15-618 Final Project: Parallel Augmented Reality with Planar Homography

Jiayu Bai, Xingsheng Wang

December 2020

1 Summary

For the final project, we parallelized a augmented reality program and achieved 45x speedup using multiprocessing and CUDA. The program uses planar homography to project a video on top of another surface in another video. The machine used has a Intel i7-8700k CPU and a Nvidia GTX 1080 GPU. An example output video could be found on the project website:

`https://checkraiseoncloud.github.io/ParallelPlanarHomography/`

2 Background

2.1 Input

The input to the program includes the original video, an image of the target surface, and the target video on which we want to project the original video. The example video uses a movie clip from kungfu panda as the original video and projects it to the target surface of a textbook in the target video.

Figure 1: original video

Figure 2: target surface

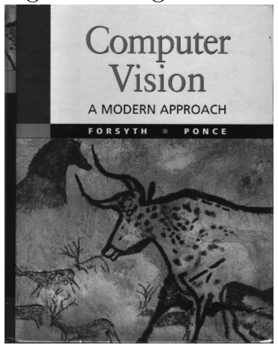
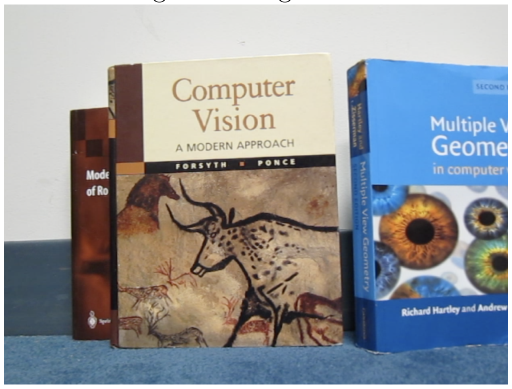


Figure 3: target video



2.2 Planar Homography

In order to project the original video onto the target surface of the target video, we need to perform planar homography on every frame of the video. For each frame, we need to find a homography matrix H that projects the target surface onto the target video frame. This homography matrix is a 3*3 matrix and it is the mathematical representation of warping the surface on to its location in the frame. For every point in the surface target, we can obtain its location the target video frame by the following equation:

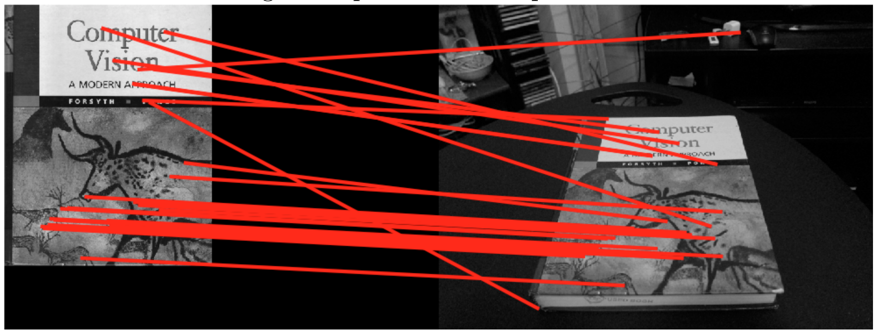
$$\begin{bmatrix} X_{video} \\ Y_{video} \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} * \begin{bmatrix} X_{surface} \\ Y_{surface} \\ 1 \end{bmatrix} \tag{1}$$

Once we have the homography matrix H, we can use it to project the target video frame to the target video frame.

2.3 Points of Correspondence

In order to find the homography matrix, we need to first find points of correspondence between the target surface and the target video. Point correspondence means where a sample physical feature point in the target surface is located in the target video frame. For example, in the following image, feature points on the target surface on the left are matched against their locations in the video frame on the right.

Figure 4: points of correspondence



To find and match these corresponding points, we first use corner detector to extract feature points in the target surface. This would give us approximately 900 to 1000 feature points from the target surface and the video frame. These are stored as numpy arrays. For example, if we have 954 feature points, then we would have two numpy array locs1 and locs2, both of shape (954,2), where the columns are the x and y coordinates of feature points. locs1 represents the feature location on the target surface, whereas locs2 represents the feature location on the video frame.

Then, we use BRIEF (Binary Robust Independent Elementary Features) descriptor to summarize the pixels within a small patch around the feature points. Each feature points are described by a 256 bit vector of 0's or 1's. Each bit represents comparing a randomly selected pair of pixels on the patch around the feature points. For example, for 954 feature points, the descriptor of feature points are represented using numpy array of size (954, 256). Once we have these descriptors, we can match them using Hamming distance. This will give us multiple point correspondences between the surface image and the target video frame. For each feature points in the surface image, we can obtain where that point is located on the target video frame. For example, if we have 74 matched points, we would have two numpy arrays of size (74,2), each representing the matched points location on the target surface and the video frame.

2.4 RANSAC

Once we have multiple point correspondence, we can move on to solve the homography matrix. Since there are miss matches and noise in the point correspondences, we use an algorithm called RANSAC (Random Sample Consensus) to select the best homography matrix. For every iteration, we first select four points to calculate H. Then we evaluate H by warping all the feature points and checking how many points actually get warped to their corresponding locations. The best homography matrix is the one where we have the largest number of in-liers: points whose warped location is close enough to its corresponding location on the target video.

2.5 Homography Matrix

For every iteration of the RANSAC algorithm, we derive the homography matrix with the four corresponding points using the following equation:

$$A * \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = 0 \tag{2}$$

where

$$A = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 & -x_1{}' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 & -y_1{}' \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 & -x_2{}' \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 & -y_2{}' \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 & -x_3{}' \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 & -y_3{}' \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 & -x_4{}' \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 & -y_4{}' \end{bmatrix} \tag{3}$$

and $(x1, y1)$ and $(x1', y1')$ represents the first pair of corresponding points.

We can then derive H by using Singular Value Decomposition of matrix A. Finally, we use the homography matrix to warp the frame of the original video to the target video.

2.6 Overall Algorithm

This is a psuedo code of the overall algorithm:

```
Data: target surface, original video, target video
Result: original video warped on top of the target surface in the target
        video
load target surface, original video, and target video;
for every frame in the target video do
    find feature points target surface and target video frame ;
    match feature points and obtain points correspondence;
    for number of RANSAC iterations do
        select four random matching points;
        calculate the homography matrix;
        projects all matching points using the homography matrix;
        calculate the number of in-liers and update the best homography
          matrix;
    end
    use the best homography matrix to warp original video frame to the
      target video;
end
```

Algorithm 1: Augmented Reality with Planar Homography Algorithm

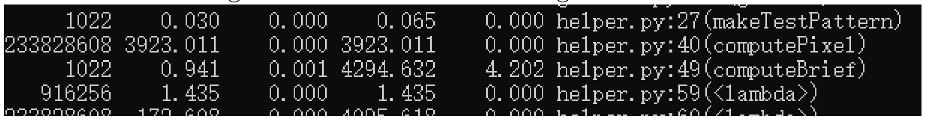
2.7 Parallelism and Workload Analysis

We can see that there are several axis of parallelism that exist in this program. The first one is that the homography matrix needs to be calculated for every frame. Thus, we can parallelize over all the frames in the video. The second one is the RANSAC algorithm on each frame, we can parallelize over the number of iterations to find the best fitting homography matrix. Moreover, for each iteration, we need to calculate the number of in-liers, so we can parallelize over all the point correspondences.

As for cache locality, there is no temporal locality between frames: once we finish one frame, we move on to the next one. However, after we obtain the point correspondence, the working set size is small enough to fit into the cache: we only need the matching points coordinates to calculate the homography matrix. The real challenge is when we are matching the feature points, we are looking at random patches all over the video frame and target surface.

Further detailed workload analysis showed that the most time consuming part of the algorithm is actually the feature point matching part. The sequential algorithm had a run time of about an hour, so we profiled the whole program to see which function call took the most time to complete. In the figure below, we see that the function computePixel took the longest time. This function was previously given as a library function to us and was never changed when we implemented the sequential algorithm. By analyzing this function, we see that it is recoding the BRIEF descriptors into 1 and 0 given a list of pixel coordinates.

Figure 5: Most time consuming function call



```
     1022    0.030    0.000    0.065    0.000 helper.py:27(makeTestPattern)
233828608 3923.011    0.000 3923.011    0.000 helper.py:40(computePixel)
     1022    0.941    0.001 4294.632    4.202 helper.py:49(computeBrief)
   916256    1.435    0.000    1.435    0.000 helper.py:59(<lambda>)
```

Typically, there are 900-1000 centre points to check and each point has 256 pixel points to map. This means that we are doing computation for a matrix of size around 256000 integers for each call of computePixels. This function is called twice per frame and with over 500 frames to process, this function takes the longest time to complete given the share amount of data to compute on. The computation itself is not complicated with 3 memory access and 12 arithmetic operations. This scenario is ideal for GPU computation as it has a lot of small cores to do simple computation.

3 Approach

3.1 Resource

We used the project assignment from Computer Vision (16-720) as our project baseline. This is a sequential python implementation of the planar homography algorithm. We parallelized this implementation on Intel i7-8700k and NVIDIA GTX1080. The way we approached to optimize our project is described in the following subsections in order.

3.2 Parallelize over frames using multiprocessing

We first started with the obvious approach: parallelize over the individual frame of the target video. When we calculate the homography matrix, we only need to look at the current video frame and the target surface, which means there is no dependency between frames.

Due to the fact that python multithreading is limited by the Global Interpreter Lock, multithreading cannot fully utilize the hardware. Thus, we used the multiprocessing to fully utilize the CPU, which had 6 cores and 12 hardware threads. Since each process will have their own memory space, we chose to directly write the result of each chunk into separate video files and combine these video files into a final video in the end. This significantly reduced the amount of communication between processes. Otherwise, we will need to send all frame result to a single process to write to the video file. Since each frame contains a large amount of data, writing into separate video files is more efficient.

There are 511 frames in total and we divided these frames into multiple chunks, each with 15 consecutive frames. We decided to go with block assignment because video frames are written one by one. Plus, if we use interleaved assignment, frame results need to be sent. To assign work to each process and ensure a balanced workload, we have a central work queue server, and the processes use sockets to connect to the server and get the next chunk to process.

3.3 Parallelize RANSAC

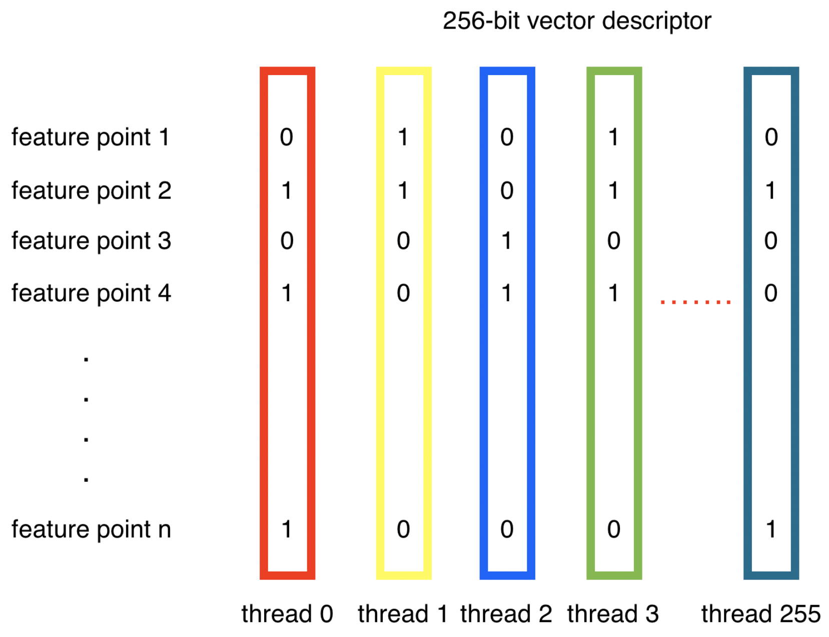
As mentioned above, there are multiple axis of parallelism in the RANSAC algorithm. After we finished parallelizing over video frames, we looked at how to parallelize RANSAC. Since we are already using all the hardware threads to parallelize over frames, we decided to use CUDA to parallelize RANSAC. Our idea is to use each CUDA thread to compute a homography matrix from a random sample of points and then use multiple threads to evaluate this homography matrix against all feature points. Since the RANSAC algorithm uses many numpy operations to compute the homography matrix, we were hoping that these numpy operations could be supported in python CUDA kernel functions. We looked at Numba, which can compile some python and numpy operations on CUDA. However, after spending a ton of time researching on how to get these numpy operations compiled, we found out that only a small subset of operations are supported. To parallelize RANSAC on CUDA, we would have to implement matrix multiplication, singular value decomposition and many other operations on CUDA kernel functions by ourselves, which is out of scope of this project.

Then we tried to parallelize RANSAC iterations using MPI with the mpi4py library. Since RANSAC runs for a fixed number of iterations, we decided to let each processes run a reduced number of iterations. For example, the sequential version RANSAC runs for 600 iterations, each time using four randomly chosen corresponding points to calcualte the homography matrix. We used 12 processes, and each runs for 50 iterations. We need to pass all the matching points to each processes using MPI. After all processes are done, we gathered the locally optimized homography from each processes and choose the best one according to the number of in-liers. Unfortunately, this did not gave us good speedup: only 2x when we have 12 processes. The reason is that there is a lot of communication between processes. Plus, other processes are sitting idle when the master process is calculating the point correspondences. Thus, we decided not to pursue this approach.

3.4 Parallelize point matching using CUDA

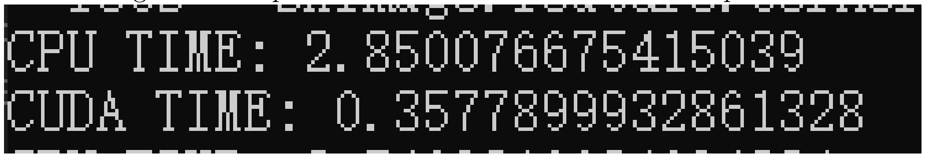
After we used multiprocessing to parallel process frames, we hit a speedup of 6x, which is way below our target. Even with fine tuning with the load balancing, we do not see much improvement with the speedup. Since the working set is small enough to fit into a cache, cache misses won't be a problem and thus suggested that there are work within the processing of each frame that can be parallelized. With the CPU at full load, we turn to parallel computation using GPU. With the workload analysis done above, we identified one place where matrix computation has become a big problem. The computePixel function is called on nearly 512000 matrix elements and a heardware thread is now in charge of doing the work. This is the bottleneck for our program after using multi-processing on frames. Within the function computePixel, the image is accessed multiple times at pixel locations and those pixel locations are accessed in a random style. There will definitely be cache misses since a frame is too large to be fit into the cache. Also, we can't improve locality here since the points of correspondence are unordered. One point in the upper corner of the image may correspond a point in the lower corner of the image and hence we won't be able to improve locality here. Therefore, GPU computation is used here. With the large amount of CUDA cores available, we will be able to process the large amount of data in parallel. The matrix generate has 256 columns and hence we have 256 threads per block. The number of rows is dynamic depending on the number of points correspondence found within the images and hence we used as much blocks as we can to fit the rows into the blocks.

Figure 6: computePixel CPU and GPU runtime comparison



In addition, within the CUDA implementation we also did some optimizations. An important thing to note is that computePixels read the matrix elements and generate a new matrix. Hence all the data supplied to the GPU should be placed in constant memory that is read-only and shared across all threads. We also tried to have some local memory per thread and save the temporary results in local memory before saving them to global memory. However, this change did not result in great performance improvement. The local memory is not capable of storing that many temporary results and if we update the global memory in intervals the improvement won't be better than updating it once. Hence we update to global memory directly. With all the optimizations in place, we achieve a 10x speedup in calling computePixels alone. The speedup is now mostly capped due to the PCIe 3.0 bandwidth designs and limited computing resource of the GPU. (An RTX 2080 GPU with more CUDA cores and higher clock speeds will result in better speedup)

Figure 7: computePixel CPU and GPU runtime comparison



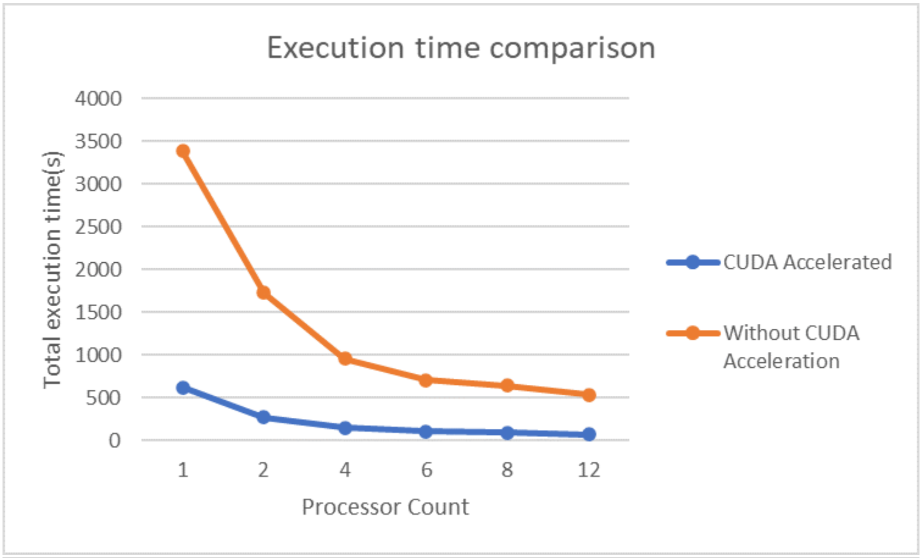
```
CPU TIME: 2.85007667541 5039
CUDA TIME: 0.357789993286 1328
```

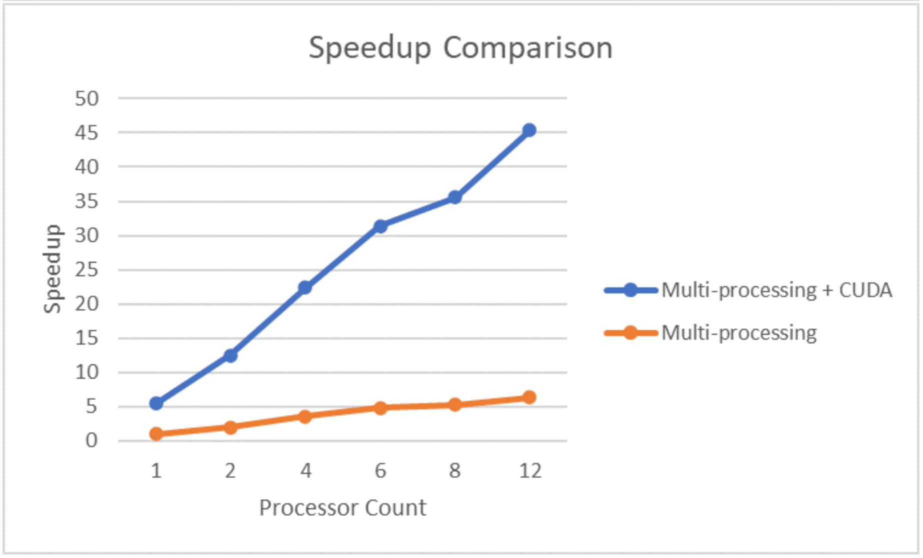
4 Results

4.1 Results and Speedup Graph

The input to the system consists of the original video and the target video. The original video is 20 seconds long, and the video file size is 6.3MB. The target video is 21 seconds long, and the video file size is approximately 28.5MB. The output video is 26 seconds long and the output file size is 16.3MB. The output video is 20 frames per second, with 511 frames. For this project, we are keeping the problem size constant and testing speedup on the same set of videos. Since we assume that the each frame has approximately the same amount of computation, and frames don't depend on one another, we expect the speedup is not dependent on the length of the videos. Since the input data size is already very large, we did not feel the need to test speedup on other inputs.

The total execution time of the sequential code is 3384s. By parallelizing over individual frames on the CPU(i7-8700k, 6 core, 12 threads, 4.3GHz), we achieved 6.3x speedup. After parallelizing feature descriptor calculation on GPU(GTX1080), we achieved final speedup around 45x. We have also measured the execution time using 4,6,8,12 cores both with GPU and without GPU. The runtime and speedup graph is shown below. We can see that the speedup is much better with the help of GPU.


Execution time comparison
Total execution time(s)
4000
3500
3000
2500
2000
1500
1000
500
0
1
2
4
6
8
12
Processor Count
CUDA Accelerated
Without CUDA Acceleration


Speedup Comparison
Speedup
50
45
40
35
30
25
20
15
10
5
0
1
2
4
6
8
12
Processor Count
Multi-processing + CUDA
Multi-processing

4.2 Speedup Limitation

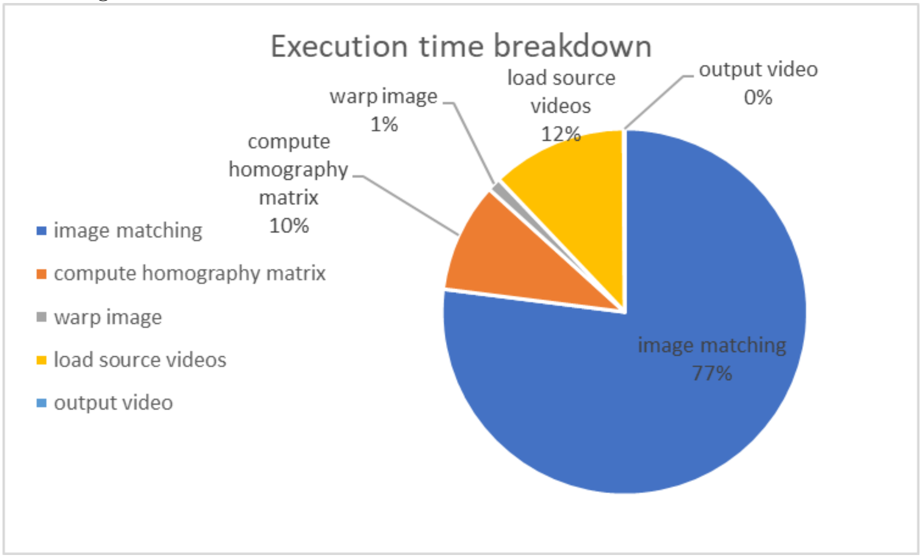
As explained in the background section, the planar homography algorithm has many potential for parallelism. The major limit of speedup is disk accesses and memory accesses. When the programs first started, the target video and the original video needs to be read from disk. The video file size is very large, 34.8 MB in total, and we are reading these video files on disk. To write the warped video frame to the output file, we are also writing 28.5MB to the disk. Since we parallelized over individual frames, frames are written to separate video files in parallel. However, the video loading process cannot be parallelized. In fact, it takes 71.2 seconds to load the videos, which accounts for 11.5 percent of the total execution time. (This time is measured with running with one processor only)

Moreover, we speculate that we also suffer from memory latency during the feature points matching process. There is no temporal locality between frames. After corner detection, the feature points exist in random locations on the video frame. Thus, when we are generating the descriptor for the feature points, we are essentially acccessing random comparator pixels on the video frames. After each descriptor is generated, we never visits these comparator pixels again. After we have matching feature points, we only need the two feature location matrices to generate the homography matrix. For every iteration of the RANSAC algorithm, we are evaluating the homography matrix against all the feature locations. Thus, we do benefit from temporal locality at this stage.

Lastly, there is also hardware limitation. The GPU we have in hand does not have enough CUDA cores to let all 12 hardware threads in the CPU to do image matching simultaneously and hence we do sacrifice some performance here. Also, if given a CPU with more cores, we will be able to process more frames at a time, but that is also subject to GPU CUDA capacity limitations mentioned previously.

4.3 Further Analysis

We measured the time spent on each stages of the algorithm: video loading, feature matching, RANSAC, video writing. This is how much time is spent on each stage:



As we can see, the feature point matching process takes the majority of the execution time. In the sequential version, feature point matching takes 93% of the total execution time. After parallelizing feature descriptor calculation, we reduced its percentage to 77%. There is still room for improvement for feature point detection and descriptor matching. For feature point extraction, we are using corner detection, which runs convolution on the whole image. For descriptor matching, every feature points needs to be compared to each other and the hamming distances needs to be sorted.

There is also room for improvements for parallelizing the RANSAC algorithm. As explained in the previous section, we failed to parallelize RANSAC on GPU due to technical limitations. The RANSAC algorithm has plenty of opportunities for parallelism. We can parallelize over the total number of iterations. We can also parallelize over all the feature points when we are using the homography matrix to calculate the number of in-liers.

We think that the combination of CPU and GPU is a good hardware platform for this project. CPU can handle file input/output, while the GPU can handle the feature matching algorithm and potentially the RANSAC algorithm.

5 References

- Project is based on Assignment 2 from Carnegie Mellon University 16-720 Spring 2020 Course
- Parallel methods from Carnegie Mellon University 15-618 Fall 2020 Course `http://www.cs.cmu.edu/~418/`
- Planar Homography `https://docs.opencv.org/master/d9/dab/tutorial_homography.html`
- Numba documentation `https://numba.pydata.org/numba-doc/latest/index.html`
- MPI for python documentation `https://mpi4py.readthedocs.io/en/stable/`

6 Work and Credit Distribution

The credit distribution is 50-50. We shared our work on most of the aspects of the project. The following is a breakdown of work done by each team member:

- Project proposal (Jiayu Bai and Xingsheng Wang)
- Program profiling and workload analysis (Jiayu Bai and Xingsheng Wang)
- Parallelize over individual frames (Xingsheng Wang).
- Dynamic work allocation for parallelizing over individual frames (Jiayu Bai).
- Checkpoint report (Jiayu Bai and Xingsheng Wang)
- Performance evaluation and further optimization discussion (Xingsheng Wang)
- Parallelize RANSAC with MPI (Jiayu Bai)
- Parallelize image matching with CUDA (Xingsheng Wang)
- Final report (Jiayu Bai and Xingsheng Wang)