PatchMatch - Finding Correspondences in 3D Regions

Team 14

PRAGYA SETHI and VIBHU AGRAWAL

1 INTRODUCTION

Patch Match is an algorithm to match sections of one image to another and thus regenerate the other image from the first one. The project will focus on implementing the Patch Match algorithm [1] for 3D images. We will try to extend Patch Match algorithm to achieve style transfer between two 3D images. We will compare the speed ups achieved on GPU as compared to CPU.

2 LITERATURE REVIEW

Patch Based sampling is used for a wide variety of use cases in image manipulation like image completion (hole-filling), image resizing, image replication, etc. Barnes et al. [1] provided a randomised Nearest Neighbour Field (NNF) algorithm to find correspondences between two images. The algorithm proceeds in 3 steps: Initialisation, Propagation and Random Search.

(1) **Initialisation**: The image is initialised with random values or uniform independent samples from B. The distances between the input and target image using the randomly mapped patches are computed for each patch, where distance could be RGB based euclidean distance or come from some other distance function D.
(2) **Propagation**: For any image, if a certain patch is a good match, the probability of the surrounding patches to be good is high. If $D(\mathbf{v})$ denotes the distance between patch in source and target image, then for every patch, the patch to the left and above are checked and the distance of the current patch is updated to min{D(f(x, y)), D(f(x-1, y)), D(f(x, y-1))}, where f(a) is the offset between **a** in source image and its nearest neighbour **b** in target image. In even, iterations, the reverse scan is followed where the patches are scanned from bottom-right and the below and right patches are considered for update.
(3) **Random Search**: While using the nearby matched patches helps find optimal solution during propagation, it can at times get stuck in local minima. To avoid that, random samples around the current best match v_0 with exponentially decreasing distance:
$u_i = v_0 + w\alpha^i R_i$
are used to find the alternate minima, if possible. Here R_i is a uniform random in [-1, 1]*[-1, 1], w is the maximum image dimension and α is a ratio between window sizes (generally taken as 1/2), i varies from 1, 2, ...n until $w\alpha^i$ scans a range below 1 pixel.

In the algorithm, propagation and random search steps are performed alternatively, and it stops after a fixed number of iterations (4-5 have been found to suffice) or when there is no update in the mapping image between two iterations.

The algorithm can be extended to 3-dimensional space by extending the computations performed using 2D vectors to 3D vectors[2]. The patch now considered is 3D instead of 2D.

The algorithm can be sped up on GPU by adapting the jump flood scheme of Rong and Tan[2006] [4]. They run the algorithm parallely for each patch, using the previous closest match of neighbours to update the new closest match for the current patch via double buffering. This differs from the CPU algorithm as in CPU version, propagation of a close match happens along a scan line (top to bottom and left to right in odd iteration). So the changes in neighbour's close patch are reflected in current patch in the same iteration. However, on shifting to

Authors' address: Pragya Sethi, pragya18067@iiitd.ac.in; Vibhu Agrawal, vibhu18116@iiitd.ac.in.

double buffering, the output image would change. The difference between the images produced by two algorithms can be minimised by running the parallelised version over multiple iterations with small patch size.

3 APPROACH

The 3D patch match algorithm is closely based on the 2D algorithm proposed by Barnes et al[1]. It starts by iterating over different 3D patches which are 3D volumes. Initially, the algorithm maps each patch randomly to any other patch in the second image and finds distance between the mapped patch and the original one. Then for propagation, the neighbours of the current patch are explored in scanline. If the current patch is at (x, y, z), then in odd iterations, (x-1, y, z), (x, y-1, z) and (x, y, z-1) are considered as nearest neighbours and in the reverse, i.e. even iteration, (x+1, y, z), (x, y+1, z), (x, y, z+1) are considered. This arises from the fact that the neighbours considered have already been mapped to a better match before coming to this cell. However, in case of double buffering this does not hold true as all values of previous iteration are used. Consequently, the double buffered version tries to explore all neighbours of the current cell. For the random search step, the search starts from a maximum distance cell, which is randomised to prevent local minimas, or hitting the same point again and again. The propagation and random search step alternate for the number of iterations defined.

The last step of the algorithm is to create the target image using the segments found. For this we iterate over all patches of the first image and pick the patch of the second image which was claimed to be the best match from the second image for the first image by the algorithm.

The input to the algorithm is a 3D continuous image, which is multiple images stacked on top of each other. To use this image, we use SimpleITK [3] library of python and segment different layers of the 3D image into 2D images. We further use Numpy and Image library to dump them as '.jpeg' images and read them using STB_IMAGE library in C++. The same process is followed in reverse order to write the images in '.mhd' format.

The algorithm runs in $O(i * (n - p + 1)^3 * p^3 * log(n))$ where i is the number of iterations for which patch match algorithm is run, p is the patch size and n is the dimension of the 3D image.

3.1 Parallelisation Strategy

The double bufferred version of the algorithm is embarrassingly parallel in nature. Randomisation can occur in parallel for all patches. Thereafter, all patches can run their propagation and random search step independently. All threads just need to sync before every iteration. But the same could not be said for the propagation step of the original algorithm as that was working on the updates made for the neighboring pixel.

Also, the distance between two patches could be computed parallely using techniques like reduction for large patch size. However, in case of small patch sizes, this might prove to be an overkill.

The input reading and output writing part can also be parallelised for image of every layer (depth) to improve overall runtime.

The task dependency graph for the discussed parallelisation strategy is as shown in Figure 1.

3.2 Challenges

The different challenges that we faced so far and are expected further on are as enumerated:

- The 3D continuous coloured images are not easily available and hence we had to try out with Point clouds, 3D meshes initially, before getting the 3D images from TA Aradhya.
- The 2D implementation available as part of Barnes [1] paper was using Bitmaps and was heavily dependent on ImageMagick. We had to rewrite all the structures and implementation to decouple it from ImageMagick and to work for general '.jpeg' images which can be handled using STBI library of C++.
- There is no standard implementation or algorithm for 3D version of patchmatch. Hence we had to extend the 2D algorithm in 3 dimension using heuristics.

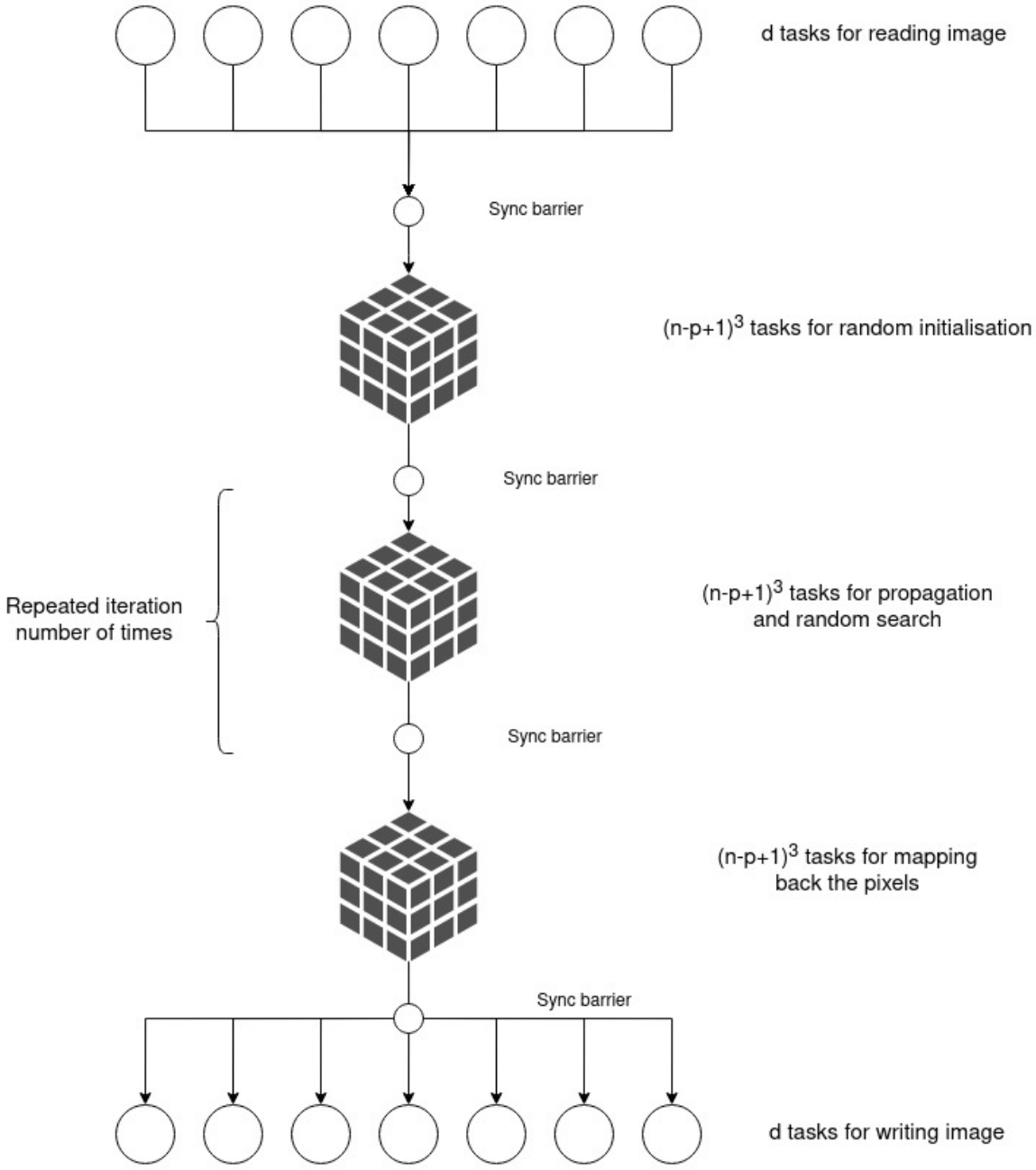

Fig. 1. **Task Dependency Graph:** The figure indicates the 5 tasks in which the whole algorithm and I/O will the broken down while parallelizing Patch Match.

- Due to absence of standard 3D algorithm, we had to analyse the serial version generated to see the error that would occur if we further try to parallelise it using double buffering.
- The huge size of images is an issue as we have to load large sections of memory, and maintain 4 such sections in general version of algorithm and they increase to 6 in double buffered case.

4 RESULTS

4.1 Patch wise results

We analyse the impact of variation in patch size on the time taken for completion and error in the input and output images. For this analysis,

(1) Iterations = 5
(2) Dimension of image = 64 x 256 x 128.

Here, the error is defined as $\Sigma\sqrt{distance}$ where distance is defined as the euclidean distance between the RGB values of two pixels.

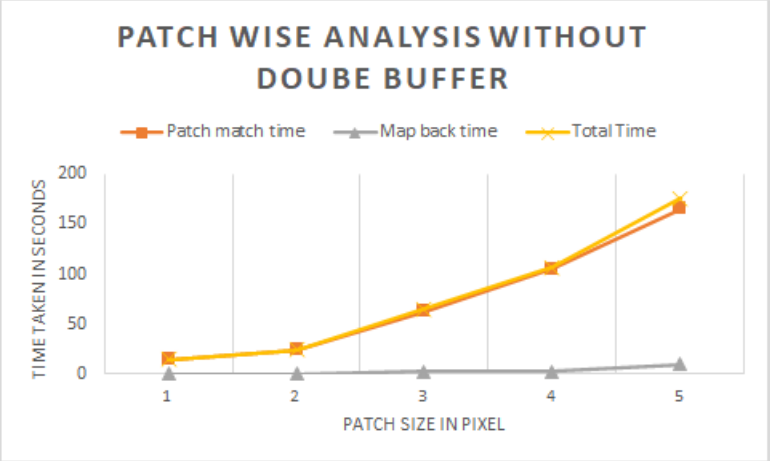

Fig. 2. Variation in time taken for different patch sizes without using double buffer.

4.1.1 *Inferences.*

(1) Core Patchmatch algrorithm takes most amount of time, and mapping back of images is almost negligible initially, but increases and becomes significant as the size of image increases. This leads us to the conclusion that parallelising both the phases in the GPU code maybe a good strategy.
(2) Time taken by the general algorithm and with double buffer is almost equal for a few initial cases. So, the overhead of double buffering may not be high in case of parallelism. However, the memory overhead is yet to be computed.
(3) The error per pixel in case of double buffer is a little higher as compared to the general algorithm, but the error ratio between the two algorithms remain almost constant even as the patch size increases. Thus modification of the algorithm to adapt to double buffers may not suffer from exponential errors as compared to general algorithm.

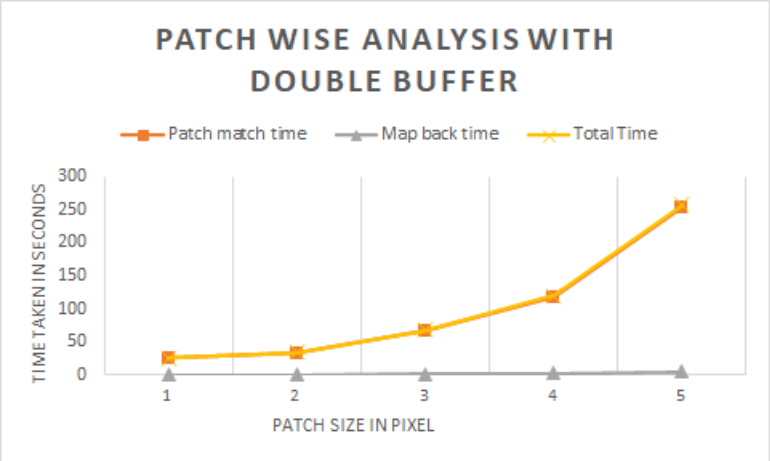

Fig. 3. Variation in time taken for different patch sizes with double buffer.

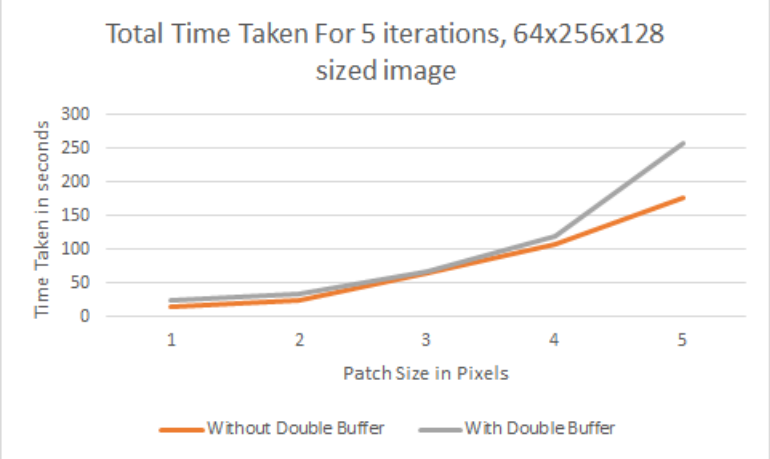

Fig. 4. Comparison of total time taken with and without double buffer on varying patch sizes.

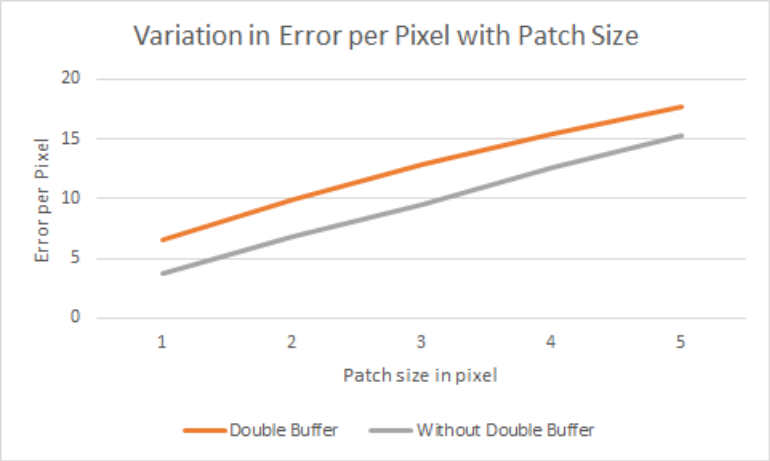

Fig. 5. Variation in error for different patch sizes with double buffer.

4.2 Volume wise results

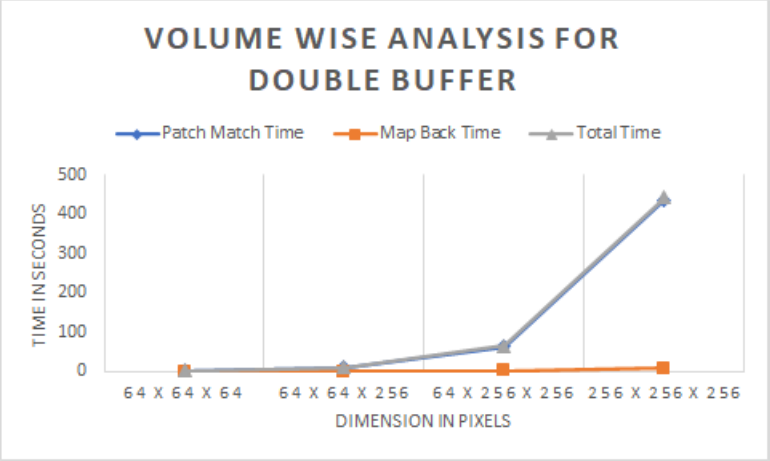

Fig. 6. Variation of time taken with variation in Volume.

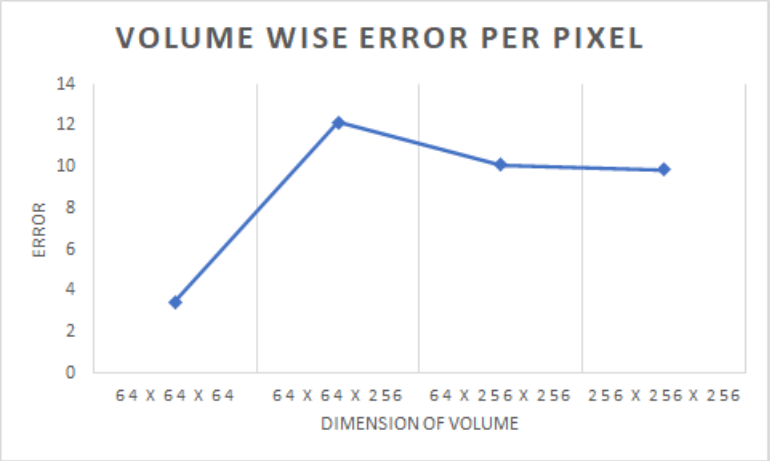

Fig. 7. Variation of Error per Pixel with variation in Volume with double buffer.

4.2.1 *Inference.*

(1) The patchmatch algorithm takes majority of time even as volume increases, and it starts increasing exponentially. So, increasing patch size and increasing volume both increases the execution time, with mapping back of output playing very small role in it.
(2) Even though the error would depend upon the images taken into consideration, still the analysis shows that the fluctuations are more or less lying in a constant range of about 3-13.

5 OUTPUT VOLUMES

Using the cubical patches of volume 2, we re-create volume 1. Here are the 4 images. Error per pixel = 9.87993.

(1) Volume 1 - Input volume 1 constructed from a 256 x 256 x 256 volume of Brain Image offset at (0, 0, 0). This is the image that has to be reconstructed.
(2) Volume 2 - Input volume 2 constructed from a 256 x 256 x 256 volume of Brain Image offset at (100, 100, 100). This is the volume whose patches are used to create volume 1.
(3) Output - 3D volume reconstructed by the cubicle patches of Volume 2 using patch match algorithm.
(4) ANN - The mapping of pixel of volume 1 to pixel of volume 2.

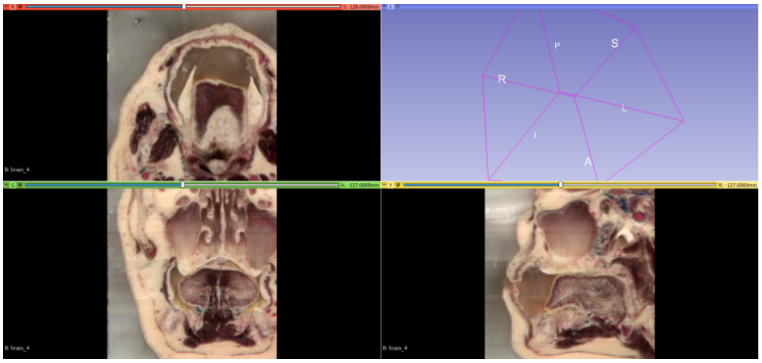

Fig. 8. Input Volume 1

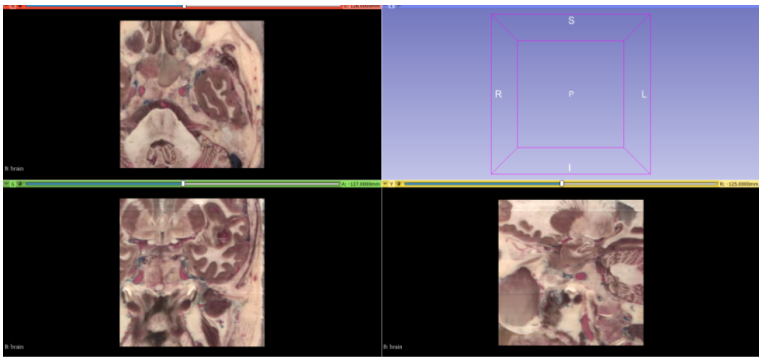

Fig. 9. Input Volume 2

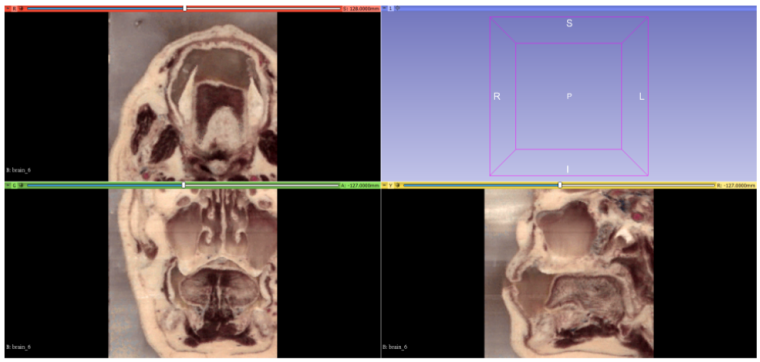

Fig. 10. Output

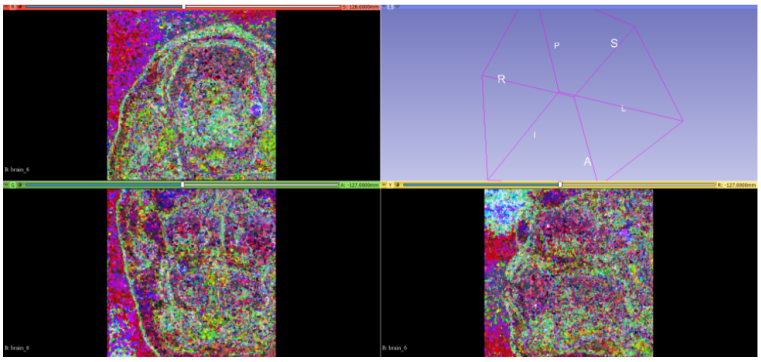

Fig. 11. ANN

6 RESULTS OF RECONSTRUCTING ORIGINAL VOLUME FROM ROTATED AND SKEWED VOLUME

Volume 1 is original volume, volume 2 is rotated by 15 degrees along the height and skewed by 2% along depth. Algorithm creates output volume which is a reconstruction of volume 1 from volume 2.

(1) Patch size = 2
(2) Iterations = 5
(3) depth = 256; height = 256; width = 256
(4) Patch Match Elapsed time: 394.725 s
(5) Map Back Elapsed time: 5.54302 s
(6) Total time Elapsed time: 400.283 s
(7) Per pixel error is 6.17076

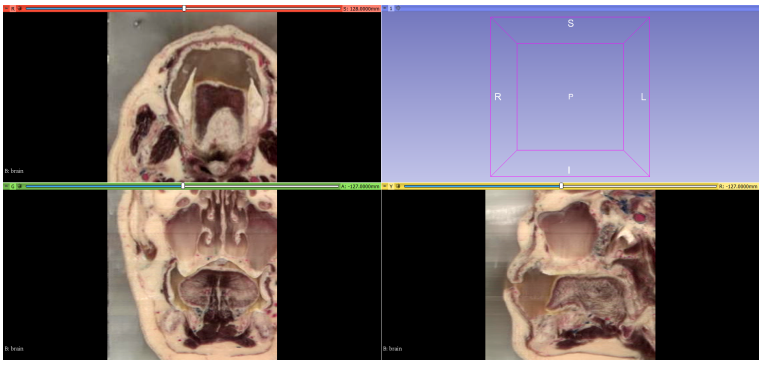

Fig. 12. Input Volume 1

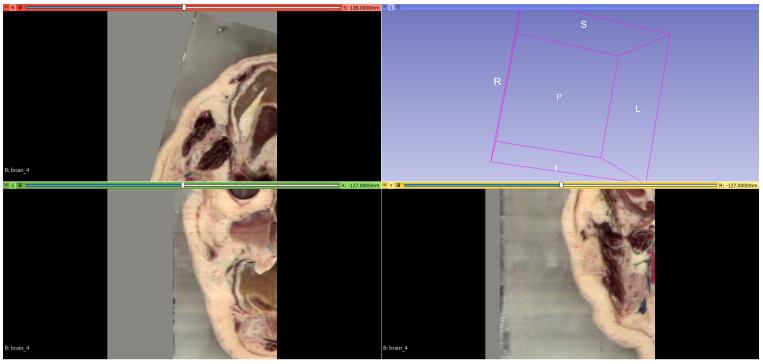

Fig. 13. Rotated and Skewed Input Volume 2

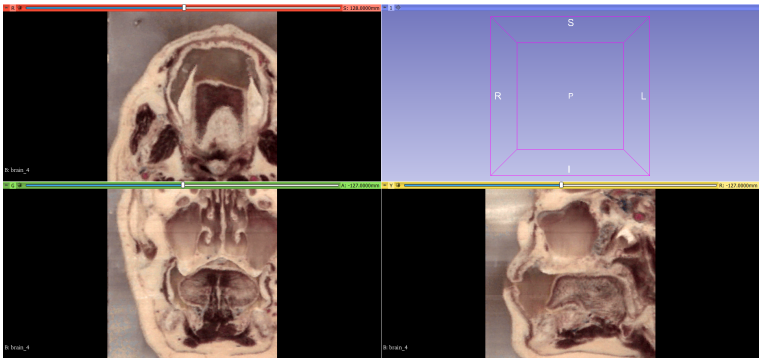

Fig. 14. Output

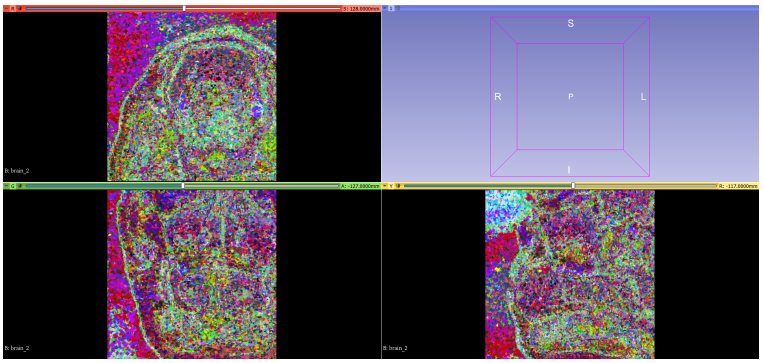

Fig. 15. ANN

7 CONCLUSION

We were able to complete the serial version of patchmatch algorithm in 3D. Alongside, we parallelised the core step of patchmatch algorithm i.e. propagation and randomisation. This allowed us to get a speedup of 207.9259 for patch size 2 and 5 iterations on the GPU workstation for the algorithm. The comparison of time taken, error on GPU and CPU for different patch sizes is shown in following graphs.
On increasing the number of iterations to 20 we get comparable error as CPU with less time. It is because we are using double buffer in GPU which takes more time to converge compared to searching through scanline (top left to bottom right and reverse) on CPU. In double buffer the updation of best patch happens after all patches have checked their best match. Without double buffer updation happens side by side. Therefore it converges within less iterations.

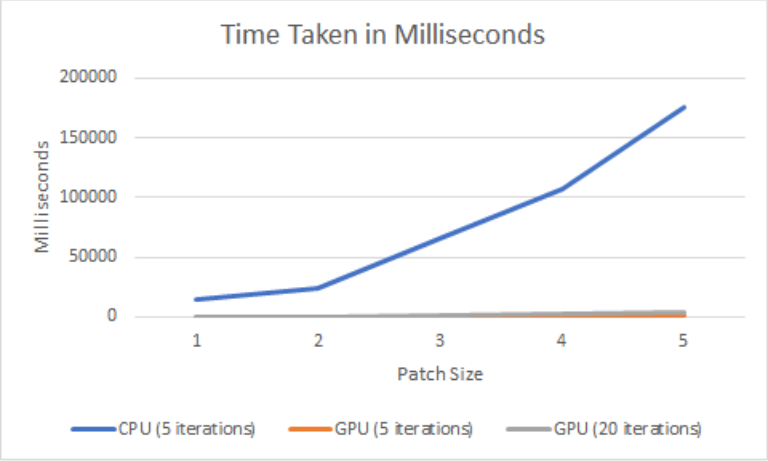

Fig. 16. Variation of Time CPU vs GPU.

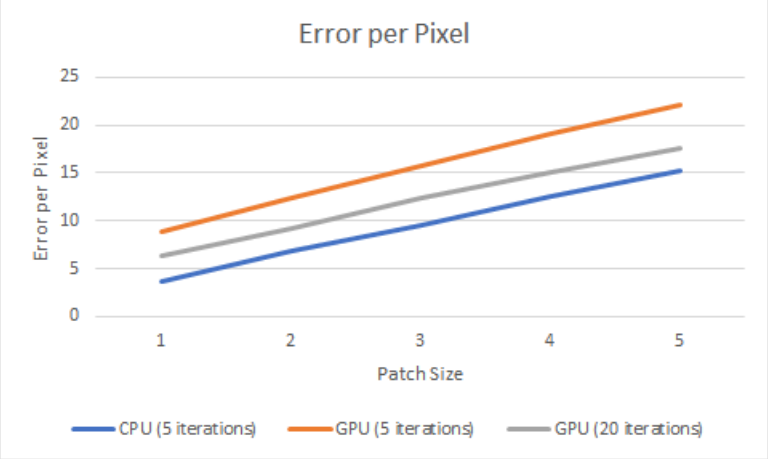

Fig. 17. Variation of Error per Pixel CPU vs GPU.

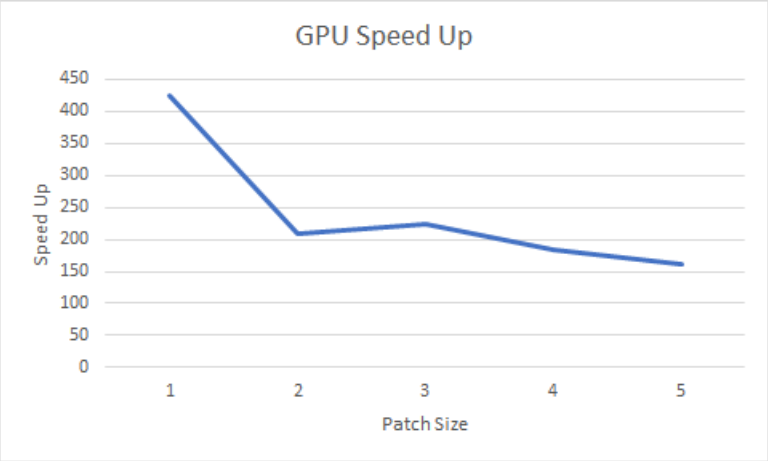

Fig. 18. Speed up on GPU with patch size

8 MILESTONES

We identified the following milestones to be covered through the course of the project.

S. No.	Milestone	Member
	Mid evaluation	
1	Understanding Input-Output format, obtaining input samples and integrating corresponding libraries for 3D objects	Pragya
2	Implementing Patch-Match Algorithm for 3-Dimensional inputs on CPU	Vibhu
3	Analysing the result of patch match on 3D images in terms of time taken and error between source and target images by changing patch size to understand the parallelisation strategy on GPU.	Pragya
4	Drawing task dependency graph for parallel algorithm	Vibhu
	Final evaluation	
5	Implement Patch Match algorithm on GPU	Pragya
6	Analyse bottlenecks and speed gain on GPU	Pragya and Vibhu
7	Extend Patch Match CPU version for style transfer	Pragya
8	Extend Patch Match GPU version for style transfer	Vibhu
9	Analyse bottlenecks and speed gain on GPU for the application	Vibhu
10	Document the code and approaches	Pragya and Vibhu

9 INDIVIDUAL CONTRIBUTIONS

Individual contributions are same as that written in Milestones. Both of us worked collaboratively and tried to contribute equally in each section. Contribution in code and analysis is 50-50% for both Vibhu and Pragya.

REFERENCES

[1] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. 2009. PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing. *ACM Transactions on Graphics (Proc. SIGGRAPH)* 28, 3 (Aug. 2009).

[2] Z. Cai, C. Wang, C. Wen, and J. Li. 2015. 3D-PatchMatch: An optimization algorithm for point cloud completion. In *2015 2nd IEEE International Conference on Spatial Data Mining and Geographical Knowledge Services (ICSDM)*. 157–161. https://doi.org/10.1109/ICSDM.2015.7298044

[3] Bradley Lowekamp, David Chen, Luis Ibanez, and Daniel Blezek. 2013. The Design of SimpleITK. *Frontiers in Neuroinformatics* 7 (2013), 45. https://doi.org/10.3389/fninf.2013.00045

[4] Guodong Rong and Tiow-Seng Tan. 2006. Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games* (Redwood City, California) *(I3D '06)*. Association for Computing Machinery, New York, NY, USA, 109–116. https://doi.org/10.1145/1111411.1111431