15-418/618 Final Project F21:
Final Report

Name: Ziying He
Andrew ID: ziyingh

Name: Zibo Gong
Andrew ID:zibog

1 Summary

We implemented panorama stitching of two images in OpenMP on the CPU and in CUDA on the GPU. We developed our algorithm from scratch and implemented every step including keypoints detection and descriptor extractor using SIFT, descriptor matching, and homography matrix calculation. We successfully outputted stitched images of great quality and achieved high speedup performance on different platforms and machines.

2 Background

Panorama stitching is a process that uses multiple images taken of the same scene and "stitched" together to create a larger cohesive view of the scene. The key data structures are vectors of *keypoints* from both images, matrix of *descriptors* from those keypoints, and the homography matrix. Keypoints are points of interest on an image. They are spatial locations, or points in the image that define what is interesting or what stand out in the image. These keypoints are used to obtain descriptors. A descriptor contain the visual description of a keypoint and is used to compare the similarity between image features. These descriptors are matched together. Then, a homography matrix is constructed and used for warping the images together and produce the final output image. Thus, the algorithm inputs two images that are of a scene with some overlap between them and outputs one image that combines the two inputs together. The specifics of the algorithm consists of the following steps: As an example, consider the two initial images:

2.1 Detect Keypoints

SIFT is used to to detect keypoints and extract descriptors. First gaussian pyramids are created. Below is a gaussian pyramid with 5 levels.

Next, Difference of Gaussian (DoG) is used on those gaussian images.

Then, these images are searched for their local extremas over scale and space. The resulting set of local extremas are keypoints.

2.2 Keypoint Localization

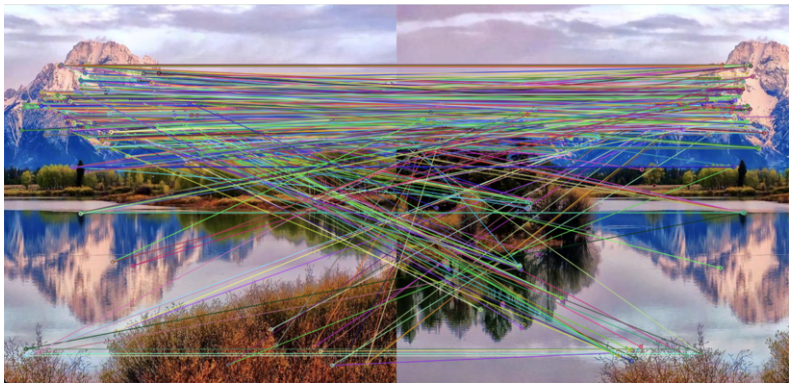
Next, the keypoints are filtered for a more accurate representation over a contrast threshold (0.3) and edge threshold (10). Below are the found keypoints for the two images, represented in red dots.

2.3 Create descriptors

From the keypoints, descriptors are obtained by searching pixels (16×16) around the keypoints, the maginitude and orientation of those pixels are obtained. Adding magnitude of related neighborhoods, we can finally get a vector with 128 elements which is the descriptors of the keypoints.

2.4 Match Key Points and Descriptors

Next, match the descriptors between two images using their distance. A brute force matching algorithm is used. Below are results from the match where each line represents a match between descriptors from the two images.



2.5 Estimate Homography Matrix with RANSAC

Then, create a homography matrix that computes the alignment between the images. RANSAC is mainly used in this step. A homography matrix is a mapping between two image planes.

2.6 Project onto Surface and Blend Images

Finally, project each image onto some surface and warp them together to create the final panorama.

In this whole pipeline each step is depended on its previous step. For example there exists the dependency that keypoints must be found first before extracting descriptors for each keypoint. Also, homography matrix can only be computed after descriptors are found. Between steps of the algorithm, it is challenging to parallelize. However, within each step, there are lots of room for parallelism. For instance, in the keypoints detection step, the creation of gaussian pyramid has no data dependencies since convolutions can be done in parallel. There is spatial locality where finding keypoints most likely depend on their neighbors. Thus, running this algorithm on a multi-core processor may result in good speedup.

3 Approach

3.1 Sequential Implementation

Our first version of the algorithm was the sequential version. OpenCV contains many image processing functions that were already implemented and ready for use. Thus, we used openCV functions to create a sequential version with the steps done. After getting a good idea of how the algorithm runs and looks like, we created our own sequential versions of each function for the baseline code. We actually changed our sequential implementation many times also since we were considering pros and cons of different algorithms and their implementation and considering taking advantage of parallelism in the future work. We've already established that we should get keypoints by using Guassian pyramids and difference of gaussians. However, there were many algorithms for getting descriptors, such as BRIEF or SIFT. We chose SIFT because the

algorithm could produce better results since it was scale-invariant and rotation-invariant. However, BRIEF was faster in terms of speed, but we chose accuracy over speed. Next, we followed through with the SIFT algorithm by following the SIFT paper. However, one part we did not fully implement was the orientation assignment but we revised the algorithm into a simple version, which still works well. Next, we chose to use a brute force descriptor matching algorithm over FLANN (nearest-neighbor matching) since it was simple. Then, we used RANSAC for calculating the homography and finally warp the resulting image. After timing the sequential code, we realized several places where it takes most of the time: 1) finding keypoints using gaussian pyramids, 2) brute force matching algorithm and 3) homography matrix calculation using RANSAC. Though we are heading for optimizing all steps of whole pipeline, our parallelization get more focused on those three parts.

3.2 Parallelization 1: OpenMP

Our first platform chosen is CPU, where our implementation was to utilize OpenMP API. The main reason we decided to use OpenMP was because in computer vision tasks images can always split into independent blocks where data parallel model (OpenMP) is very suitable for these kinds of problems. Specifically in our project when we were coding the sequential version of the code, we were considering how to split the problem so that each thread can joyfully do its own job without much communication.

3.2.1 Gaussian Pyramid

The Creating gaussian pyramids require convolving a gaussian filter with an image. We were considering parallelism in two separate directions. The first is parallelism in images; Splitting a image into blocks and do convolution for each image block is expected to get good parallelism speedup. But the problem lies in extra communication cost for edge pixels of the image block which need to grab other threads' pixels to complete convolution. But this problem can be solved by giving threads the whole (or the neighborhood) image information (i.e. global variables). The second parallelism direction is towards convolution process. Since the convolution is just multiplication of pixels where those multiplication are all independent, good performance is predictable for no communication is needed whatsoever, but the problem is that the covolution task is so trivial and it may lead to great workload imbalance and extra overhead. Image set mentioned in **section2 Background** are used and We compared speedup results as follows (4 threads used in this case):

	Speedup
Parallelism in images	3.983
Parallelism in convolution	2.231
Mixture	3.371

Obviously, we picked the first method as our final implementation.

3.2.2 DoG

Since DoG is subtraction between adjacent gaussian pyramid, creating DoGs is not time consuming at all. The only paralleism lies in pixels which can do subtraction independently, but this is not expected good speedup since the workload is so trivial so that overhead for creating a thread may be relatively heavy.

3.2.3 Detect Keypoints

Similar to last section, keypoint detection is a trivial work and the only paralleism lies in pixels. But unlike DoG, workload of each thread is largely unbalanced so the speedup is expected to be very limited.

3.2.4 Descriptor Extraction

Since keypoints are obtained by previous steps, we can not only parallize in pixels but also in keypoints. There are thousands of keypoints and for each keypoints work flow is almost the same. However in pixel direction, this task is only considering neighbors of the keypoints so the task may be trivial and has problem as mentioned. We tests both methods and results are as follows (4 threads are considered):

	Speedup
Parallelism in keypoints	3.361
Parallelism in image	2.763
Mixture	3.032

Obviously, we picked the first method as our final implementation.

3.2.5 Descriptor Match

Since we used brute force way to match the keypoint descriptors, we parallize the task in keypoints so that each thread is expected to do similar amount of work the speedup is supposed to great.

3.2.6 Homography matrix Calculation

Unlike other steps, we parallize this task in iterations (RANSAC is an iterative algorithm updating the result until its convergence).

3.3 Parallelization 2: CUDA

Our second platform choice is GPU. We implemented CUDA version for all steps we take to get a stitched image. Since OpenMP and CUDA are similar and they are both data parallel programming model, we followed the similar way as we did in OpeneMP part. However, there are also several points we are paying attention to. For example for guassian pyramid, unlike CPU where we access global variables to avoid communication, it will generate bottleneck if we access the global very frequently on CUDA device. So in order to maximize our speedup we tried to use shared memory. Moreover we tried different block size to determine one with the best speedup.

4 Results

We used two machines to test performance.

Name	Hardware	Model
Machine 1 (PSC)	CPU	two AMD EPYC 7742 (64 core)
Machine 2	GPU	GeForce RTX 2070Super

4.1 Data Set

Two sets of images are used to demonstrate the results.

Image Set 1

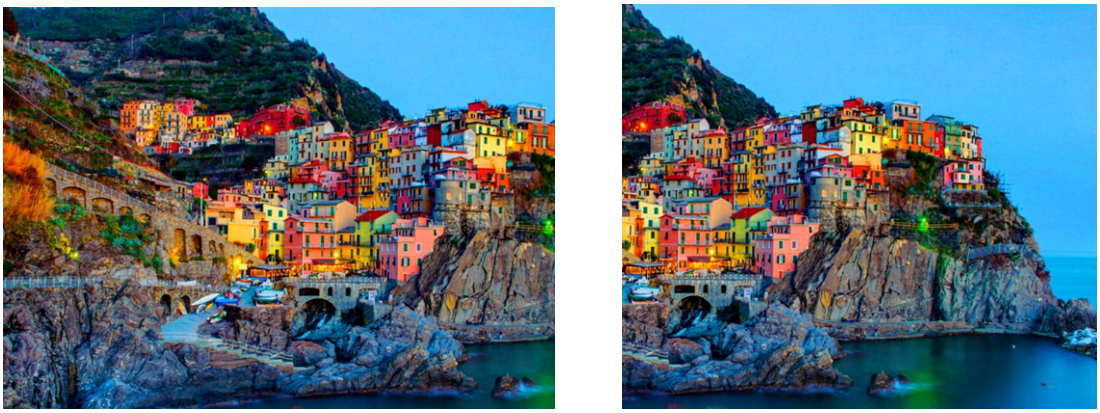


Image Set 2

Image Parameters:

	Image #1 size	Image #1 keypoints	Image #2 size	Image #2 keypoints
Image Set 1	675 ×700	3357	675 ×700	4110
Image Set 2	1474 ×1073	9158	1270 ×1080	6291

Notice that image set 1 contains smaller images with less keypoints while image set 2 contains larger images with more keypoints.

4.2 Stitched Results

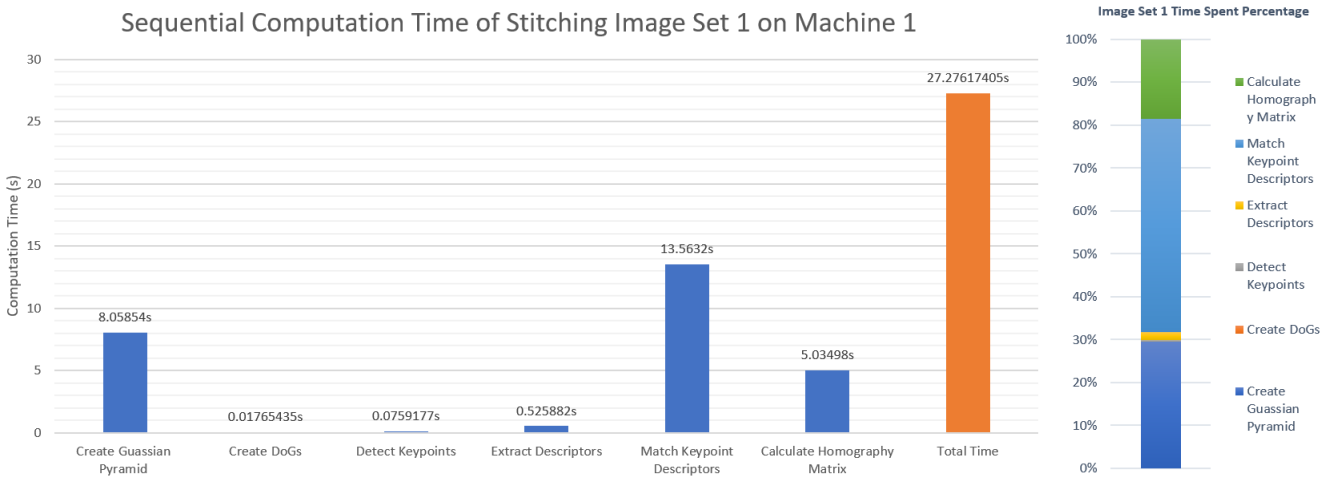
Image Set 1

Image Set 2

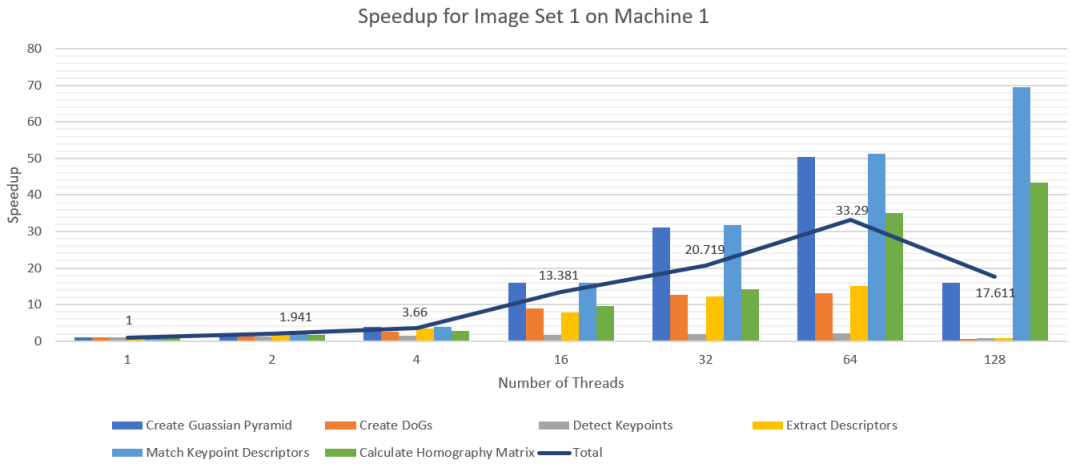
As can be seen the two image sets are stitched together to create a panorama view.

4.3 OpenMP Results

4.3.1 Image Set 1



As shown in the sequential time, the most time-consuming tasks are Guassian Pyramid Creation, Keypoints matching and Homography Matrix Calculation occupying 29.5%, 50% and 18.5% of total time correspondingly. We test performance for every steps of panorama within different threads using OpenMP shown in histogram below and a total speedup shown as a line chart below.

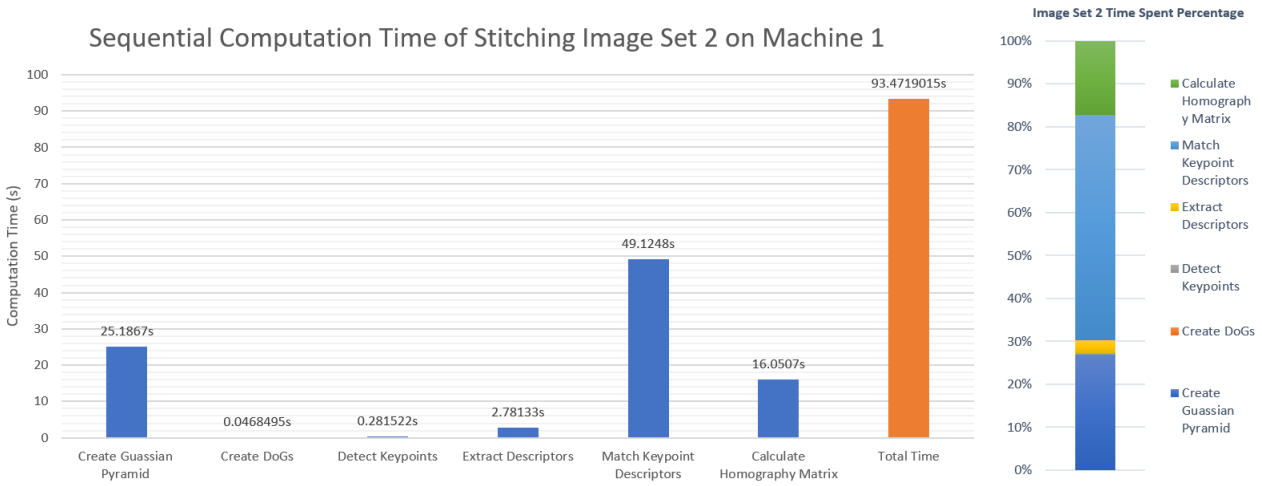


As demonstrated in the graph above, the speedup is mainly about gaussian pyramid, keypoints matching and homography matrix which are also the most time-consuming three steps in our pipeline. Specifically, the speedup for these three steps are as belows:

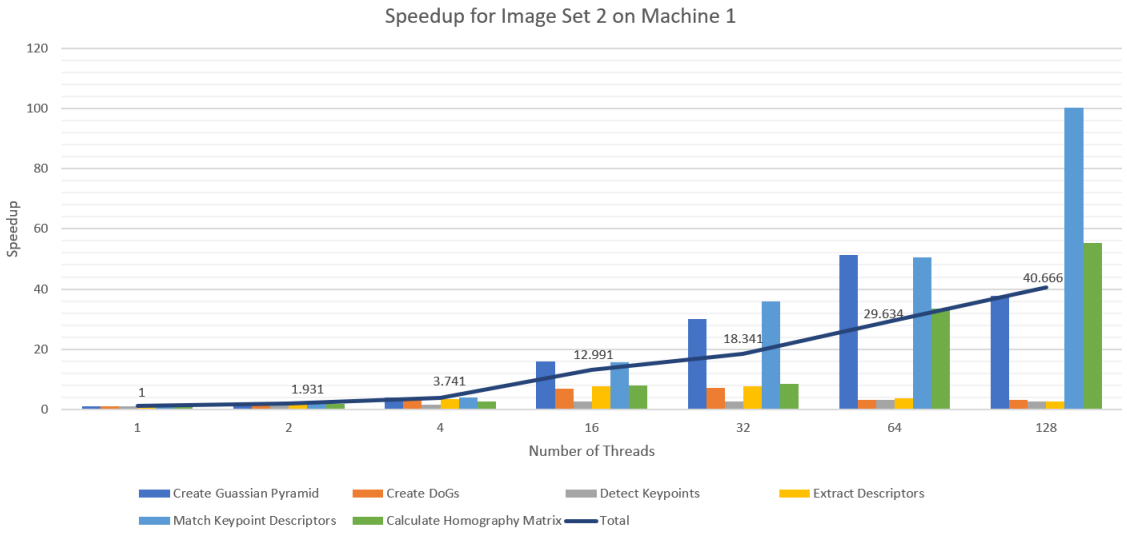
number of threads	1	2	4	16	32	64	128
gaussian pyramid speedup	1	1.997	3.983	15.909	31.031	50.425	15.962
matching keypoints speedup	1	1.999	3.984	15.987	31.764	51.275	69.555
homography matrix speedup	1	1.589	2.802	9.5	14.189	34.965	43.405

Those three steps generally have good speedup with threads increasing except that for gaussian pyramid with 128 threads the speedup slopes down since the task for each thread becomes trivial causing low computation-communication ratio. Also since Machine 1 has two separate 64-core CPU rather than one 128-core, communication between these two CPU also need taking into account. Moreover we are seeing that there are several steps that are not benefiting from the increasing cores very much. Those bottleneck are specifically discussed in **section 3.2**. Generally it's due to the fact that they are all trivial tasks and the communication cost and overhead are high with increasing threads.

4.3.2 Image Set 2



For image set 2, the percentage of computation time is similar to the first image set, where the gaussian pyramid takes 27.0% of total time and the keypoint matching takes 52.55% of total time and the homography matrix calculation takes 17.17% of total time. Speedup performance is also tested like image set 1.



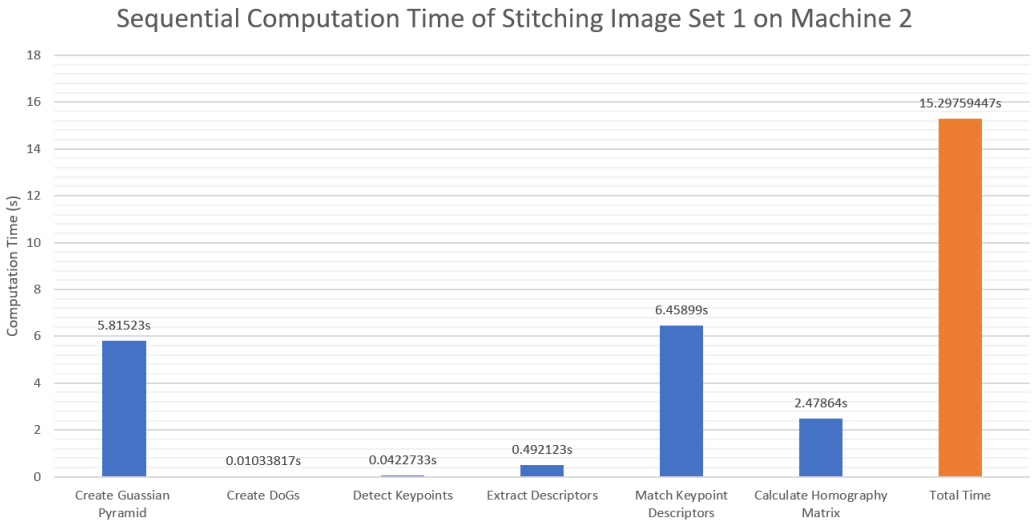
Specifically focused on the most three time-consuming tasks, we can have a great speedup with threads increasing.

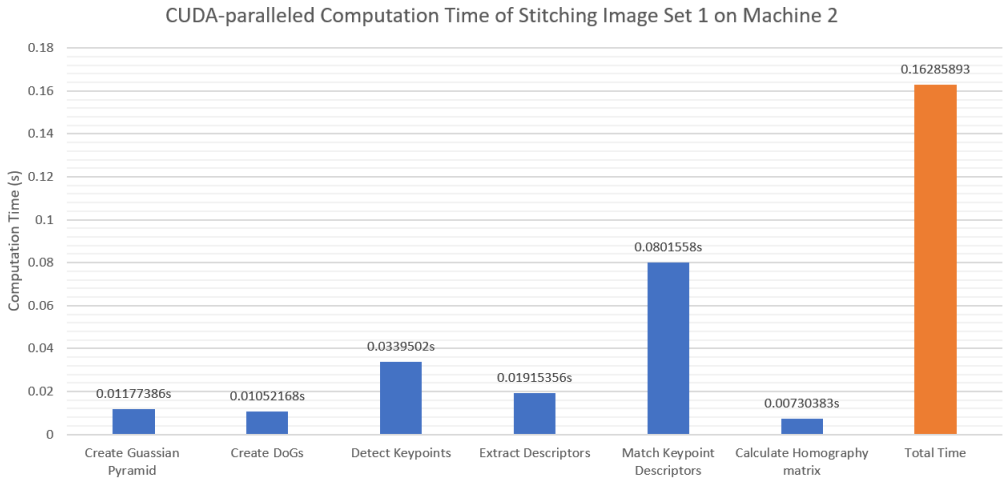
number of threads	1	2	4	16	32	64	128
gaussian pyramid speedup	1	1.999	3.985	15.801	29.981	51.282	37.709
matching keypoints speedup	1	1.999	3.993	15.694	35.764	50.644	100.255
homography matrix speedup	1	1.707	2.589	7.932	8.448	33.359	55.3472

Unlike image set 1, total speedup still boosts for 128 threads. That's because image set 2 is of higher resolution so for gaussian case the task will not deteriorate too much. Also image set 2 has more keypoints so the matching keypoints will benefit more than image set 1.

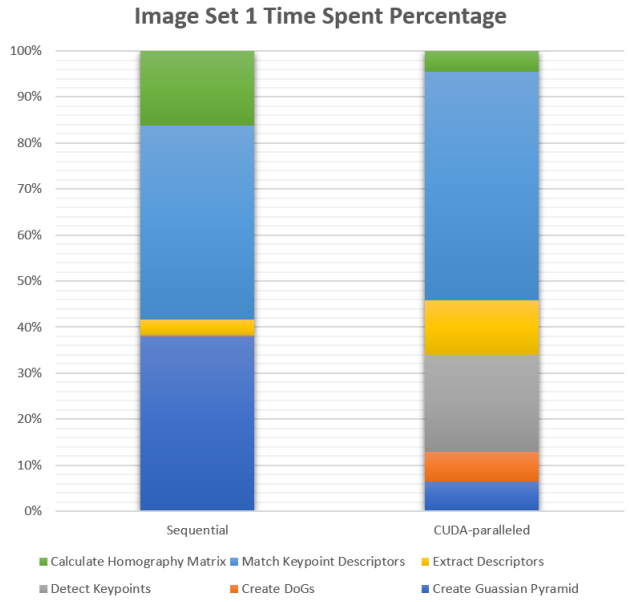
4.4 CUDA Results

4.4.1 Image Set 1

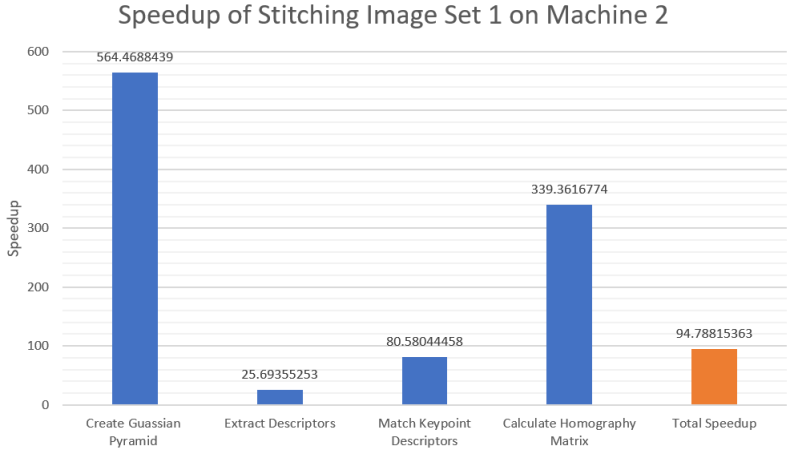




Comparing the computation time between sequential and CUDA-paralleled implementations, we see huge speedup in terms of the most time consuming steps.

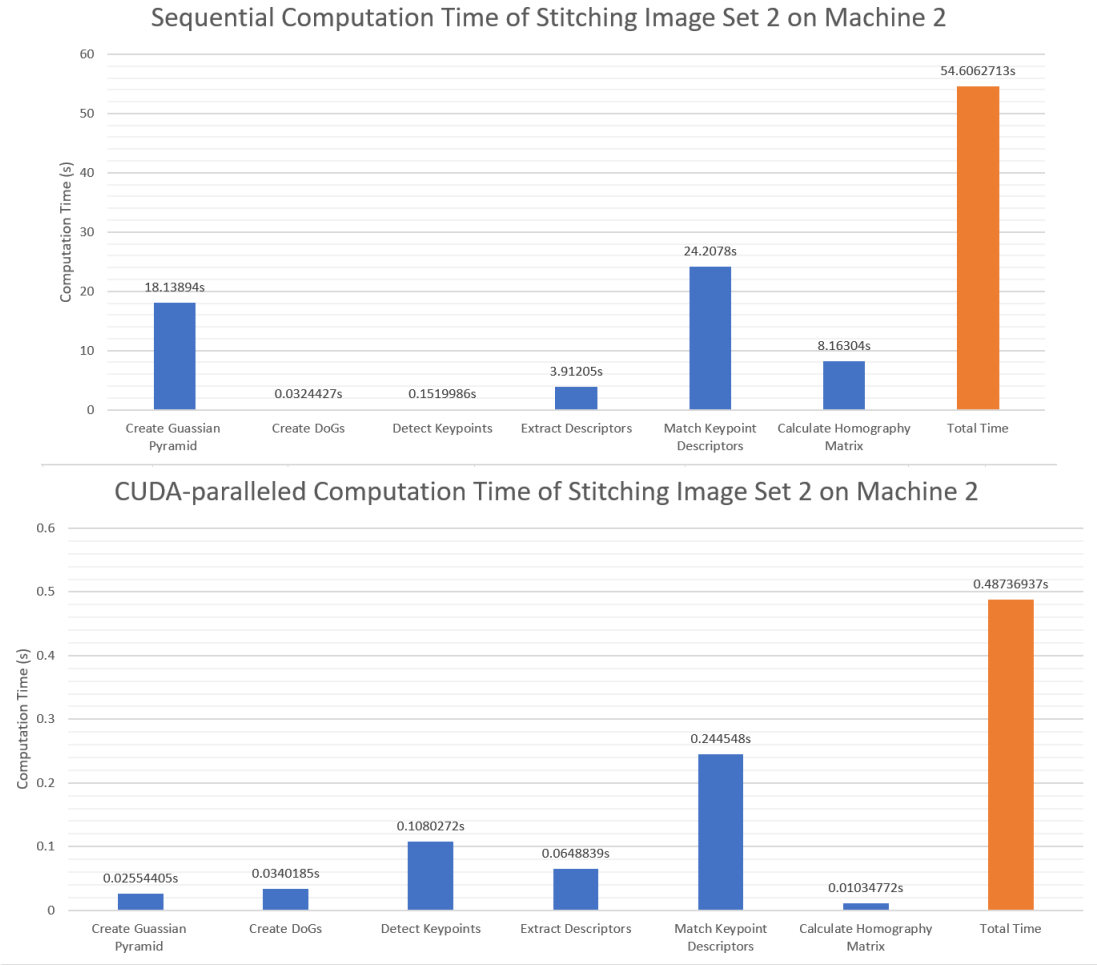


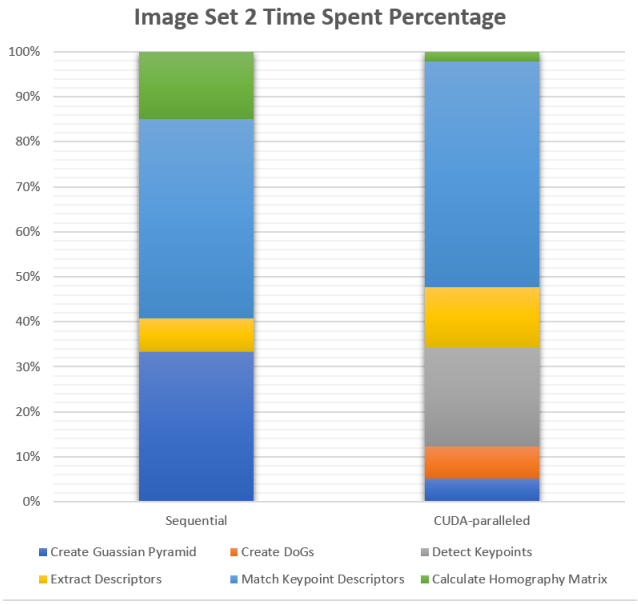
Using our CUDA implementaion, the gaussian pyramid and homography matrix calculation is not bottleneck at all.

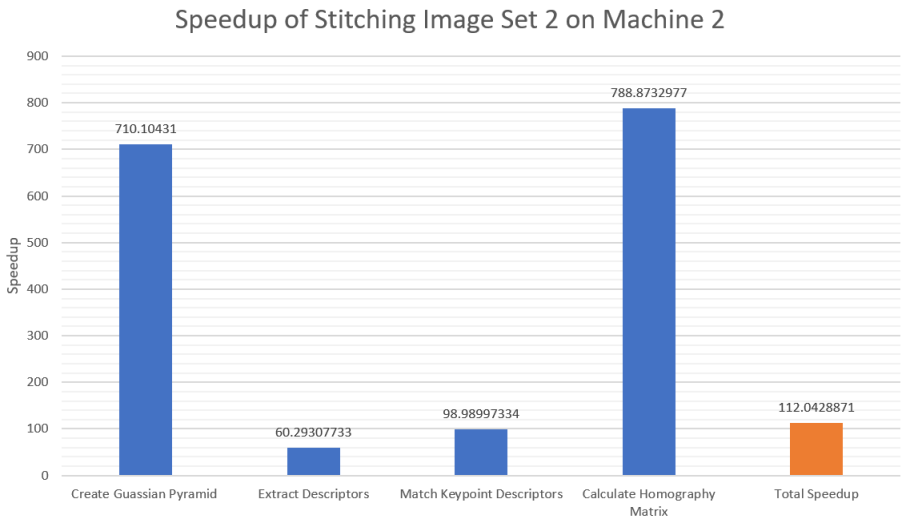


Some of our steps have great speedup but some steps are not. In matching keypoints case, since our implementation unavoidably reads global variables, this becomes a bottleneck in contrast with the OpenMP implementation. Similar problem still exists when extracting descriptors, the task for each thread is trivial so the workload easily gets unbalanced leading to a non-ideal performance.

4.4.2 Image Set 2







Similar Results are shown in image set 2. But since the images have higher resolution and more keypoints the speedup is higher than that in image set 1.

5 Distribution of Work

Ziying: 50%
Keypoint detection (OpenMP & CUDA), Keypoint localization (OpenMP & CUDA)
Zibo: 50%
Extract descriptors match (OpenMP & CUDA), Calculate homography matrix (OpenMP & CUDA)

6 References

http://6.869.csail.mit.edu/fa12/lectures/lecture13ransac/lecture13ransac.pdf

https://itzone.com.vn/en/article/image-stitching-the-algorithm-behind-the-panorama-technology/

http://vision.stanford.edu/teaching/cs131_fall1718/files/07_DoG_SIFT.pdf

http://www.micc.unifi.it/delbimbo/wp-content/uploads/2011/10/slide_corso/A34_keypoint_descriptors.pdf

https://docs.opencv.org/3.4/da/df5/tutorial_py_sift_intro.html

https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf

https://stackoverflow.com/questions/3149279/optimal-sigma-for-gaussian-filtering-of-an-image

//http://www.cse.psu.edu/~rtc12/CSE486/lecture10.pdf

https://www.cs.utah.edu/~srikumar/cv_spring2017_files/Keypoints&Descriptors.pdf

https://www.cs.toronto.edu/~mangas/teaching/320/slides/CSC320L10.pdf

https://medium.com/analytics-vidhya/a-beginners-guide-to-computer-vision-part-4-pyramid-3640edeff

https://www.cs.utexas.edu/~grauman/courses/fall2009/papers/local_features_synthesis_draft.pdf

http://6.869.csail.mit.edu/fa12/lectures/lecture13ransac/lecture13ransac.pdf

7 Supplement Results