15618 Project Midpoint Report

Jiayu Bai, Xingsheng Wang

November 2020

1 Schedule

In terms of the original project schedule, we think we are still on track to complete the project on time. Because the project checkpoint is delayed by a week, our schedule is also extended. By our original schedule, we would have finished reviewing the sequential version of the code and parallelizing over individual frames before the checkpoint. This is all completed before the checkpoint. Due to the extension of the checkpoint, here is a revised schedule for the following weeks up to the final poster session.

1. 11.30 - 12.3
    - Environment and dependency setup to run project with Cuda. (Xingsheng Wang)
    - Parallelize RANSAC: parallelize over each iteration to calculate homography matrix. (Jiayu Bai)
2. 12.4 - 12.6
    - Parallelize RANSAC: parallelize over point correspondences to evaluate homography matrix. (Xingsheng Wang)
    - Further optimization and performance evaluation. (Jiayu Bai & Xingsheng Wang)
3. 12.7 - 12.11
    - Finalize project. (Jiayu Bai & Xingsheng Wang)
    - Finish project report. (Jiayu Bai & Xingsheng Wang)
    - Finish presentation for poster session. (Jiayu Bai & Xingsheng Wang)

2 Work Completed

First, we reviewed the sequential version of the code implemented in python with numpy, skimage. Opencv is also used for reading and writing video frames. We combined our implementation and solved a few bugs that we did not have time

to solve previously.

Then we decided to first parallelize using CPU. Due to the fact that python thread is limited by the Global Interpreter Lock, we used the multiprocessing library to fully utilize all the cores in the CPU. Python has both multi-threading and multi-processing library. Multi-threading does not use all the hardware cores and all the threads share the same space. Multi-processing will be able to use all the hardware cores but each process will have their own memory space. Since we need to utilize all the cores,we decided to go with multi-processing. This brings a problem where we have separate memory space for thread. We can either choose to process each frame and send the frame data to one process to compile it into a final video or we can directly write the frames into video segments and then combine them later. Since the frames contain a large amount of data, we tested both methods and found out that writing into segments and then combining them is more efficient. The overhead with send frame data is much larger.

The next problem is how we assign work to each process. In lecture we see there are block assignment and interleaved assignment. Interleaved assignment will have better cache performance but that is on a shared memory space implementation. Video frame have a specific order and we have to write the frames in order. If we used interleaved assignment, then we either create a video file for each frame and then combine them together or we send it over to one process and that process will output the video frames. Either way is not ideal as it introduces either a large communication overhead or a lot files that need to be combined and reading the files will give a large overhead. Hence, we decided to use block assignment.

After that, we will need to pay attention to workload balancing. For each frame, the amount of data that needs to be processed is roughly the same. The way homographies are calculated is that we randomly choose points and calculate the homography matrix. Since the process is random, the accuracy will be random and the way to solve that is to do a loop and each loop will randomly select point to calculate the matrix and report an accuracy. The matrix with the best accuracy is chosen and this is done for all frames and hence this part of the work is same. However, before finding the matrix, we will find points of correspondence between the two sources and hence this will lead to a bit of variance in workload. Generally speaking, work distribution is balanced. To account for the fact that some threads will complete slightly faster, we have a central work queue server, and the processes use sockets to connect to the server and get the next segment to process. This way we can keep the processes as busy as possible.

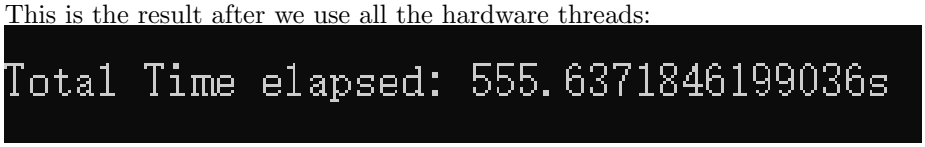
3 Results

For now, we parallelized over the videos frames and achieved 6.1x speedup. This is achieved using multi processing on CPU (i7-8700k, 6 core, 12 threads, 4.3GHz).

The figure below is the sequential time: (The value is negative because we calculated startTime - endTime).

We see that it took almost an hour.

This is the result after we use all the hardware threads:



The speedup is quite significant the 20s video took an hour to process previously and now it can be done in 10 minutes.

4 Goals and Deliverables

In our proposal, we aimed to get 20x speedup. Right now, we are unsure if that goal is achievable. By parallelizing over 12 hardware thread, we are able to achieve 6.1x speedup, which is less than what we expected. On the other hand, we have not yet utilized the power of GPU. We plan to parallelize the RANSAC algorithm on GPU cores, which would increase our speedup. The speedup that CUDA will provide is not yet known since we are using python and the overhead of using python CUDA library is not tested yet. Hopefully, that would help us achieve our goal and it is also equally likely that in the end we may not achieve our goal due to the shear amount of work-related computation needed and current hardware limitations.

5 Poster Session

For the poster session, we want to show augmented reality video as well as how we parallelized RANSAC. We will also show our speedup results.

6 Concerns

Our biggest concern right now is running numpy code on GPU. The RANSAC algorithm has a ton of numpy operations to calculate the homography matrix from the corresponding points vector. We have no experience running numpy

on GPU before, but hopefully with the help of Numba, we will figure out how to do that. The speedup may not be ideal due to us not being able to use the best hardware available on market and that may limit how many matrix multiplication we can do at the same time.