Patch-Match for Style Transfer

ANUBHAV CHAUDHARY; 2016013 and YASHIT MAHESHWARY; 2016123

ACM Reference Format:
Anubhav Chaudhary; 2016013 and Yashit Maheshwary; 2016123. 2019. Patch-Match for Style Transfer. 1, 1 (April 2019), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Github Link

1 INTRODUCTION

Adding color to a grayscale images is a neat system that we see in a lot of softwares like Adobe Photoshop, gimp, etc. Most of the color grading software that are used in the industry are proprietary therefore their implementation is not known publicly. Color is usually added to improve the visual fidelity of old images or images which can't be captured with color (electron microscopic images). The process of coloring a grayscale image requires a colored input image which will be used to color a grayscale image depending on the pixel luminance and the neighbourhood statistics of the pixel. This implementation of Style Transfer over the GPU provides an optimized implementation to allow colorization of gray-scale images using existing similar colored images with the Patch-Match Algorithm. [1]

2 LITERATURE SURVEY

Before patch match nearest-neighbor search [3] was used to provide similar patches in an image. However, the cost of computation of finding this patch was high and instead a patch match algorithm was devised.

The Patch Match algorithm is widely used for adding missing patches or moving subjects [4] in an image while replacing the background of the place from where the subject is moved. The algorithm is based on the fact that good patch match can be found by doing random sampling in the image.

But this implementation of the algorithm involves transfer of a colored patch from a colored image to a gray-scale image (that is to be colored).

The concept behind the implementation can be achieved using two methods:

(1) A **colored image**, it's **gray-scale variant** and a **target image** that is to be colored are sent as an input to the algorithm. The algorithm finds the matching patch in the provided colored image and finds a suitable match based on the luminance of both the images, and transfers the entire patch onto the gray-scale image.
(2) A similar setup to the previous method is required. After a suitable patch has been found, the center pixel in the patch is transferred onto the gray-scale image to the corresponding center of the matched patch.

On comparing both the implementations, we found that Method 1 is faster; but it didn't seem to distinguish the boundaries of objects within the image; thus, causing some loss of information in the final image. We decided

Authors' address: Anubhav Chaudhary; 2016013; Yashit Maheshwary; 2016123.

XXXX-XXXX/2019/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

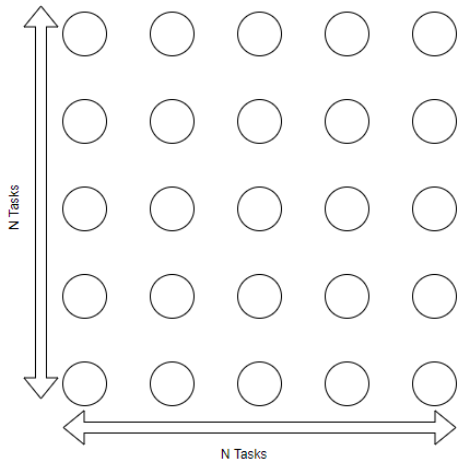
to put more emphasis on the quality of the image, as the performance would significantly be improved by implementing the same algorithm on the GPU.

3 ANALYSIS OF THE ALGORITHM

The algorithm requires a **colored image**, it's **gray-scale variant**, and a **target image** that is to be colored. A pixel wise matching approach is used to compare the luminance of the pixels in the converted gray-scale image and the image to be colored. A pixel wise matching approach is used to compare the luminance of the pixels in the converted gray-scale image and the image to be colored. For each pixel in the target image we go through each pixel in the gray-scale source image and try to find the best color for it using patch match algorithm which compares the luminance values of the neighbourhood pixels in a patch (usually of 5x5 size) to determine the best matching pixel to copy the color from. The algorithm assumes that the size of the input images will be a multiple of 32 in both width and height.

(1) Complexity of algorithm is $O(n^4)$.
(2) As shown in figure 1 below there is no dependency between any tasks.

Fig. 1. Dependency Graph. Each circle represents a pixel to be colored as a task.



4 PARALLELISATION STRATEGY

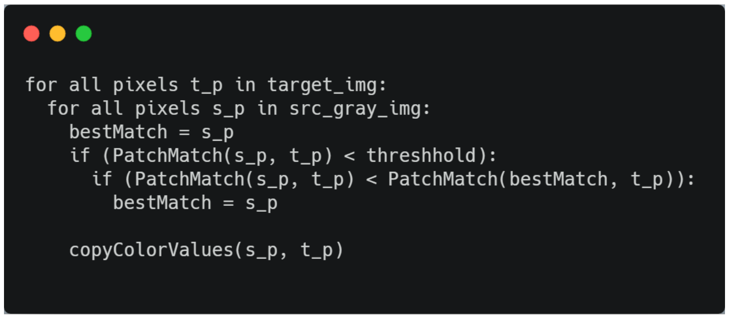
The complexity of our current algorithm is $O(n^4)$. While running the algorithm on the GPU we can reduce the computations required to iterate over all the pixels of the image. As we have a lot of threads in the GPU we can devote each thread to compute the best match for each pixel in the target image. Since the computation of the best matches for each individual pixel is independent, we can simply parallelize each one of them on the GPU.

4.1 GPU Input

We start with generating input for GPU. GPU Kernel takes in the following arguments *d_c_image, d_c_as_g_image, d_g_image, d_finalImage, maskCols, maskRows, c_width, c_height, g_width, g_height, threshold.*

(1) d_c_image: The colored image in GPU Memory.
(2) d_c_as_g_image: The colored image stored as grayscale in GPU Memory.

Fig. 2. Pseudo Code



```
for all pixels t_p in target_img:
  for all pixels s_p in src_gray_img:
    bestMatch = s_p
    if (PatchMatch(s_p, t_p) < threshhold):
      if (PatchMatch(s_p, t_p) < PatchMatch(bestMatch, t_p)):
        bestMatch = s_p

    copyColorValues(s_p, t_p)
```

(3) d_g_image: The grayscale image will be colored.
(4) d_finalImage: The finalImage where the colored result will be stored.
(5) maskCols: The columns of the neighbouring search region.
(6) maskRows: The rows of the neighbouring search region.
(7) c_width: The width of the colored image.
(8) c_height: The height of the colored image.
(9) g_width: The width of the grayscale image.
(10) g_height: The height of the grayscale image.
(11) threshold: The threshold used to define a "match" between colored and grayscale image.

We begin by selecting a **block size** of 32×32 and a **grid size** of image_width / $32 \times$ image_height / 32. Each block in a kernel will be responsible for coloring a square grid of 32×32 pixels. Inside the kernel we start by defining a region in shared memory which will hold the best match of the current pixel. Then for each thread (or pixel) in the current block we go through the colored image exhaustively and search for the best match of current pixel.

4.2 Deciding the best match

To determine the best match of a pixel we use the following metric: the sum of the absolute difference of the gray scale values in the neighbour search region around the current pixel and all the pixels in the colored image (only grayscale/luminance values are used to determine the best match) is treated as the **error** that we need to minimize. This method is implemented in a modular way, therefore it can be easily changed without even knowing how the rest of the code works. Once a best match is determined for a pixel then there are two cases possible.

(1) Current pixel has not been assigned any best match: In this case if the above mentioned error is below the **threshold** value (configurable by the user) then the current pixel is assigned as a best match for now. By rigours testing we found out that 200 works really well as the threshold value.
(2) Current pixel has a previously matched best pixel: In this case if the current error value is less than the threshold value and the previously assigned pixel's error value then the existing best match pixel is replaced by the new best match pixel.

As soon as we have the best match of a pixel, we transfer the color from the best matching pixel to the gray scale pixel. The algorithm continues like this for the remaining pixels in the gray scale image.

5 RESULTS[2]

The CPU code was run on a PC with 16 GB of ram and an i7-4710hq processor. The following results were obtained while trying to color various gray scale images of different sizes. The time taken to generate the results using CPU and GPU are mentioned in tables 1-3.

The graphs for the speedup obtained when using GPU as compared to CPU are given in Figures 27-29.

5.1 Special Case Results

An important observation made was that if the predefined colored image had a resolution smaller than that of the target image, the speedup obtained was higher; with almost no loss in quality. The speedup comparison graph has been given in Fig. 29.

We found an optimum balance between the quality of the image, and the time taken; when the predefined colored image has half the resolution of the target image.

5.2 64 x 64 images

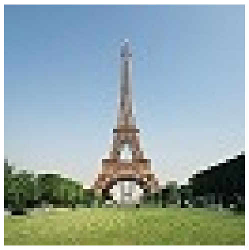


Fig. 3. Color Image

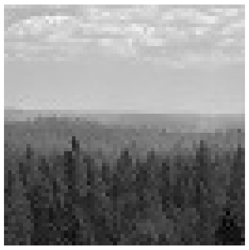


Fig. 4. Target Image

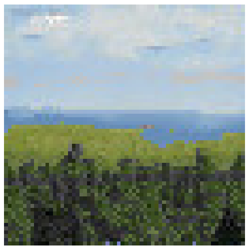


Fig. 5. Final Image

5.3 128 x 128 images

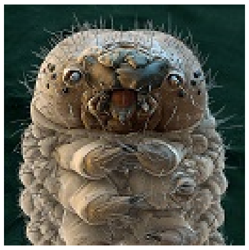


Fig. 6. Color Image

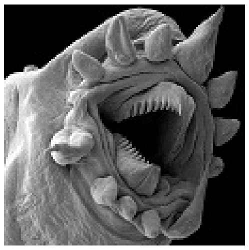


Fig. 7. Target Image

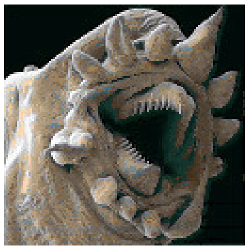


Fig. 8. Final Image

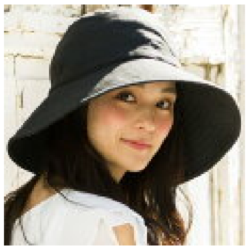


Fig. 9. Color Image

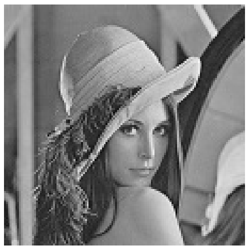


Fig. 10. Target Image

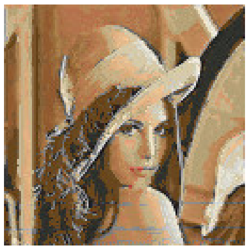


Fig. 11. Final Image

5.4 256 x 256 images

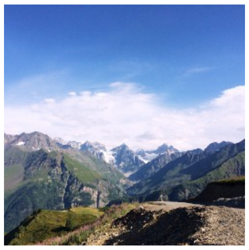


Fig. 12. Color Image

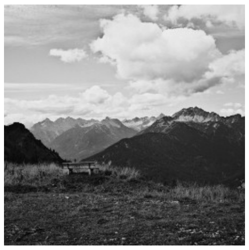


Fig. 13. Target Image

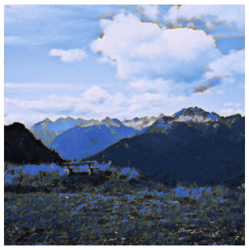


Fig. 14. Final Image

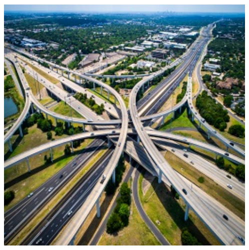


Fig. 15. Color Image

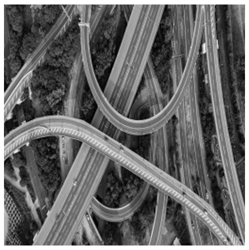


Fig. 16. Target Image

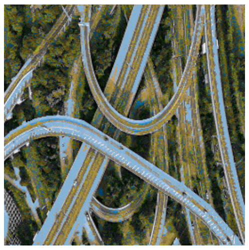


Fig. 17. Final Image

5.5 512 x 512 images

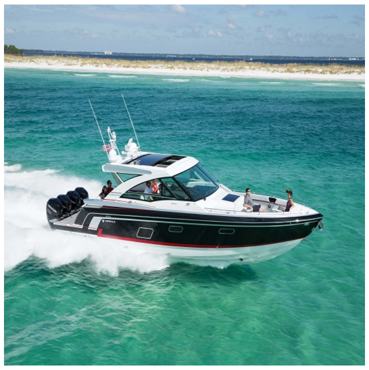


Fig. 18. Color Image

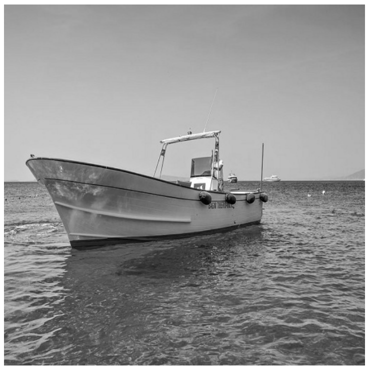


Fig. 19. Target Image

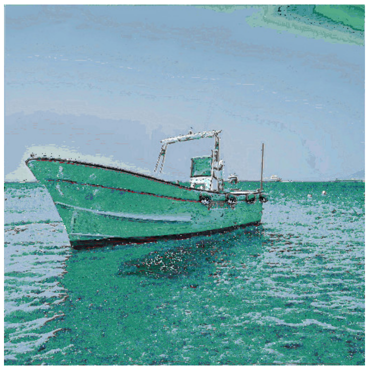


Fig. 20. Final Image

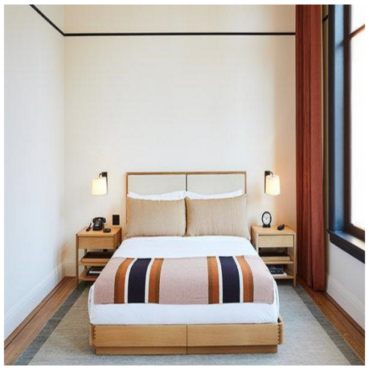


Fig. 21. Color Image

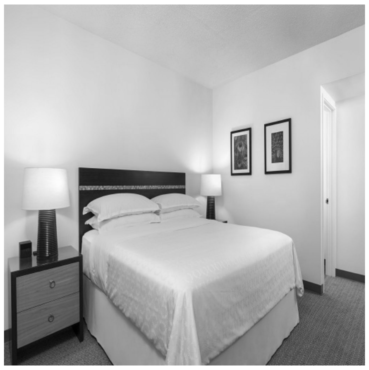


Fig. 22. Target Image

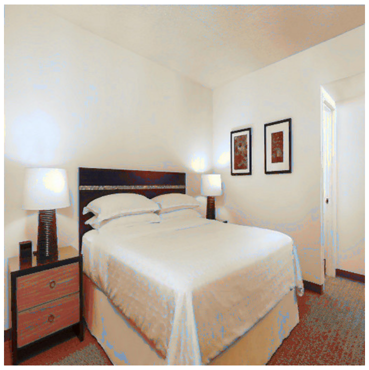


Fig. 23. Final Image

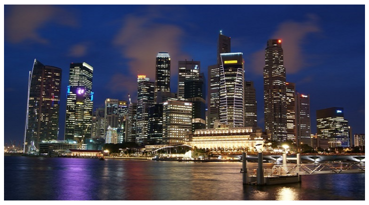


Fig. 24. Color Image

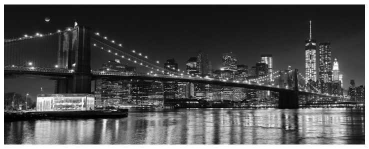


Fig. 25. Target Image

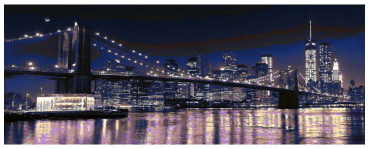


Fig. 26. Final Image

Image Size	64 x 64	128 x 128	256 x 256	512 x 512
Sample 1	2.34s	39.18s	649.70s	7560.58s
Sample 2		40.37s	647.51s	

Table 1. The time taken (in seconds) to color different size images on CPU.

Image Size	64 x 64	128 x 128	256 x 256	512 x 512
Sample 1	0.1181s	0.17063s	1.0570s	12.8487s
Sample 2		0.1677s		
*Sample 3		0.1291s	0.3757s	3.2815s

Table 2. The time taken (in seconds) to color images on GPU (Including IO).

Image Size	64 x 64	128 x 128	256 x 256	512 x 512
Sample 1	0.01742s	0.06947s	0.98785s	12.7973s
Sample 2		0.06977s		
*Sample 3		0.02059s	0.26871s	3.1829s

Table 3. The time taken (in seconds) to color images on GPU (Excluding IO).

*** - The colored image used was half the size of the target image**

	Speedup With IO	Speedup Without IO	Speical Case Speedup
64 x 64	134.328	19.813	
128 x 128	571.315	235.125	308.070
256 x 256	656.683	613.722	1726.656
512 x 512	590.794	588.830	2304.001

Table 4. Speedup with IO, Speedup Without IO and Special Case Speedup.

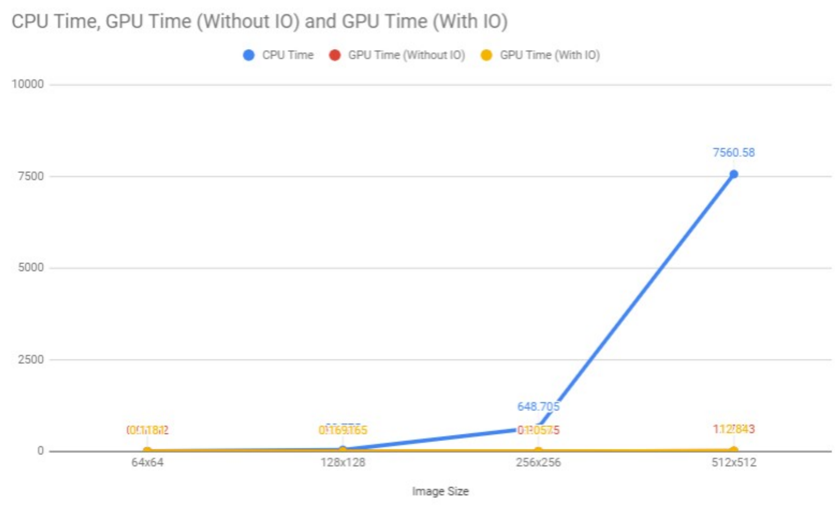


Fig. 27. Time taken by CPU vs GPU Code.

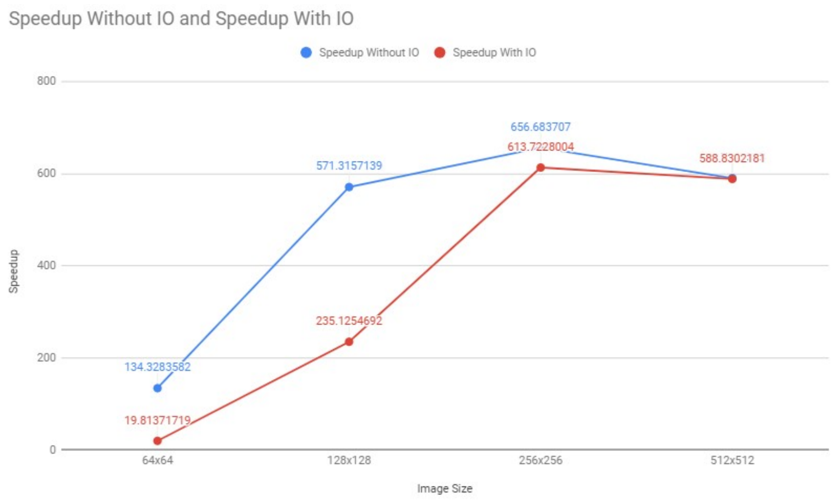


Fig. 28. Speedup including IO and excluding IO time.

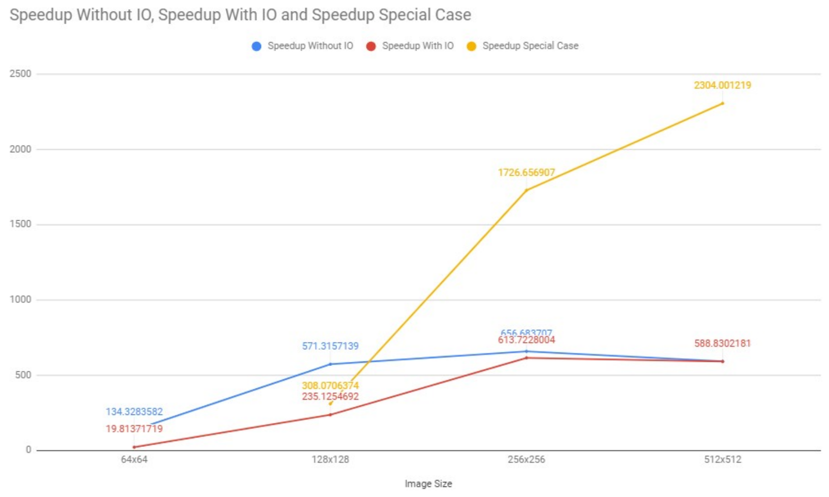


Fig. 29. Special Speedup compared to previously obtained Speedups

6 MILESTONES ACHIVED

(1) Implemented a CPU algorithm to provide a base case for calculating speedup.
(2) Decided neighbour N region to parallelize.
(3) Implemented algorithm to work on a GPU.
(4) Optimized the algorithm for quality vs speed trade off.
(5) Bench marked the code.
(6) Optimized the code for memory vs speed trade off.

REFERENCES

[1] Tomihisa Welsh, Michael Ashikhmin, Klaus Mueller [*Transferring color to greyscale images*]

[2] Google Images

[3] Nearest Neighbor Search - Wikipedia

[4] Connelly Barnes, Eli Shechtman, Adam Finkelstei, Dan B Goldman [*PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing*]