP ARCHITECTURES. (176 MHz WORKING FREQUENCY)

		PD	TT	FR
512×512				
Proposed	10 ICBs	29.7μs	1.52ms	671.4fps
	15 ICBs	44.6μs	1.53ms	671.4fps
	20 ICBs	59.4μs	1.55ms	671.4fps
Buffer-loop	10 loops	13.4ms	14.89ms	67.2fps
	15 loops	20.8ms	22.29ms	44.9fps
	20 loops	28.3ms	29.79ms	33.6fps
1024×1024				
Proposed	10 ICBs	58.8μs	6.01ms	167.8fps
	15 ICBs	88.2μs	6.03ms	167.8fps
	20 ICBs	117.6μs	6.07ms	167.8fps
Buffer-loop	10 loops	53.6ms	59.55ms	16.8fps
	15 loops	83.4ms	89.35ms	11.2fps
	20 loops	113.2ms	119.15ms	8.4fps
2048×2048				
Proposed	10 ICBs	116.9μs	23.94ms	41.9fps
	15 ICBs	175.5μs	24.0ms	41.9fps
	20 ICBs	234.0μs	24.05fms	41.9fps
Buffer-loop	10 loops	214.4ms	238.22ms	4.2fps
	15 loops	333.6ms	357.42ms	2.8fps
	20 loops	452.8ms	476.62ms	2.1fps

B. Comparison Against Other Implementations

As mentioned above, the hardware implementations of image destriping algorithms are rarely reported, let alone variational model-based methods. Only work in [45] presents an image destriping hardware architecture based on FPGA, which adopts a filtering-based destriping algorithm, and its destriping performance is not as good as variational-based algorithms. Therefore, we choose other real-time implementations of iterative algorithms for comparisons. Usually, the implementation of iterative algorithms is in the form of buffer-loop, as the works in [46] and [47]. The core structure of the buffer-loop architecture is implementing only one ICB and using massive memory resources for buffering the intermediate results. The fixed number of iterative computation tasks are all performed only by one ICB at different times. In this section, we mainly compare the performance of the proposed architecture against that of the buffer-loop architecture.

1) Resource Usage: Apparently, the biggest advantage of the buffer-loop architecture is its slight logic resource usage for implementing only one ICB. No matter how many iterations are required to perform on one image, its total logic resource usage equals nearly only one ICB of our proposed architecture, as can be seen in the fifth row of Table III. However, as for the memory usage, the disadvantage of the buffer-loop architecture is evident. Different from our proposed architecture, the intermediate results such as $\boldsymbol{u}(i, j)$, $\boldsymbol{d}_x(i, j)$, $\boldsymbol{d}_y(i, j)$, $\boldsymbol{b}_x(i, j)$, $\boldsymbol{b}_y(i, j)$ and the original image $\boldsymbol{I}_s$ are all in the same size of the input images and need to be buffered, so the memory overhead have increased a lot, as listed in the bottom row of Table IV. When dealing with images in size of 2048×2048, the overall memory usages reach up to 512 Mbits, which is a huge burden even if OFF-chip memory is used.

2) Real-Time Performance: The processing flow of the buffer-loop architecture is that each input image needs to be iteratively computed for a fixed number by only one ICB, and the subsequent image should wait until the current image is completely processed, thus the follow-up image sequences are required to be stalled for a moment, decreasing the overall processing frame rate. To comprehensively compare the real-time performance of the proposed and the buffer-loop architecture, we select three indexes such as PD, total processing time, and maximum frame rate for quantitative comparisons. PD is the time interval from the input of the first pixel in the original image to the output of the first pixel in the destriped image. Total processing time is the time interval from the input of the first pixel in the original image to the output of the last pixel in the destriped image. For convenience of comparison, we assume the proposed and the buffer-loop architectures both work in the frequency of 176 MHz. As shown in Table VI, we can see clearly that no matter how many iterations are required and how large the size of the image is, the PD of the buffer-loop architecture is three orders longer than that of the proposed architecture. The total processing time of the buffer-loop architecture is 1 order longer than that of the proposed architecture. As with the maximum frame rate, the proposed architecture still outperforms the buffer-loop architecture. It is worth noting that, the maximum frame rate of the proposed architecture does not vary with the number of deployed ICBs, and it has negative correlation with the image size. However, the maximum frame rate of the buffer-loop architecture is influenced both by iteration number and image size. Combining the comparisons with respect to the three indexes above, the proposed architecture shows much better real-time performance than that of the buffer-loop architecture. In essence, compared with the buffer-loop architecture, our proposed architecture mainly promotes the real-time performance and saves memory resources at the expense of consuming more computational resources.

TABLE VII
RUNTIME OF HARDWARE AND SOFTWARE IMPLEMENTATION

run time	256×256	512×512	1024×1024	2048×2048
software	0.602 s	2.397 s	9.363 s	37.342 s
hardware	372 μs	1.488 ms	5.956 ms	23.824 ms
speedup	1620×	1611×	1572×	1567×

3) Comparison Against Software Implementation: We also compare the hardware and software runtime performance in different image sizes, as listed in Table VII. The software implementation is performed on the C++ simulation model with a 3.20 GHz Intel Core i7-8700 CPU with 16 GB of RAM. Obviously, comparing with software implementation, the achieved speedup for the proposed hardware implementation is at least 1500×, which is rather considerable.

VI. CONCLUSION

In this article, a fully pipelined iterations unrolled hardware architecture for real-time image destriping with a unidirectional variational model is proposed. Firstly, the split Bregman and Gauss–Seidel methods are introduced to speed up solution and thus reduce the number of iterations. Then the involved iteration loop is unrolled and a coarse-grained pipelined architecture is proposed, in which a series of ICBs are deployed in cascade, alleviating the bottleneck caused by the data dependency between adjacent iterations. Secondly, for each ICB, a fine-grained pipelined architecture and dedicated timing arrangement are designed to break through the bottleneck caused by the data dependency within each iteration. Besides, an approximate simplification scheme is proposed to accelerate the critical path with less resource usage.

The proposed architecture is implemented on a XILINX 6vcx240t FPGA and achieves a maximum throughput up to 176 MPixels/s with delay of only tens of row cycles for 8-bit images. It is the first high-performance hardware implementation of variational image destriping for large-swath remote-sensing images with high data rate. The proposed architecture can be fully implemented by some basic logic resources and easily exported to other FPGA or ASIC technologies. In the future, the proposed architecture can be developed to intellectual property (IP), and it can be embedded with the space-borne ship detection [48] or cloud detection [49] units to build an integrated system.

References

[1] K.-S. Chen, M. M. Crawford, P. Gamba, and J. S. Smith, "Introduction for the special issue on remote sensing for major disaster prevention, monitoring, and assessment," *IEEE Trans. Geosci. Remote Sens.*, vol. 45, no. 6, pp. 1515–1518, May 2007.

[2] K. E. Joyce, S. E. Belliss, S. V. Samsonov, S. J. McNeill, and P. J. Glassey, "A review of the status of satellite remote sensing and image processing techniques for mapping natural hazards and disasters," *Progr. Phys. Geogr.*, vol. 33, no. 2, pp. 183–207, Jun. 2009.

[3] L. Zhang and B. Luo, "Recent developments in remote-sensing technologies for disaster management in China," *URSI Radio Sci. Bull.*, vol. 87, no. 1, pp. 19–24, Mar. 2014.

[4] B. Zhong, W. Chen, S. Wu, L. Hu, X. Luo, and Q. Liu, "A cloud detection method based on relationship between objects of cloud and cloud-shadow for Chinese moderate to high resolution satellite imagery," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 10, no. 11, pp. 4898–4908, Nov. 2017.

[5] Y. Ji-Yang, H. Dan, W. Lu-Yuan, G. Jian, and W. Yan-Hua, "A real-time on-board ship targets detection method for optical remote sensing satellite," in *Proc. IEEE 13th Int. Conf. Signal Process. (ICSP)*, Nov. 2016, pp. 204–208.

[6] H. He, Y. Lin, F. Chen, H.-M. Tai, and Z. Yin, "Inshore ship detection in remote sensing images via weighted pose voting," *IEEE Trans. Geosci. Remote Sens.*, vol. 55, no. 6, pp. 3091–3107, Jun. 2017.

[7] A. Di Simone, H. Park, D. Riccio, and A. Camps, "Sea target detection using spaceborne GNSS-R delay-Doppler maps: Theory and experimental proof of concept using TDS-1 data," *IEEE J. Sel. Topics Appl. Earth Observ.*, vol. 10, no. 9, pp. 4237–4255, Sep. 2017.

[8] C. Zhang, B. Li, H. Jiang, H. Li, and J. Chen, "High throughput hardware architecture of a MIMO-based sea land segmentation for on-orbit remote sensing image processing," in *Proc. Int. Conf. Vis., Image Signal Process. (ICVISP)*, Sep. 2017, pp. 142–146.

[9] J.-J. Pan and C.-I. Chang, "Destriping of landsat MSS images by filtering techniques," *Photogramm. Eng. Remote Sens.*, vol. 58, p. 1417, Jan. 1992.

[10] J. Torres and S. O. Infante, "Wavelet analysis for the elimination of striping noise in satellite images," *Opt. Eng.*, vol. 40, no. 7, pp. 1309–1315, Jul. 2001.

[11] P. Rakwatin, W. Takeuchi, and Y. Yasuoka, "Stripe noise reduction in MODIS data by combining histogram matching with facet filter," *IEEE Trans. Geosci. Remote Sens.*, vol. 45, no. 6, pp. 1844–1856, Jun. 2007.

[12] B. Münch, P. Trtik, F. Marone, and M. Stampanoni, "Stripe and ring artifact removal with combined wavelet-Fourier filtering," *Optics Exp.*, vol. 17, no. 10, pp. 8567–8591, May 2009.

[13] H.-S. Jung, J.-S. Won, M.-H. Kang, and Y.-W. Lee, "Detection and restoration of defective lines in the SPOT 4 SWIR band," *IEEE Trans. Image Process.*, vol. 19, no. 8, pp. 2143–2156, Aug. 2010.

[14] J. Chen, Y. Shao, H. Guo, W. Wang, and B. Zhu, "Destriping CMODIS data by power filtering," *IEEE Trans. Geosci. Remote Sens.*, vol. 41, no. 9, pp. 2119–2124, Sep. 2003.

[15] R. Pande-Chhetri and A. Abd-Elrahman, "De-striping hyperspectral imagery using wavelet transform and adaptive frequency domain filtering," *ISPRS J. Photogramm. Remote Sens.*, vol. 66, no. 5, pp. 620–636, Sep. 2011.

[16] F. Tsai and W. W. Chen, "Striping noise detection and correction of remote sensing images," *IEEE Trans. Geosci. Remote Sens.*, vol. 46, no. 12, pp. 4122–4131, Dec. 2008.

[17] F. L. Gadallah, F. Csillag, and E. J. M. Smith, "Destriping multisensor imagery with moment matching," *Int. J. Remote Sens.*, vol. 21, no. 12, pp. 2505–2511, Nov. 2000.

[18] B. K. P. Horn and R. J. Woodham, "Destriping LANDSAT MSS images by histogram modification," *Comput. Graph. Image Process.*, vol. 10, no. 1, pp. 69–83, May 1979.

[19] M. Wegener, "Destriping multiple sensor imagery by improved histogram matching," *Int. J. Remote Sens.*, vol. 11, no. 5, pp. 859–875, May 1990.

[20] G. Corsini, M. Diani, and T. Walzel, "Striping removal in MOS-B data," *IEEE Trans. Geosci. Remote Sens.*, vol. 38, no. 3, pp. 1439–1446, May 2000.

[21] H. Shen and L. Zhang, "A MAP-based algorithm for destriping and inpainting of remotely sensed images," *IEEE Trans. Geosci. Remote Sens.*, vol. 47, no. 5, pp. 1492–1502, May 2009.

[22] H. Carfantan and J. Idier, "Statistical linear destriping of satellite-based pushbroom-type images," *IEEE Trans. Geosci. Remote Sens.*, vol. 48, no. 4, pp. 1860–1871, Apr. 2010.

[23] M. Bouali and S. Ladjal, "Toward optimal destriping of MODIS data using a unidirectional variational model," *IEEE Trans. Geosci. Remote Sens.*, vol. 49, no. 8, pp. 2924–2935, Aug. 2011.

[24] J. Zhao *et al.*, "Single image stripe nonuniformity correction with gradient-constrained optimization model for infrared focal plane arrays," *Opt. Commun.*, vol. 296, pp. 47–52, Feb. 2013.

[25] Y. Chang, H. Fang, L. Yan, and H. Liu, "Robust destriping method with unidirectional total variation and framelet regularization," *Opt. Exp.*, vol. 21, no. 20, pp. 23307–23323, Oct. 2013.

[26] Y. Chang, L. Yan, H. Fang, and H. Liu, "Simultaneous destriping and denoising for remote sensing images with unidirectional total variation and sparse representation," *IEEE Geosci. Remote Sens. Lett.*, vol. 11, no. 6, pp. 1051–1055, Jun. 2014.

[27] N. Acito, M. Diani, and G. Corsini, "Subspace-based striping noise reduction in hyperspectral images," *IEEE Trans. Geosci. Remote Sens.*, vol. 49, no. 4, pp. 1325–1342, Apr. 2011.

[28] J. Fehrenbach, P. Weiss, and C. Lorenzo, "Variational algorithms to remove stationary noise: Applications to microscopy imaging," *IEEE Trans. Image Process.*, vol. 21, no. 10, pp. 4420–4430, Oct. 2012.

[29] X. Lu, Y. Wang, and Y. Yuan, "Graph-regularized low-rank representation for destriping of hyperspectral images," *IEEE Trans. Geosci. Remote Sens.*, vol. 51, no. 7, pp. 4009–4018, Jul. 2013.

[30] H. Zhang, W. He, L. Zhang, H. Shen, and Q. Yuan, "Hyperspectral image restoration using low-rank matrix recovery," *IEEE Trans. Geosci. Remote Sens.*, vol. 52, no. 8, pp. 4729–4743, Aug. 2014.

[31] Y. Chang, L. Yan, H. Fang, and C. Luo, "Anisotropic spectral-spatial total variation model for multispectral remote sensing image destriping," *IEEE Trans. Image Process.*, vol. 24, no. 6, pp. 1852–1866, Jun. 2015.

[32] X. Kuang, X. Sui, Q. Chen, and G. Gu, "Single infrared image stripe noise removal using deep convolutional networks," *IEEE Photon. J.*, vol. 9, no. 4, pp. 1–13, Aug. 2017.

[33] Z. He, Y. Cao, Y. Dong, J. Yang, Y. Cao, and C.-L. Tisse, "Single-image-based nonuniformity correction of uncooled long-wave infrared detectors: A deep-learning approach," *Appl. Opt.*, vol. 57, no. 18, pp. D155–D164, 2018.

[34] P. Xiao, Y. Guo, and P. Zhuang, "Removing stripe noise from infrared cloud images via deep convolutional networks," *IEEE Photon. J.*, vol. 10, no. 4, pp. 1–14, Aug. 2018.

[35] Y. Chang, L. Yan, L. Liu, H. Fang, and S. Zhong, "Infrared aerothermal nonuniform correction via deep multiscale residual network," *IEEE Geosci. Remote Sens. Lett.*, vol. 16, no. 7, pp. 1120–1124, Jul. 2019.

[36] Y. Chang, M. Chen, L. Yan, X.-L. Zhao, Y. Li, and S. Zhong, "Toward universal stripe removal via wavelet-based deep convolutional neural network," *IEEE Trans. Geosci. Remote Sens.*, vol. 58, no. 4, pp. 2880–2897, Apr. 2020.

[37] V. Rana, I. Beretta, D. Atienza, A. A. Nacci, M. D. Santambrogio, and D. Sciuto, "Design methods for parallel hardware implementation of multimedia iterative algorithms," *IEEE Des. Test. IEEE Des. Test. Comput.*, vol. 30, no. 4, pp. 71–80, Aug. 2013.

[38] V. Rana, I. Beretta, F. Bruschi, A. A. Nacci, D. Atienza, and D. Sciuto, "Efficient hardware design of iterative stencil loops," *IEEE Trans. Comput-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 12, pp. 2018–2031, Dec. 2016.

[39] T. Goldstein and S. Osher, "The split Bregman method for L1-regularized problems," *SIAM J. Imag. Sci.*, vol. 2, no. 2, pp. 323–343, Apr. 2009.

[40] D. L. Donoho, "De-noising by soft-thresholding," *IEEE Trans. Inf. Theory*, vol. 41, no. 3, pp. 613–627, Mar. 1995.

[41] J. N. Tritsiklis, "A comparison of Jacobi and Gauss-seidel parallel iterations," *Appl. Math. Lett.*, vol. 2, no. 2, pp. 167–170, Nov. 1988.

[42] F. Auger, B. Feuvrie, F. Li, and Z. Luo, "Multiplier-free divide, square root, and log algorithms," *IEEE Signal Process. Mag.*, vol. 28, no. 4, pp. 122–126, Jul. 2011.

[43] Y. Mathlouthi, A. Mitiche, and I. B. Ayed, "Regularised differentiation for image derivatives," *IET Image Process.*, vol. 11, no. 5, pp. 310–316, May 2017.

[44] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Nov. 2017.

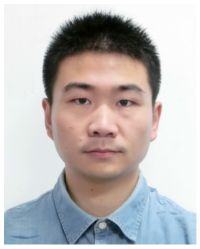
[45] J. Chen and H. Jiang, "The efficient-parallel stripe noise removal algorithm with low resource utilization based on FPGA," in *Proc. IEEE 2nd Int. Conf. Signal Image Process. (ICSIP)*, Aug. 2017, pp. 137–143.

[46] M. Martina and G. Masera, "Mumford and Shah functional: VLSI analysis and implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 3, pp. 487–494, Mar. 2006.

[47] J. Zeng, J. Lin, and Z. Wang, "An improved gauss-Seidel algorithm and its efficient architecture for massive MIMO systems," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 65, no. 9, pp. 1194–1198, Sep. 2018.

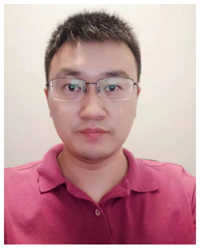
[48] K. Willburger, K. Schwenk, and J. Brauchle, "AMARO-an on-board ship detection and real-time information system," *Sensors*, vol. 20, no. 5, pp. 1324–1346, Feb. 2020.

[49] N. Shan, T.-Y. Zheng, and Z.-S. Wang, "Onboard real-time cloud detection using reconfigurable FPGAs for remote sensing," in *Proc. 17th Int. Conf. Geoinformatics*, Aug. 2009, pp. 1–5.



Liqun Chen received the B.S. degree from the School of Materials Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, China, and the Ph.D. degree from the School of Artificial Intelligence and Automation, HUST, in 2012 and 2019, respectively.

He is currently a Post-Doctorate with the School of Artificial Intelligence and Automation, HUST. His research interests include computer vision and image processing, in particular, embedded image processing system.



Yi Chang (Member, IEEE) received the B.S. degree in automation from the University of Electronic Science and Technology of China, Chengdu, China, in 2011, the M.S. degree in pattern recognition and intelligent systems from the Huazhong University of Science and Technology, Wuhan, China, in 2014, and the Ph.D. degree from the School of Artificial Intelligence and Automation, Huazhong University of Science and Technology in 2019.

From 2014 to 2015, he was a Research Assistant with Peking University, Beijing, China. He was a Research Intern with the Machine Learning Group, Tencent Youtu Laboratory, Shenzhen, China. He is currently a Post-Doctorate with the Artificial Intelligence Research Center, Peng Cheng Laboratory, Shenzhen. His research interests include multispectral image processing and structural noise removal.

Luxin Yan (Member, IEEE) received the B.S. degree in electronic communication engineering and the Ph.D. degree in pattern recognition and intelligence system from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2001 and 2007, respectively.

He is currently a Professor with the School of Artificial Intelligence and Automation, HUST. His research interests include multispectral image processing, pattern recognition, and real-time embedded system.