MEDIAN BASED PARALLEL STEERING KERNEL REGRESSION FOR IMAGE RECONSTRUCTION

G Dada Khalandhar, V Sai Ram, M Srinivasa Rao, Lalith Srikanth C, Pallav Kumar Baruah, Balasubramanian S, R Raghunatha Sarma

Sri Satya Sai Institute of Higher Learning, India
{dadakhalandhar, v.sairam1, msrini.svn}@gmail.com, {lalithsrikanthc, pkbaruah, sbalasubramanian, rraghunathasarma}@sssihl.edu.in

ABSTRACT

Image reconstruction is a process of obtaining the original image from corrupted data. Applications of image reconstruction include Computer Tomography, radar imaging, weather forecasting etc. Recently steering kernel regression method has been applied for image reconstruction [1]. There are two major drawbacks in this technique. Firstly, it is computationally intensive. Secondly, output of the algorithm suffers form spurious edges (especially in case of denoising). We propose a modified version of Steering Kernel Regression called as Median Based Parallel Steering Kernel Regression Technique. In the proposed algorithm the first problem is overcome by implementing it in on GPUs and multi-cores. The second problem is addressed by a gradient based suppression in which median filter is used.

Our algorithm gives better output than that of the Steering Kernel Regression. The results are compared using Root Mean Square Error(RMSE). Our algorithm has also shown a speedup of 21x using GPUs and shown speedup of 6x using multi-cores.

KEYWORDS

Steering Kernel Regression, Local Gradient, Multi-core, GPU.

1. INTRODUCTION

Images have become inherent part of day to day life. The image processing algorithms to improve the quality of images are innumerable, but many of them are application specific. Recently, Takeda et al.[1] have proposed a novel regression technique for image reconstruction known as Steering Kernel Regression. There are two major drawbacks associated with this technique.

It is computationally intensive and, the output of the algorithm suffers from spurious edges (especially in case of denoising). In the proposed algorithm, the first problem is overcome by implementing the algorithm on GPUs and multi-cores. The second problem is addressed by deblurring of the output using median filter. GPU based efficient solutions for data parallel image processing applications have been proposed by many authors [5][6][7][8]. Like many image processing applications, Steering Kernel Regression also has inherent data parallelism. Taking advantage of this inherent parallelism, we have identified and implemented the three most time consuming parts of the algorithm both on multi-core and GPUs. We have addressed the second problem by suppressing the spurious edges. These edges are produced due to noise and get stronger with the number of iterations of the algorithm. In the proposed technique we use median filter at the end of each iteration to remove the spurious edges.

2. Data Adaptive Kernel Regression

Regression is a process of finding the underlying signal in a given data, where the original signal is corrupted by noise. Many image regression techniques like edge-directed interpolation [2], and moving least squares [3] were proposed. Classical parametric regression methods assume that there is a specific model for underlying signal and estimate the parameters of this model. The parametric estimation has been used in major image processing techniques. The model generated out of the estimated parameters is given as the best possible estimate of the underlying signal.

In contrast, non parametric regression methods don't assume any underlying model, but depend on the data itself to arrive at the original signal. Regression function is the implicit model that has to be estimated. Takeda et al. [1] introduced steering kernel regression for image processing and reconstruction and have shown that it out-performs existing regression techniques. A brief introduction to the algorithm is given as follows:

The measured data is given by $y_i = z(x_i) + \varepsilon_i$, i=1,...,P, where $z(x_i)$ is a regression function acting on pixel coordinates, ε_is are independent and identically distributed zero mean noise values, P indicates total number of pixels. Assuming that the regression function is smooth to a certain order N, the objective functional estimation of $z(x)$ can be deduced (detailed derivation is given in [1]) by minimization of the following functional:

$$\min_{\{\beta_n\}} \Sigma_{i=1}^{P} [y_i - \beta_0 - \beta_1^T(x_i - x) - \beta_2^T(x_i - x)^2 - \ldots \beta_N^T(x_i - x)^N]^2 \frac{1}{h} K_H\left(\frac{x_i - x}{h}\right) \quad (1)$$

where β_i is i^{th} derivative of $z(x)$ and h is global smoothing parameter. $K_H(x_i - x)$ is the kernel weight assigned to the samples and is defined such that the nearby samples are given higher weight than the farther ones.

Takeda et al.[1] have proposed to use a non-linear combination of data. In other words, data adaptive kernel regression not only depends on the sample location and density but also on the radiometric properties.

The incorporation of these radiometric properties were done by adapting the regression kernel locally, so that the kernel aligns itself along the features such as edges, than across them. This enables us to capture the features in better detail. The major contribution of Takeda et al.[1] is to use adaptive kernel regression for image processing and reconstruction. They call this adaptive technique, steering kernel regression.

The central notion of the *steering kernel regression* is to estimate the local gradients. The gradient information captures the features of the image. These in turn are used to find the weights to be assigned to the neighboring samples. Pixel near an edge will be influenced by the pixels of the same side of the edge. With this intuition in mind, the dominant orientation of the local gradients are measured. The kernel is then effectively steered locally based on this dominant orientation.

Our *Median Based Parallel Steering Kernel Regression Technique* is an iterative technique. The output of the previous iteration is passed as input for the next iteration. As the first step of the algorithm, output of the Classical Kernel Regression is passed as input for the first iteration of the algorithm. Brief description of the algorithm is as follows:

1. Finding local gradients at each image point.
2. Based on the gradient value at a pixel, the scaling, elongation and rotation parameters are estimated. There are used to construct the steering matrix.
3. Application of Steering Kernel Regression algorithm.
4. Post-processing of the resultant image to suppress spurious edges.
5. Repeating the steps 2 to 4, till the noise level in the output image is below the predefined threshold.

3. IMPLEMENTATION

The serial implementation of steps 1 to 3 have been implemented by Takeda et.al. We have used their code, which is available at http://www.soe.ucsc.edu/ htakeda/MatlabApp, for the improvement. The parallel implementation of the algorithm is discussed in section 3.1. In section 3.2 we dwell on the method to suppress spurious edges.

3.1 PARALLEL IMPLEMENTATION

The serial code was programmed in Matlab and is available at http://www.soe.ucsc.edu/ htakeda/MatlabApp. The code was profiled using Matlab Profiling Tool and it was observed that the steps 1 and 3 of steering kernel regression are most time consuming. On careful analysis, we discovered that these steps have a lot of scope for parallelism. The serial implementation of step 3 consists the following ideas:

1. To all pixels, in each iteration running till the square of the upscale factor, determine the feature matrix.
2. For each pixel,

   - Obtain the weight matrix using the neighboring samples of covariance matrix.
   - Compute equivalent kernel which involves inverse of a matrix resulting from the product of the feature and weight matrices.
   - Estimate the pixel values and gradient structure values for the output image.

In Step 1, classic regression method computes feature matrix, weight matrix and equivalent kernel. The most time consuming action involved in this method is to estimate the target values, local gradient structure values along the axes directions.

Steps 1 and 3 of steering kernel regression involve a lot of data-parallelism and CUDA is used to parallelize these steps. Mex interface feature integrates CUDA into Matlab. We shall now discuss the parallel implementation of the step 3 :

- Copy the original image, covariance matrix, feature matrix from the CPU host memory to GPU global memory.
- Kernel is launched with the total number of threads equaling the total number of pixels in the source image.
- A thread is assigned to each pixel and it does the above mentioned steps of determining weight matrix and equivalent kernel.
- It contributes R^2 pixels in the estimated output image, where R is the upscaling factor.

Computing the steering matrix (step 2) is done on multi-core for speed up of the whole process.

OPTIMIZATIONS

Since shared memory is on chip, accesses to it are significantly faster than accesses to global memory. Feature matrix is copied from host memory to shared memory of GPU. Only one thread in a block will get the data from global to shared memory. Shared memory could only fit feature matrix as the image size exceeds it's space limitations.

For the remaining data, the global memory request for a warp is split into two memory requests, one for each half-warp, which are issued independently. GPU hardware can combine all memory requests to a single memory transaction if the threads in a warp access consecutive memory locations[4]. Our model is designed in such a way that the memory access pattern by threads is coalesced, thereby improving performance and ensuring low latency. The multi-core code is implemented using the Parallel Computing Toolbox of Matlab environment. The performance comparison of these two implementations is given in the following section.

3.2 SPURIOUS EDGE SUPPRESSION

Spurious edges produced by single iteration of the steering kernel algorithm are not as strong as the original edges in the image. Due to which the gradient strength of the spurious edges is much lower than that of the original edges. Using this idea, we can suppress the edges which have the gradient strength less than a given threshold(it will be set based on the value of edges in the image). The value of the threshold depends upon the application. It can be static or given by a heuristic. We have used thresholds derived from the maximum gradient values. For suppressing the spurious edge pixels we applied median filter over the pixel's neighbourhood.

4. RESULTS

4.1 COMPUTATIONAL EFFICIENCY

In this section we show the results of using GPUs and multicores to improve the computational complexity of the first three steps of our algorithm(Steering Kernel Regression). We observe that the computational efficiency of the algorithm has been improved while maintaining the quality of the output images. We can show that the GPU and multi-core implementations are consistent with results of original serial implementation. In the next section, we present the results of our algorithm(which incorporates post processing module for suppressing the spurious edges)in comparison with Steering Kernel Regression.

EXPERIMENTAL SETUP

For all experiments, the parallelized version of the steering kernel regression was run on Tesla T20 based Fermi GPU. The serial and multi-core implementations of the same were run in Matlab R2012a on a node with Intel(R) Xeon(R) 2.13GHz processor and 24 GB RAM. CUDA timers were employed, which have resolution upto milliseconds to time the CUDA kernels and more importantly they are less subject to perturbations due to other events like page faults, interrupt from the disks. Also CudaEventRecord is asynchronous; there is less of Heisenberg effect when timing is short, exactly suitable to GPU-intensive operations as in these applications. For the multi-core code timers provided by Matlab were used.

EXPERIMENTS.

Experiments and the performance results of GPU, multi-core implementations on simulated and real data are presented. These experiments were conducted on diverse applications and attest the claims made in previous sections. In all experiments, we considered regularly sampled data.

An important note to consider is the determination of the maximum speed up achieved by an application. Understanding the type of scaling that is applicable in any application is vital in estimating the speed up. All the applications that are mentioned here exhibit scaling. An instance of this phenomenon is shown in Figure 1. One can observe that for a fixed problem size, increase in the number of processing elements, the time of execution decreases.

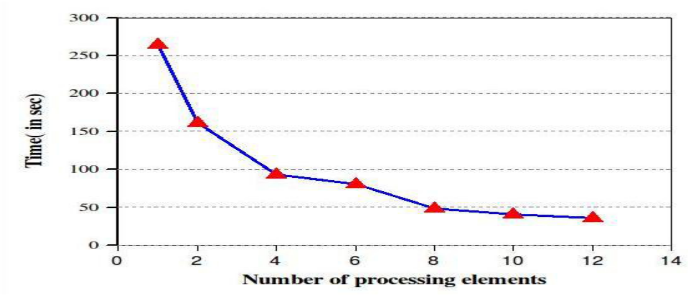


Fig. 1. Plot of execution time against number of processing elements.

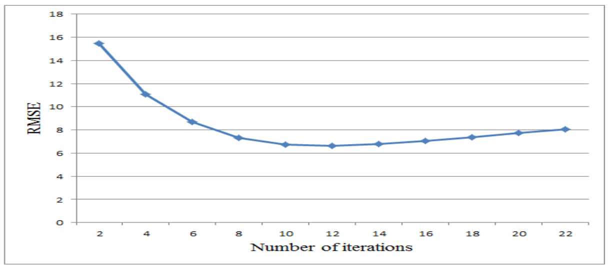


Fig. 2. RMSE(root mean square error) of the resulted image with respect to original image against Number of iterations

Quality Test.

We performed the quality test for the image by visual inspection and RMSE (root mean square error) values obtained from all the implementations. We have tested the comparisons between the serial code, multi-core and CUDA when applied on a CT scan image with Gaussian noise with standard deviation $\sigma = 35$. The RMSE values of the images resulted from multi-core and GPU implementations are 13.2, which is closer to serial result. We observed that quality was maintained with out any visually plausible differences. For all the experiments, the initial estimates are given by the classical kernel regression method. Different experiments that were performed are:

Denoising Experiment.

Well known Lena image of 512×512 size and a picture of a pirate with 1024×1024 size were consideration for testing. Controlled simulated experiment was set up by adding white Gaussian noise with standard deviation of $\sigma = 25$ to both the images. We set global smoothing parameter $h = 2.4$. The choice of number of iterations was set after careful analysis of the behavior of the application. The graph in the Figure 2, indicates that RMSE (root mean square error) values of the resulted image drop till a point and then it raises. This point was observed to be 12 for this application. In this way, a limit for the number of iterations is deduced for all the mentioned images. The RMSE values of the resulted images are 6.6426 and 9.2299.

Clearly, the dominance of the GPU performance over multi-core can be evidently seen in Figure 3(a). The multi-core code was run with 12 Matlab workers and as expected a near 10x performance is achieved for image size 1024×1024. The slack is possibly due to the communication overhead of the function calls performed by Matlab.

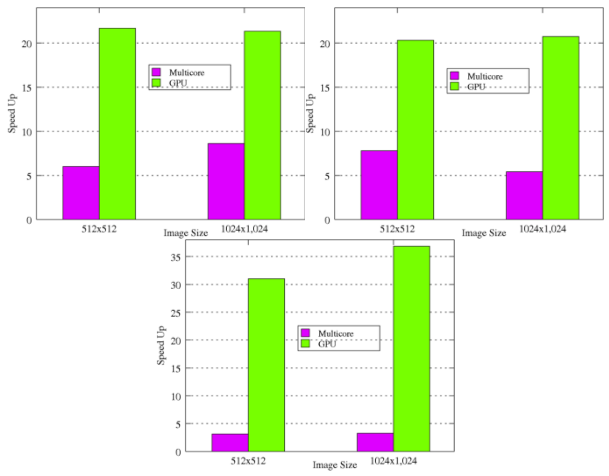


Fig. 3. Comparison of various algorithms: (a) Speedup factors against image sizes for the denoise application. (b) Speedup factors against image sizes for the compression artifact removal. (c) Speedup factors against image sizes for the upscaling.

Compression artifact removal.

Pepper image was considered for this experiment and compressed it by Matlab JPEG routine with a quality parameter 10 and RMSE of 9.76. The number of iterations and h were set to 10 and 2.4. RMSE values for single core version is 8.5765, multi-core version is 8.5759 and that of GPU is

8.5762. The performance is shown in Figure 3(b). The multi-core code has a average speed up factor of 6x over serial implementation whereas GPU has speed up of 20x.

Upscaling.
We performed an upscale operation on the lena image and on the pirate's picture. The performance is noted in the Figure 3(c). In this case, the GPU performance is achieved to be significantly higher than the multi-core. The arithmetic intensity is high for this application as each thread needs to estimate more number of pixels in the target image, precisely 4 (the upsampling factor).

Kernel Timing Results.
Table I presents the runtime measurements of the kernels in single core, multi-core and GPU for different image sizes. The timings given for GPU code also include the data transfer time from host to GPU memory and vice-versa. The GPU code maintains the efficiency even with the increase in the image size. The GPU version of classic kernel has achieved a factor of 175x over the serial implementation.An improvement factor of 75x was observed for steering kernel.

Table 1. Kernel execution timings(in seconds)

Image size	Regression method	Single core	Multi-core	GPU
512x 512	Classic kernel	11.765	1.765	0.067
512x 512	Steering kernel	105.140	18.200	1.470
1024x 1024	Classic kernel	47.741	6.930	0.270
1024x 1024	Steering kernel	427.701	102.234	5.819

4.2 QUALITATIVE IMPROVEMENT

The proposed algorithm has been compared with the algorithm proposed by Takeda et al. We found that by incorporating an additional post processing module based on median filter(in which the spurious edges are suppressed)into the Takeda algorithm, the proposed algorithm is giving better than the original algorithm. Both the algorithms have been applied on Lena gray scale image. The image is corrupted by Gaussian noise with standard deviation of 15, 25, 35 and 45 and then algorithms have been applied to denoise them. As shown in the figure, the error obtained in the proposed algorithm is less than that of the Takeda's algorithm. In our algorithm we have used threshold value as 1% of the maximum edge strength. We have used median over a neighbourhood of 1 pixel distance in fixing the value corresponding to spurious edges.

The RMSE value reaches a minimum after some iterations. Graphs corresponding to different images after different iterations is presented in Figure 4.

In denoising the Lena image with Gaussian noise = 15, Takeda's algorithm reached RMSE of 6.3917 while our algorithm reached 6.386. Both of them reach their minimum RMSE value in 8^{th} iteration and it is shown in Figure 4(a).

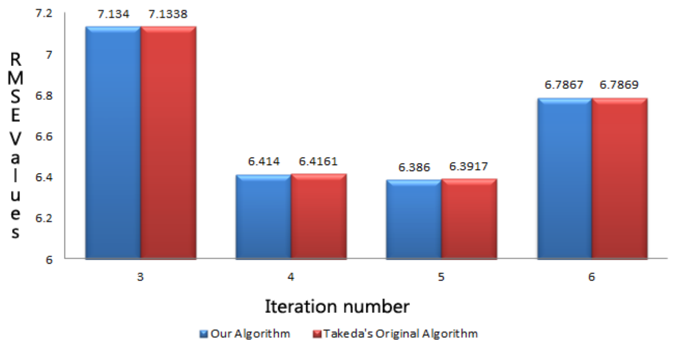
In denoising the Lena Image with noise of 25, Takeda's algorithm reached 7.8935 while our algorithm reached 7.8778. Both of them reach their minimum RMSE value in 9^{th} iteration and it is shown in Figure 4(b).

In denoising the Lena Image with noise of 35, Takeda's algorithm reached 9.2702 while our algorithm reached 9.2059. Both of them reach their minimum RMSE value in 18^{th} iteration and it is shown in Figure 4(c).

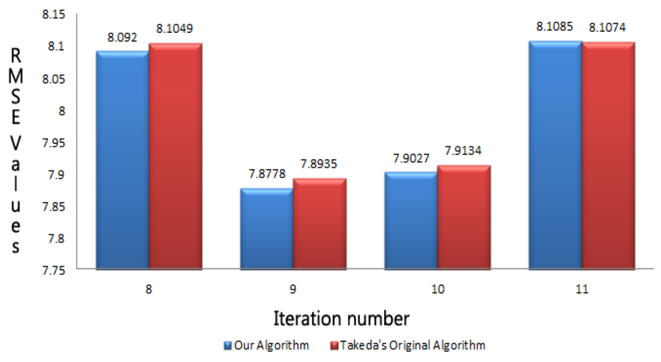
Finally, in denoising the Lena Image with noise of 45, Takeda's algorithm reached 10.823 while our algorithm reached 10.822. They reached their minimum RMSE value in 18^{th} iteration and it is shown in Figure 4(d).

We have shown the results of denoising the Lena Image with noise 35 in the Figure 5. Figure 5(a) denotes the original Lena image. Figure 5(b) is the Lena image with a Gaussian noise of 35.

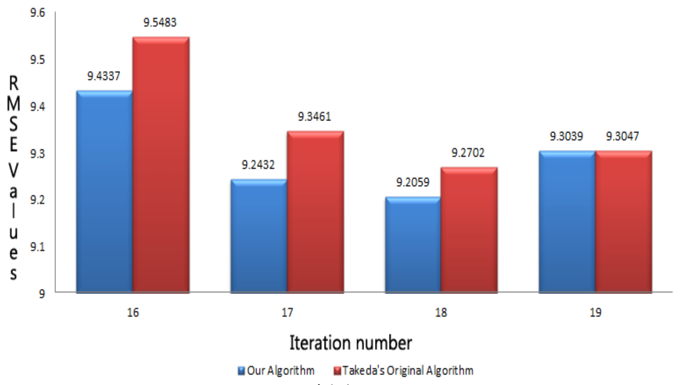
Figure 5(c) gives us the suppressed edges, the edges which have been suppressed by the proposed algorithm.



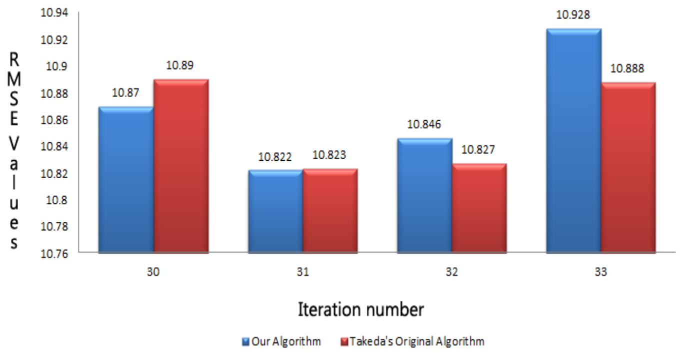
4(a)



4(b)

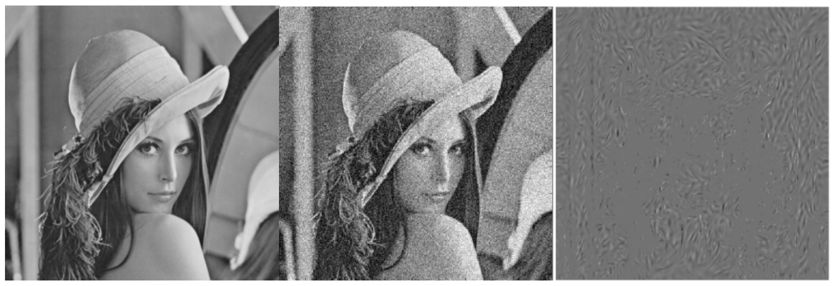


4(c)



4(d)

Fig. 4. Comparison of Our *Median Based Parallel Steering Kernel Regression Technique* with Takeda's Original algorithm on Lena image with Gaussian noise of (a)15 (b)25 (c)35 (d)45.



5(a) 5(b) 5(c)

Fig. 5. Comparison of our algorithm to Steering Kernel Regression. (a) Original Lena image. (b) Lena image with Gaussian noise of 35. (c) Suppressed spurious edges

5. Conclusions And Future Work

Steering kernel regression is indeed an innovative work in image reconstruction. This algorithm was successfully parallelized on both multi-core, GPU platforms using Matlab and CUDA. Implementations for different applications with varying parameters have been evaluated. From the observations, it is clearly evident that the time complexity of steering kernel has been reduced. On an average, a gain of $6\times$ and $21\times$ speed-up was achieved on the multi-core and GPU platforms respectively. Also an additional post processing module, based on median filter, has been introduced for the purpose of suppressing the spurious edges. This improvement has given better results than the original algorithm.

In future, this work could be extended to video data. All the image datasets considered here fit within the limits of GPU memory. Computing Steering Kernel regression on single GPU may not be scalable to larger images, as the on-board memory of GPU is a major constraint. Thus, parallelizing the algorithm for multiple GPUs may lead to better results.

Acknowledgment

We would like to dedicate our efforts to our Divine Founder Chancellor Bhagawan Sri Sathya Sai Baba, without whom this work wouldn't have been possible. We would like to acknowledge Takeda et.al. for making the Kernel Regression Tool-box software available on-line. This work was partially supported by nVIDIA, Pune, grant under Professor partnership program and DRDO grant under Extramural Research and Intellectual Property rights.

References

[1] Takeda, Hiroyuki, Sina Farsiu, and Peyman Milanfar. "Kernel regression for image processing and reconstruction." Image Processing, IEEE Transactions on 16.2 (2007): 349-366.

[2] Li, Xin, and Michael T. Orchard. "New edge-directed interpolation." Image Processing, IEEE Transactions on 10.10 (2001): 1521-1527.

[3] Bose, N. K., and Nilesh A. Ahuja. "Superresolution and noise filtering using moving least squares." Image Processing, IEEE Transactions on 15.8 (2006): 2239-2248.

[4] NVIDIA CUDA Programming Guide, [Online], Available at : http://developer.download.nvidia.com/.

[5] Kraus, Martin, Mike Eissele, and Magnus Strengert. "GPU-based edge-directed image interpolation." Image Analysis. Springer Berlin Heidelberg, 2007. 532-541.

[6] Tenllado, Christian, Javier Setoain, Manuel Prieto, Luis Piñuel, and Francisco Tirado. "Parallel implementation of the 2D discrete wavelet transform on graphics processing units: filter bank versus lifting." Parallel and Distributed Systems, IEEE Transactions on 19, no. 3 (2008): 299-310.

[7] Allusse, Yannick, Patrick Horain, Ankit Agarwal, and Cindula Saipriyadarshan. "GpuCV: A GPU-accelerated framework for image processing and computer vision." In Advances in Visual Computing, pp. 430-439. Springer Berlin Heidelberg, 2008.

[8] Risojević, Vladimir, Zdenka Babić, Tomaž Dobravec, and Patricio Bulić. "A GPU imple-mentation of a structural-similarity-based aerial-image classification." The Journal of Su-percomputing: 1-19.