Parallel Image Stitching

James Li, Jesse Chan

Summary

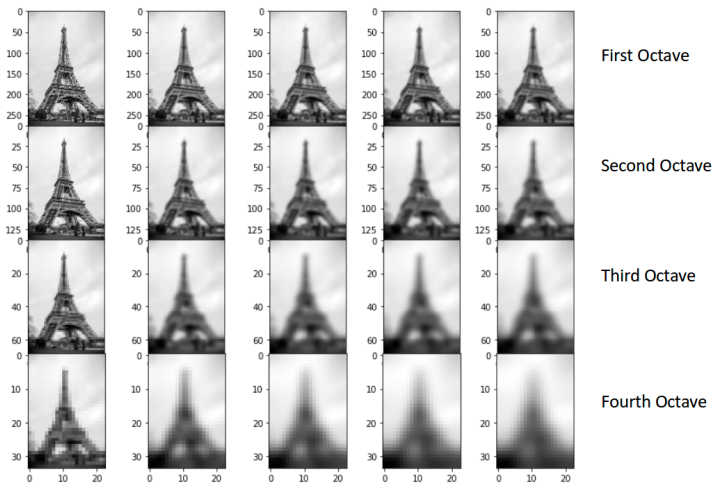
This project will explore the level of parallelism that can be applied to various parts of the image stitching workflow. An image stitching workflow typically comprises finding keypoints in images, matching keypoints, finding homographic transforms, and warping and stitching the images. To improve the quality of the output, an additional step was added to the beginning of the workflow, which projected images to a cylindrical space. A serial image stitching program was implemented detailing these steps using C++, and each step was then parallized using OpenMP in a parallel version. The cydrincical perspective transformation was parallelized across the image dimension using OpenMP. Keypoints in images were found using the scale-invariant feature transform (SIFT) algorithm and parallelized across the image dimension using OpenMP. Keypoints matching was done using a Fast Library for Approximate Nearest Neighbors (FLANN) matcher and also parallelized across the image dimension also using OpenMP. Homographic transformation was also parallelized across the image dimension using OpenMP. Warping and stitching was parallelized across image pair dimensions using OpenMP. Performance between the serial and parallel version of the image stitching program was then compared. Our parallel implementation using OpenMP running on 8 cores was able to achieve a 2.76x speedup compared to the serial version stitching a panorama of 8 images.

Background

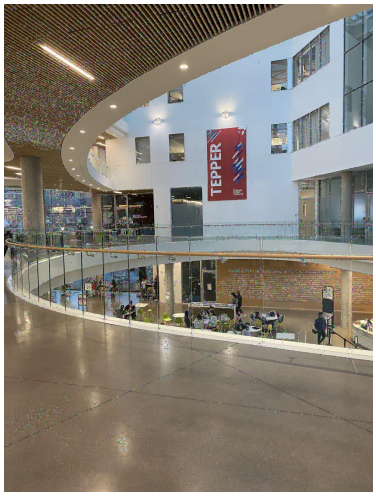
To stitch two images together, we need to roughly follow the following workflow:

1. Find important keypoints and descriptors in images
2. Match these keypoints and descriptors across images
3. Find homography transform matrix using matching keypoints and descriptors
4. Warp and stitch images together using transform matrix

The image stitching workflow begins by identifying key features in the images. Key points need to be detected in the images and this is commonly done using feature extractors such as SIFT, SUFT, and ORB. We briefly compared and contrasted the different feature extractors to determine the algorithm that could benefit the most from parallelism, and opted into using a SIFT algorithm for our keypoint matching algorithm because SIFT consistently provides better keypoints than the other algorithms albeit at the cost of slower execution. As a result, we believe SIFT could benefit the most from parallelism speedup, especially considering how SIFT works. SIFT first works by applying a gaussian blurring and scaling reduction over images. Since each pixel in a gaussian blur or scale reduction is independent, this operation can easily be parallelized without dependency concerns; a pixel in an output gaussian blur or scaled image depends only on the neighboring pixel values in the input image.



Blurs and scales comprising octaves



"Artifacts" are circles identifying keypoints in image

Since this process is repeated many times in the SIFT algorithm in what are called octaves, there is large grounds for potential speedup here. Further along the SIFT workflow, difference of gaussian is also taken where pixels from different blurs are subtracted from one another. This is again easily parallelizable in the pixel dimension due to a lack of pixel dependency. Local pixel minima and maxima are then found using sampling windows, which is again a simple thing to parallelize across the pixel dimension. Needless to say, SIFT has a lot of potential to be parallelized across the pixel dimension.

After identifying keypoints and key features in each image, we will need to match these points and features between different images. To do so, we need to make pairs between keypoints of images so that the Euclidian distance between keypoints are minimized. We briefly explored and compared FLANN matching and Brute Force (BF) matching for this subtask to determine which algorithm can benefit more from parallelism. We initially leaned towards BF matching due to its simple nature of iterating between pairs of keypoints, which could be easily parallelized.

Having finally identified matching keypoints, the image stitching workflow then takes these matching points to calculate the homographic matrix that needs to be applied to the images such that they all align properly. The homographic matrix encompasses transformations such as rotation, translation, and perspective transformation. We preliminarily explored homography estimation with RANSAC (Random Sample Consensus) algorithm.

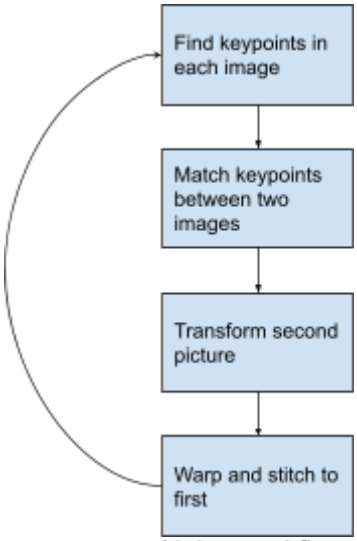
Automatic Homography Estimation with RANSAC
1. Choose number of samples N
2. Choose 4 random potential matches
3. Compute H using normalized DLT
4. Project points from x to x' for each potentially matching pair:
5. Count points with projected distance < t
6. Repeat steps 2-5 N times and choose best H

Since the whole process is repeated N times, we also preliminarily determined we could also parallelize RANSAC.

After calculating the homography matrix, the matrix needs to be applied to each image to warp the image. Applying said transformation is a simple matrix multiplication, which needless to be said is parallelizable. To finally stitch the warped images together to form a panorama, aligned images are worked on in pairs, and laid over one another. Parallelizing this is slightly less trivial because there is a dependency and order that we have to stitch pictures in using the homographic transforms. This will be touched on briefly later.

Having said all that, we see that each algorithm in the image stitching workflow has grounds to be parallelized in the algorithmic dimension, whether it be across pixels or across iterations.

However, there is another dimension of image stitching that can be parallelized. It is important to keep in mind that a very naive image stitching workflow would resemble the workflow below, where the identified procedure is repeated for many iterations, with each iteration stitching one image to the result image. As you can see, this workflow is very serial because there is strict dependency between iterations. We can only start the process to stitch image 3 after image 2 has already been stitched to image 1.



Naive workflow

This naive approach, however, has potential grounds to be parallelized in the image dimension. Keypoints for each image has no dependencies, so we should be able to calculate them for each image in parallel. Keypoints between images can also be matched between each adjacent image pair in parallel because image pairs should be independent from one another. However, because each image is bound to have a different amount of keypoints, and image pairs are bound to have a different amount of matching keypoints, we will without a doubt run into very divergent behavior between image pairs. This problem is slightly reduced considering the input images we give to the program are identical in size, so the amount of keypoints in each image is slightly bounded, but this problem is regardless not negligible. It is important to keep in mind that we will lose some potential speedup due to this divergent behavior.

The challenge aspect we will have to overcome if we want to parallelize the image stitching workflow across the image dimension is the homographic transform. While we can technically calculate each homographic transform matrix between image pairs in parallel, there is a dependency between the transforms when we want to ultimately warp and stitch all the images together at the end to a resulting image. It is important to keep in mind that our transform matrices are calculated in relation to their immediate adjacent image neighbors should we parallelize across the image pair dimension. If we applied those transformations in parallel, our output would be incorrect, because the net transformation matrix for an image depends on the transform matrix for all prior image pairs. Consider image pairs 1 and 2 and 2 and 3. When stitching the end output image, we would want image 3's alignment to be calculated in relation to image 1, which means it has dependency on its own homography transformation relative to image 2 as well as image 2's homography transformation relative to image 1. This will be a difficult thing to tackle in light of this dependency issue across the image dimension.

All in all, we have identified two dimensions to potentially parallel our program: across pixels or iterations in each sub-algorithm, or across image and image pairs.

Approach

There were many areas in our image stitching workflow where we could implement parallelism, so we prioritized analyzing the portions of code where it took the longest time and was the most complex in terms of dependencies. This is because we have two goals - to improve the speedup of the overall workflow and to learn to implement non-trivial parallelism strategies.

With that goal in mind, part of our goal during the checkpoint was to investigate which portion of the code took the longest time, and to target those sections that take the longest time to get the most speedup. We were roughly able to profile the following during our checkpoint using a python implementation of a image stitching program.

Section	Runtime	Percentage
Finding Keypoints	0.14747s	56.47%
Matching Keypoints	0.10617s	40.65%
Homographic Transform	0.00752s	2.28%

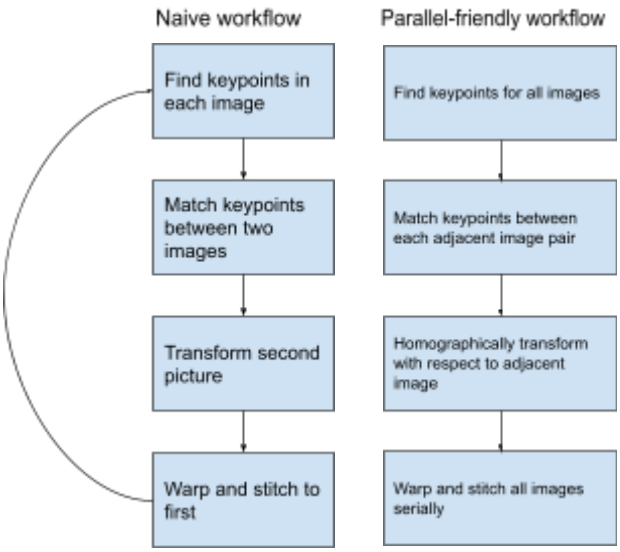
Thus, we initially believed we should focus our main effort on parallelizing the finding and matching keypoint algorithms. We also identified a key problem in our workflow. When running the workflow with more images, images which are stitched later will become more and more distorted and skewed because their perspectives are all calculated with respect to the first image. To resolve this, we opted to add a step to our workflow which preliminarily projected images to the cylindrical space, resolving this distorted perspective problem. **We referenced this cylindrical projection code from a stackoverflow post.**

Since we identified two dimensions for parallelism, we were also initially a little lost which specific dimension we should work on. Our TA suggested that the more interesting dimension to parallelize would be across image and image pairs. When we encountered library difficulties working with CUDA on our local development environment with a RTX 2070 Max-Q, and disk quota difficulties setting up OpenCV on the GHC machines, we also begun preferring the image dimension approach. Our local development environment comprised of a WSL environment without GPU support running on a Intel i7-9750H with 12 threads, so we opted for OpenMP.

The first and foremost thing we had to do was alter the serial workflow. Our initial serial workflow and profiling version was heavily serial, so we redesigned the serial algorithm so that it was more parallel friendly. Instead of identifying keypoints for an image in each iteration, we would find keypoints for all the images at once. Likewise, instead of matching keypoints for two images in each iteration, we would pre-match all image pairs and pre-calculate homographic transforms before warping and stitching. The redesigned workflow can be original naive workflow.

Our first parallelism approaches encompassed the following:

1. **Keypoint Detection**



All keypoints for images were detected in a batch. Here, we used OpenMP and shared memory to achieve this. Each image was mapped to a thread, and each thread would find the keypoints for that image. Both fixed and dynamic scheduling were attempted, but dynamic scheduling was preferred because a lot of divergent behavior was observed for findKeyPoints() when run serially. Since minima and maxima windowing can break, findKeyPoints() on different images in our input were found to take anywhere from 80-125 ms. Despite overhead, dynamic would allow us to best exploit the threads that end early so that they can be assigned another image to detect. Code implicitly synchronizes at the end where all threads have finished detecting keypoints for its mapped image.

2. **Keypoint Matching**

   With keypoints for images detected, matches between adjacent picture pairs were also made in a batch. Here, we also used OpenMP and a shared memory model. Descriptors of keypoints for the images were paired together, and each thread would be mapped a image pair to process, matching any keypoints between an adjacent image pair. Both fixed and dynamic scheduling were attempted here as well, but dynamic scheduling was vastly preferred. matchKeyPoints() varied greatly between image pairs, and execution time ranged anywhere from 19-108 ms depending on input pair. Despite overhead, dynamic would allow us to best exploit the threads that end early so that they can be assigned another image pair to match. Code implicitly synchronizes at the end where all threads have finished matching keypoints for its mapped image pair.

   Although we initially favored a BF matcher, likely due to matcher implementations and we suspect differences in how library code was being loaded into .text section of memory, we had to use a FLANN matcher instead to see actual speedup.

3. **Homographic Transform**

   Despite little absolute performance we could gain from parallelizing the homographic transform due to its relative low runtime, we regardless opted to do so. Similar to keypoint matching, all the homographic transforms were now done in a batch. Using OpenMP and a shared memory model, an image pair including both of their keypoints and common matches would again be mapped to a thread, and the thread would calculate a homography transform matrix to align the latter image to the former. A fixed and dynamic scheduling approach was found to be virtually identical in performance because runtime was only 2-13 ms, and overhead of dynamic scheduling essentially overshadowed any performance gain from dynamic scheduling. Code again synchronizes at the end where all threads have finished computing homographic transform for each image pair.

We must also now address the earlier challenge we pointed out. Although we calculated transforms in parallel earlier, they cannot just be used to warp images because they are calculated in relation to their preceding neighbor, when we really want the transformation to be relative to the first image. If stitched together, we would end up getting the wrong end result. As a result, we need to accumulate the homographic transforms as we move from left to right. We need to add the accumulated transformation to each new image, and add this image's homographic transformation to the total for the next image in our pipeline.

The serial implementation was likewise changed to reflect this necessary change where we continually accumulate the total transformation as we work from left to right to stitch the end result.

```
homographies[i].ptr<double>(0)[2] += dx;
homographies[i].ptr<double>(1)[2] += dy;
dx = homographies[i].ptr<double>(0)[2];
dy = homographies[i].ptr<double>(1)[2];
```

However, this change requires strict dependency due to having to accumulate homographic matrix as we work from left to right. The image warping and stitching was subsequently kept serial as we worked on warping and stitching images from left to right one after the other.

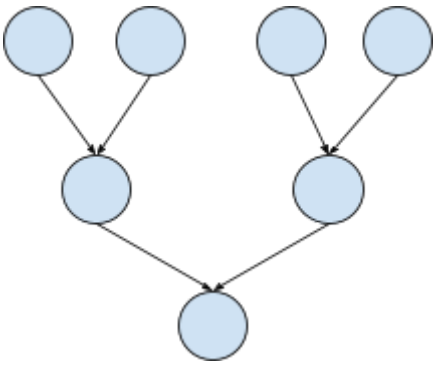
It was at this point that the code again as profile, and it was realized that non-trivial work was being done to project the images in a cylindrical space as well as the final warping and stitching.

```
Total Elapsed Time: 4487 ms
Computational Time 4194 ms
Cylindrical: 1988 ms
Keypoints: 789 ms
Matching: 83 ms
Transform: 107 ms
Stitching: 1223 ms
Cropping 0 ms
```

Profiling on 16 images with 4 threads

Parallelizing the cylindrical warping was trivial due to its similar nature to the prior parallelism. Using OpenMP and a shared memory model, an image would again be mapped to a thread, and the thread would project the image to a cylindrical perspective. A fixed scheduling approach was opted because our input images were always the same size, and the work performed by the cylindrical algorithm depended on the amount of pixels. Code again synchronized at the end where all threads have finished computing homographic transform for each image pair.

The image warping and stitching was the most challenging part to parallelize. Eventually, we opted for a top-down tree approach, or a divide and conquer approach.



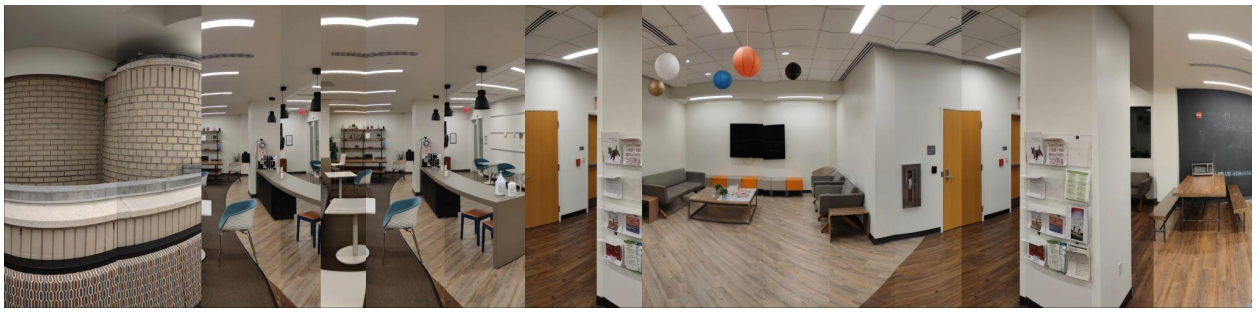
The problem encountered was that we had to find a way to eventually work our way down to accumulate all of the homography matrices when we get to warping and stitching images on the right. Since we can only warp and stitch two images together at the same time, the most efficient way is to work with pairs of image at a time. To stitch four images together, we can stitch image 1 and 2 together and image 3 and 4 together in parallel, and afterwards stitch their results together to get our end result. We would be able to eliminate the dependency image 3 has on image 1 and 2. The important thing to keep in mind with this approach is that by the time we stitch the two results together, we have to transform the second result image using image 1 and image 2's

homography transform matrix. Although we have removed a lot of dependencies, there is still a dependency of the second result image on the first.

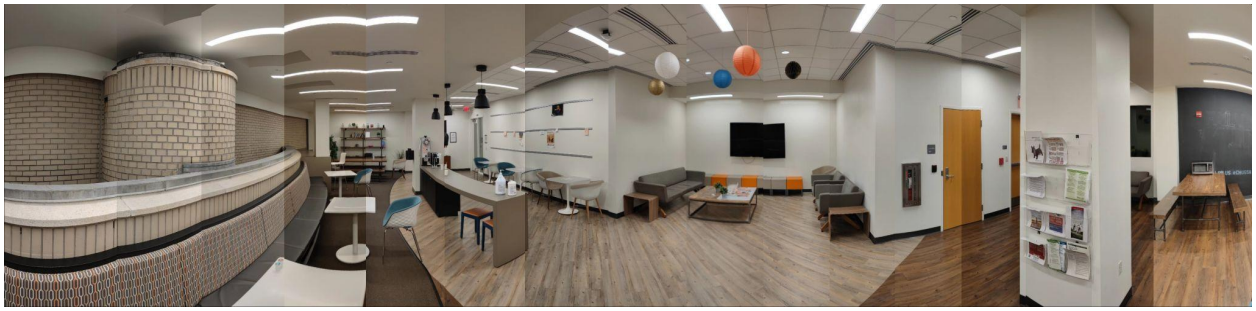
With this in mind, we set out to parallelize the warping and stitching using OpenMP. This top-down tree parallel approach effectively reduces our runtime to $log_2 N$ iterations, or O(logN). This is significantly better than the native serial runtime of N-1 stitching iterations, or O(N).

Using OpenMP and a shared memory model, image pairs that do not share images are mapped to a thread, and the thread accordingly warps and stitches the images according to their homography transform matrices. Their homography transform matrices is then written to an array for use later in the next iteration. This is repeated as many times until all images are stitched to a single image. Code synchronizes every iteration because every new iteration depends on the homography matrices that are accumulated in the last iteration. A dynamic scheduling scheme was opted because different amounts of pixels are overwritten depending where the image is stitched, so each thread can run for different amounts of time. Despite overhead, dynamic would allow us to best exploit the threads that end early so that they can be assigned another image pair to stitch.

While implementing this top-down approach, we also encountered a bug where we would occasionally run into segmentation fault or run into very erroneous output.



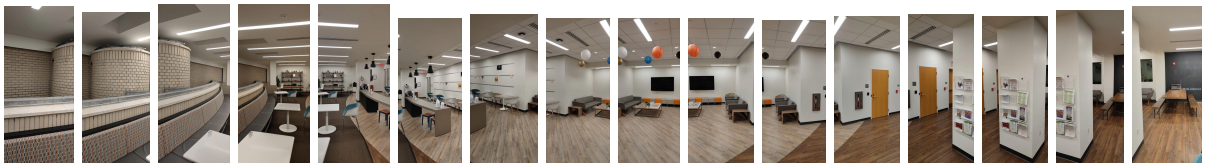
This was found out to be a racing condition where multiple threads was simultaneously reading and modifying homography matrix data, causing erroneous transformations. This bug was resolved once homography matrix writes were moved out of the omp block into a critical serial section



Do note that the coloration differences in the resulting panorama is due to a lack of blurring.

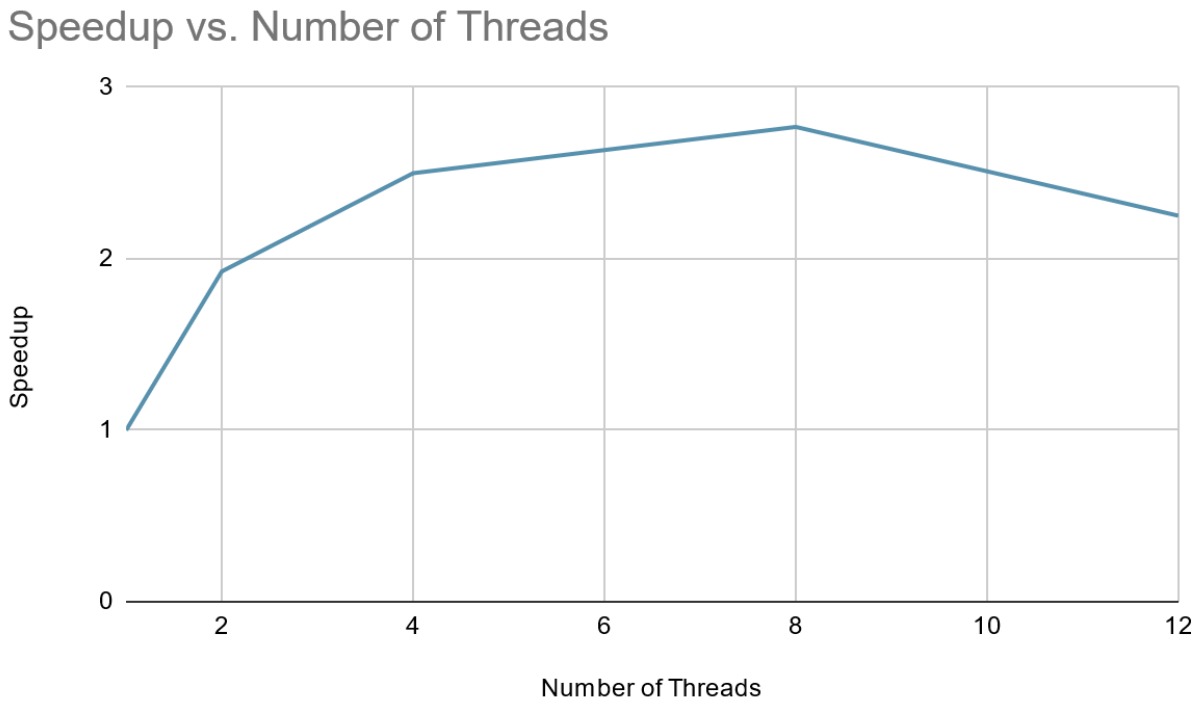
Result

We measured the performance of our parallel implementation using speedup and execution times compared to a serial implementation.



The following performance evaluation was conducted with a 12 thread machine on an input of 16 images. The baseline was the parallel implementation with the top-down approach being run with 1 thread.

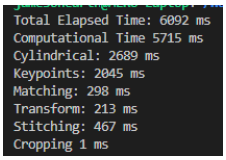
Number of Threads	Computational Time	Speedup
1	5715 ms	1x
2	2967 ms	1.9261x
4	2287 ms	2.4989x
8	2064 ms	2.7688x
12	2539 ms	2.2508x

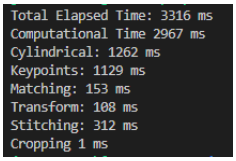


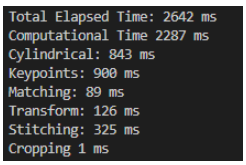
Graph 1: A chart of speedup against the number of threads. We see that the speedup peaks at 8 threads. and drops as it moves toward 12 threads.

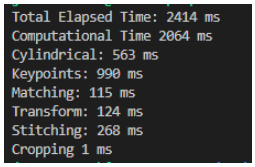
This is a breakdown of the timing of separate sections of the workflow.

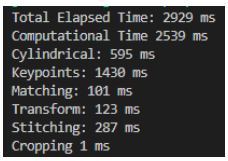
- Cylindrical: Converting image to a cylindrical form for improving visual of final combined image
- Key Points: Detection of key points with the SIFT algorithm
- Matching: Matching of key points of the adjacent images
- Transform: A pairwise homography transform for the images
- Stitching: This is our warp and stitch top down algorithm
- Cropping: Serial crop to cut out black border of result



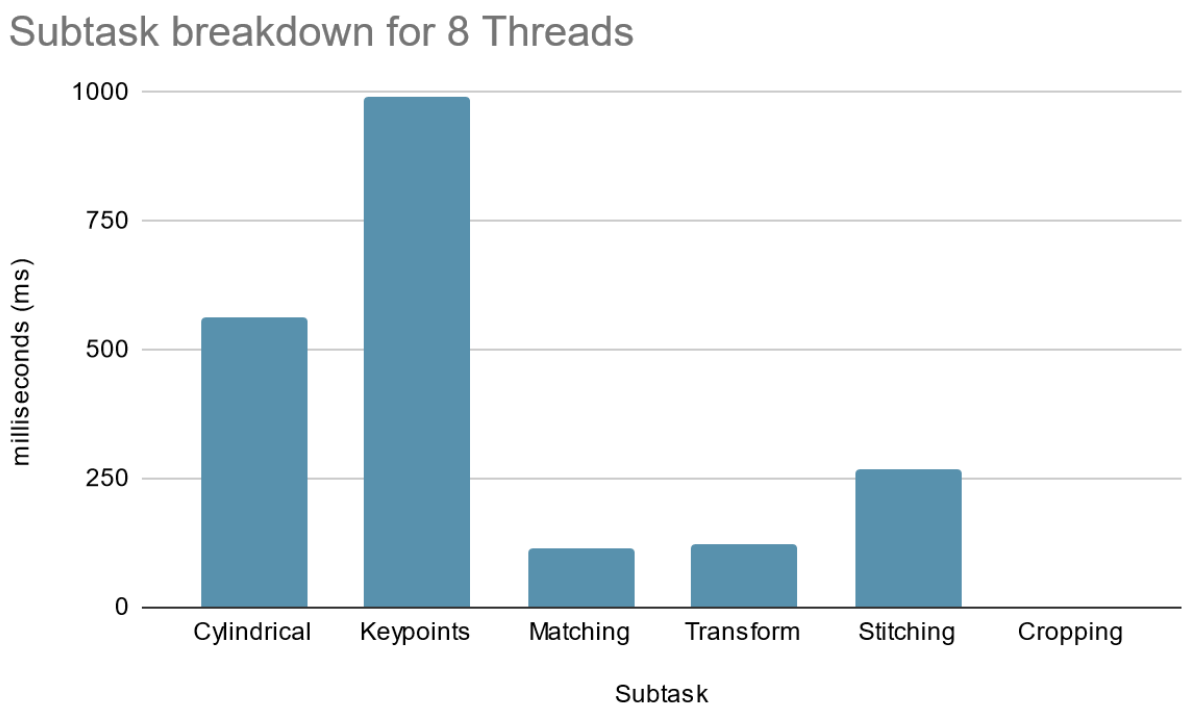




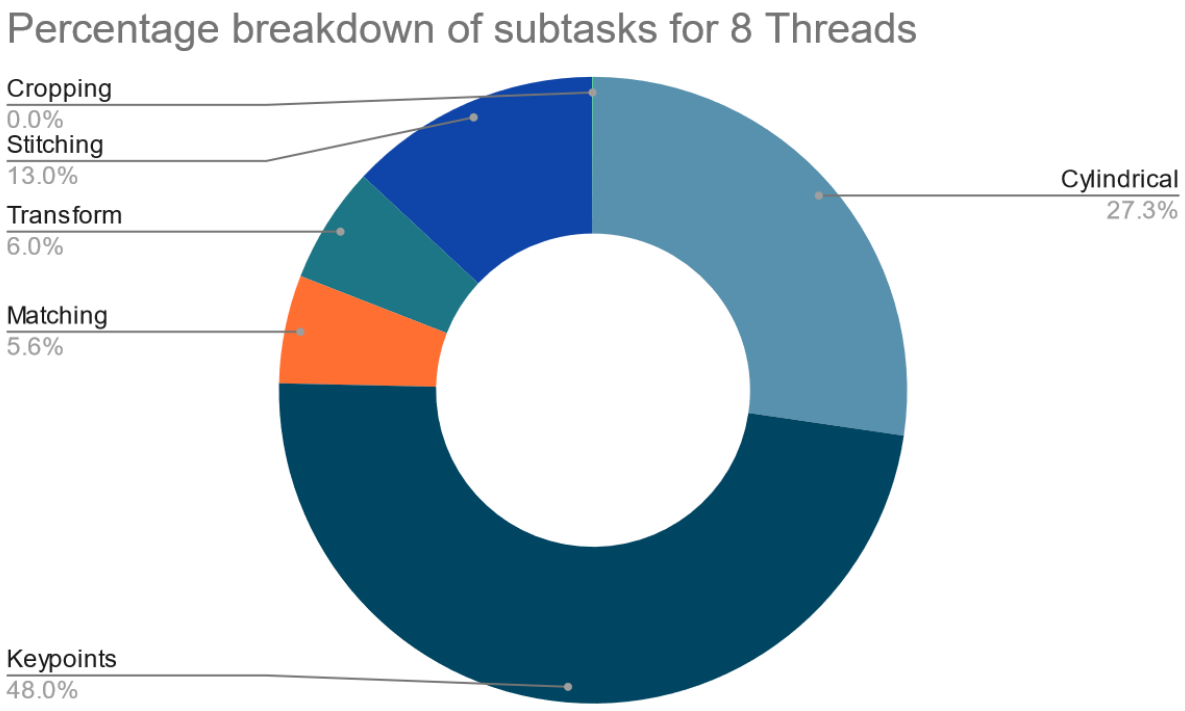




From left to right, we have provided the workflow breakdown running on 1, 2, 4, 8, 12 threads.



Just at a glance, if given more time, it would be ideal if cylindrical projection and keypoint detection could be parallelized more. This can be elaborated further in limitations, but removing OpenCV dependencies such as resize and copyMakeBorder in cylindrical projection and SIFT feature extractor should improve their execution times.



It is also interesting to specifically analyze and compare the warp and stitch approach with different number of threads. It is important to note the speedup gain of the top-down approach compared to the left-to-right approach. The baseline here in this case is a parallel implementation running on 1 thread without our top-down approach and running the original sequential left-to-right approach.

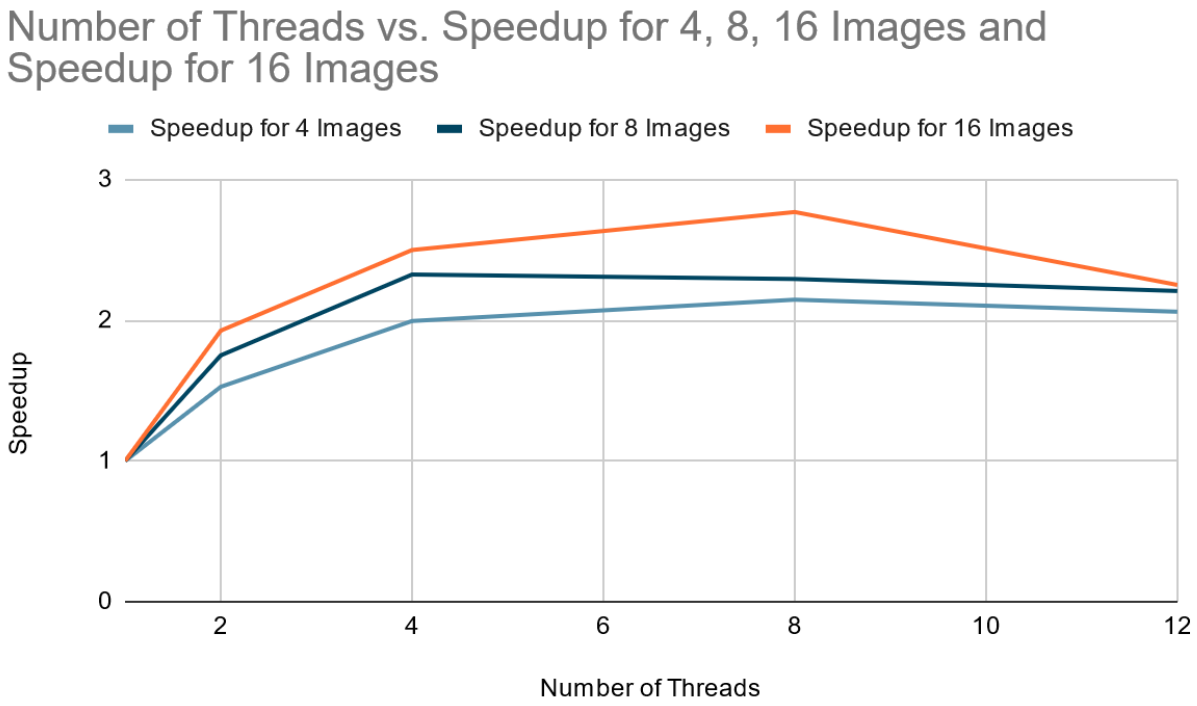
Number of Threads	Computational Time - Warp and Stitch	Speedup - Warp and Stitch
1 (Left-to-right Sequential)	1316 ms	1x
1	467 ms	2.817x
2	312 ms	4.217x
4	325 ms	4.049x
8	343 ms	3.837x

A potential dimension of problem size that can potentially affect the speedup of our program. The first one is the number of input images that needs to be stitched together. The above performance evaluation was conducted using 16 input images, but consider a 4 image input instead.

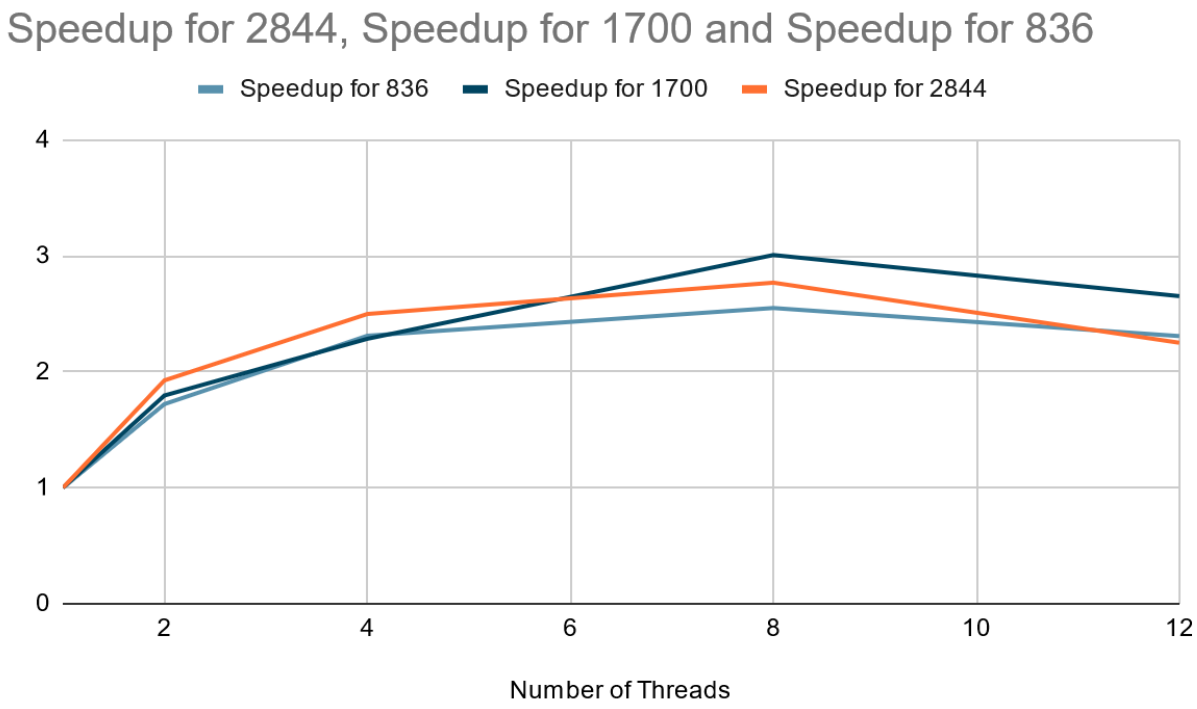
Number of Threads	Computational Time	Speedup
1	3616 ms	1x

2	2368 ms	1.5270x
4	1813 ms	1.9944x
8	1685ms	2.1459x
12	1755 ms	2.0603x

As we can see, the performance speedup has decreased quite a bit for an input of 4 images as we increase the number of threads. This is because there are less images and image pairs all the threads can work with, since there are only 4 images and 3 image pairs given these inputs. We end up wasting a lot of threads as a result. Our implementation speedup will increase the more the images in a given input. The following graph visualizes the speedup increase we get the more input images we have.



A second potential problem size that can affect the speedup of our program that we investigated is the size of the image that is being fed as input. We initially did not think it would have any effect on the speedup because the number of keypoints and matches should be relatively proportional to the image size. Since images of the same size should have the same magnitude of matches, the threads should not get more divergent just because we increase or decrease the size of the input images. We conducted this experiment by horizontal cropping images to small and mini dimensions while keeping the width the same; original images were 2844 pixels heightwise, the small dataset was cropped to 1700 pixels heightwise, and mini dataset was cropped to 836 pixels heightwise. The same images at different dimensions was then fed as input sets to the program. Our experiments confirmed our hypothesis that image size has no impact on performance or speedup gain.



As an extension of this experiment, we also suspect that the execution behavior of this program would likely not change given other similar changes such as varying the amount of overlap between images. Of course, this is all assuming that these traits are shared by all input images. If input images differed in size or image pairs differed in the overlap they shared, we would likely get more divergent behavior in the keypoint detection and matching because input images would now range wildly in terms of keypoints and matches. As a result, it is very likely that we would get worse speedup performance.

Limitations

There were several things that limited our speedup. First of all, for the warp and stitch algorithm, we needed to have 2^n images and threads, but in our case, we had 16 images and the environment we ran the code in only had a maximum of 12 threads. In addition, the speedup is partially dependent on our problem size, or rather the number of images, since we are parallelizing across images.

It is shown for 16 images, the best speedup comes from using 8 threads. It is hypothesized that this could be because of certain pairwise operations that are being performed. When we look specifically at the warp and stitch algorithm, we can see that the first iteration of the top-down gives a lot of parallel speedup, but as images are being combined together, there are less and less stitching happening after each iteration. In the last iteration, we would only have two images to be stitched, which doesn't benefit much from parallelism.

Another limitation we found is from the overhead of OpenMP, with the encapsulation of some parts of the code in the OpenCV library. Hence for each new thread, when we call a library function from OpenCV, there is an overhead of reinitialization and setup that comes from the library call. With the overhead on

each thread, this causes a bottleneck in parallelism. This is likely the biggest culprit for our poor parallelism performance and this was confirmed with a very basic analysis of the subpart speedup. At a brief glance of the breakdowns, something that uses more heavier openCV dependencies such as finding keypoints gets worse speedup than something very lightweight such as cylindrical projection, which we have implemented partly natively and doesn't call on as many heavy openCV functions.

We also see that there is divergent behavior in detecting keypoints and matching keypoint. Since minima and maxima windowing can break, findKeyPoints() on different images in our input were found to take anywhere from 80-125 ms. Despite overhead, dynamic would allow us to best exploit the threads that end early so that they can be assigned another image to detect. Both fixed and dynamic scheduling were attempted here as well, but dynamic scheduling was vastly preferred. matchKeyPoints() varied greatly between image pairs, and execution time ranged anywhere from 19-108 ms depending on input pair.

Finally, we have dependencies when writing and updating the final homography matrix. There is a race condition when multiple threads try to write to the same location in memory, causing the output image to differ from expected. Hence we used a critical section and modified homography matrices serially, which limited our parallelism because threads had to stop at the implicit barrier before writing to the homography matrices.

Stitched example results

References

[1]https://www.analyticsvidhya.com/blog/2019/10/detailed-guide-powerful-sift-technique-image-matching-python/

[2]https://courses.engr.illinois.edu/cs498dh/fa2011/lectures/Lecture%2018%20-%20Photo%20Stitching%20-%20CP%20Fall%202011.pdf

[3]https://stackoverflow.com/questions/12017790/warp-image-to-appear-in-cylindrical-projection

List of Work by Student

The work was split approximately 50% and 50% between Jesse and James. Jesse implemented the initial serial version in C++ and James implemented the parallel version building on top of it. James also

implemented the top-down warp and stitch approach. The writeup was written collaboratively and the video was edited by Jesse.