COP290: Assignment1

Saurabh Verma 2019CS50129
Sachin 2019CS10722

Subtask 3

1 Introduction

Following is our interpretation of required terms given in the problem statement used to do trade-off analysis of the code:

- **Benchmark:** For all the methods our benchmark is the video "trafficvideo.mp4" provided.
- **Baseline:** We consider the method 0 (see subsection "Method 0" for details) as our baseline since we delayed the analysis of frame in sub-task 2 and made speed of analysis user dependent hence we cannot use that.
- **Utility metric:** We are taking the absolute difference in value of queue density at any frame with respect to what we got in subtask 2 (stored in file "asli.csv" in folder "analysis") as the framewise error and its square mean average as the total error of this utility.

1.1 Method 0

This method does the normal analysis of benchmark without changing any trade-off i.e. it reads every frame of of the video in the same size provided sequentially and perform background subtraction of the bird eye view of the frame as a whole.
Time taken to run this method is **83s** which we will use as reference for the comparison of the performance of different utilities.

1.2 Method 1

- **Parameter:** Frames skipped

In this method we had to skip some given number of frames for the analysis and use the last known values of queue density for them. Since this is more or less same as approximating any curve with straight lines (what we done in approximating area with the integral), we got the same nature of results in error.

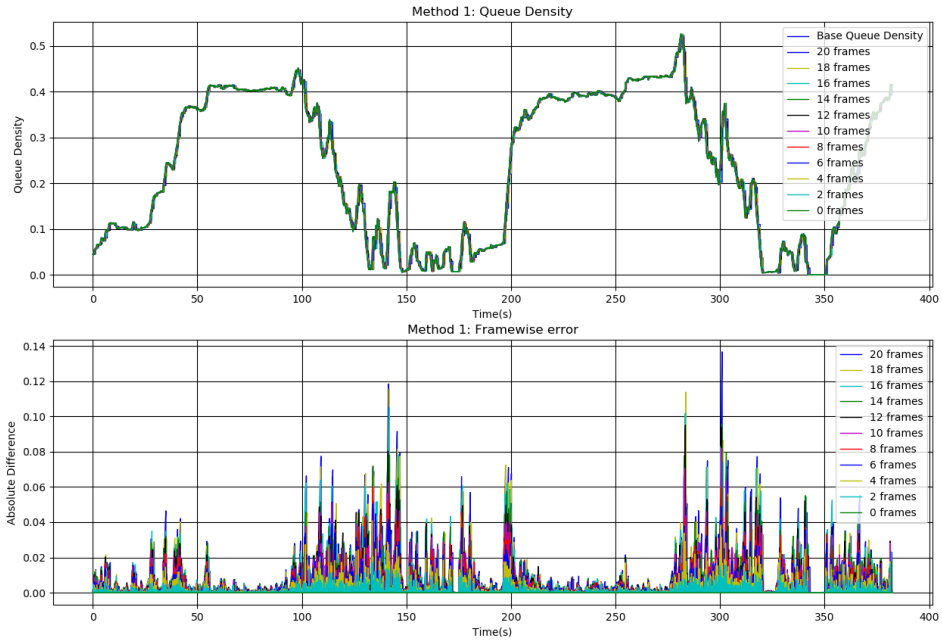

Figure 1: Method1- Density & Error

NOTE: Due to limited availability of colours in matplotlib, there is a repetition of colours, but one can differentiate by noting that we have plotted the graph from higher to lower, i.e. curve with more number of frames skipped is behind the one with less.

Following inferences can be made from the above graph of framewise error and queue density values by skipping arbitrary number of frames:

1. As the number of frames skipped increases absolute error at each frame increases.

2. For given number of skipped frame during the interval of frames where queue density fluctuate more have higher error compare to interval where queue density has less fluctuations.

3. But the overall nature of the graph remains same, one cannot tell the difference by examining all shape of the complete graph, we have to zoom the graph and see the inner picture.

   Following graph clearly justifies point 1 and shows how increasing number of skipped frames make graph away from the ideal one.

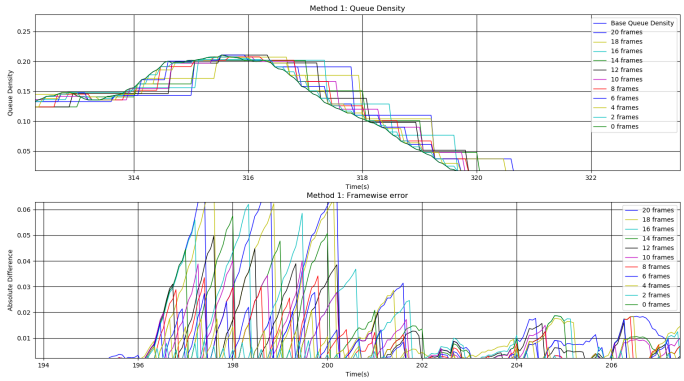

Figure 2: Method1- Zoomed

Now that we have done the framewise analysis, lets see the result of the loss that we suffered in increment in error, since skipping frames decreases the work for the processor hence decreasing the time, clearly seen from the following graph, also the graph shows how averaged error increases as we increase the skipped frames, that is what we inferred from the previous graph.
Also comparing with baseline, the time that we got by without skipping any frames is nearest to the baseline, although it is a bit high, (due to checking of if condition for each frame) but we have got a decent decrement in time after that.

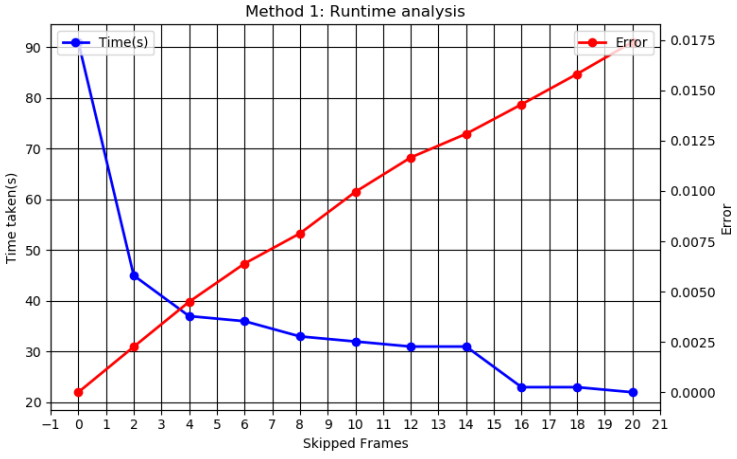

Figure 3: Method1: Runtime

1.3 Method 2

In this method we had to decreases the size of the frame and the analyse that, size we can decreases the size in many different ways, but we have kept the ratio of height vs width same and just down factored both, like decreasing the height and width both by half and so on, calling it collectively as decreasing size by 2.

- **Parameter:** Down factor of size

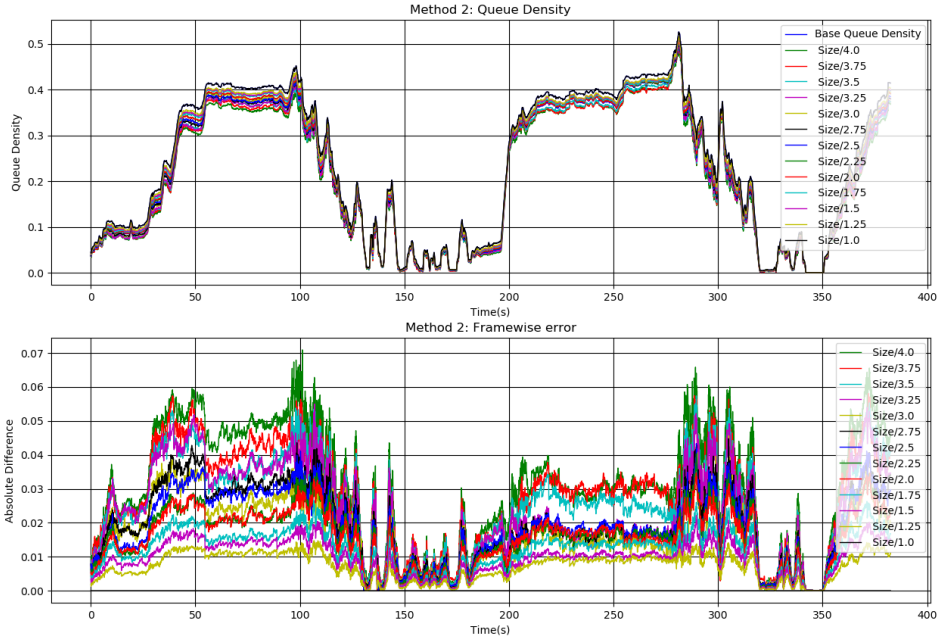

Figure 4: Method2: Density & Error

Following inferences can be made from the above graph of framewise error and queue density values by skipping arbitrary number of frames:

1. As the size decrement increases absolute error at each frame increases.
2. For given downsize factor during the interval of frames where queue density have less value, absolute error is also less, and where queue density is high, absolute error is also high and increases by a larger factor by increasing the downsize factor, this can be explained by the fact that low queue density means less vehicles and hence less white pixels to be checked, so lower size can be taken to get the values.

3. Nature of Queue density as downsize increases can be seen from the above graph only, it decreases that is as expected as decreasing size results loss of information causing less detection of vehicles (non black pixels in differenced frame).

Now that we have done the framewise analysis, lets see the big picture, how the time varies as we increase the downsize factor.

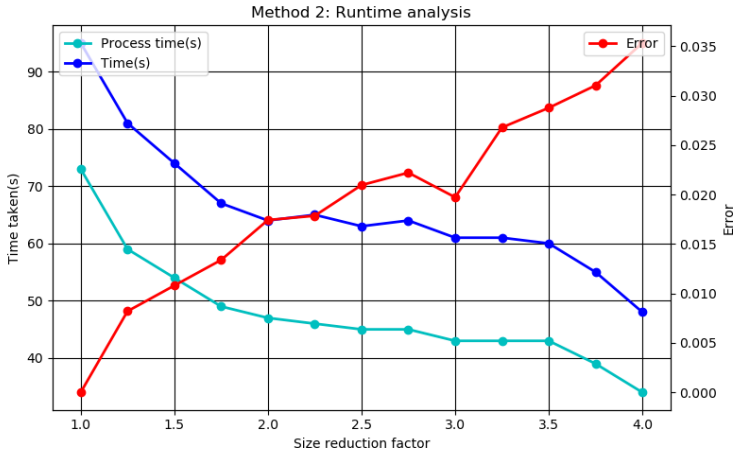

Figure 5: Method2: Runtime

Following inferences can be made from the above graph:

1. As we increase the downsize factor time taken by complete process have a decreasing nature, (it increases a bit for some point 2.5 to 2.75 here, just by 1s) but if we take the time to do the process only, (total - cropping) it has strict non increasing nature. This is because for some parameter, gain by downsizing can be overcomed by cropping time (not by a lot though).

2. But if we compare to baseline we have decreased time by a lot (around 40s with 0.04 error factor), for downsize factor 1 time is more than baseline since we are cropping the image to rectangle of its own size for each frame, which takes some time.

3. Error increases if we increases downsize factor, that is same what we inferred and explained from framewise analysis, but for 2.75 to 3 error decreases a bit.

1.4 Method 3

In this method we had to analyse each frame of video in parts by providing them to different threads and process them in parallel. Since we are hard coding

the points for homography change, we cannot perform homography in parts (actually we can by cropping and shifting the origin for the points accordingly but is not an ideal approach since it causes an significant amount of info loss even for 2 threads only, as our desired view might not be distributed in same proportion in all the parts of frame).
So we cropped the image after homography and them provided all parts to different threads for further evaluation.

- **Parameter:** Number of threads

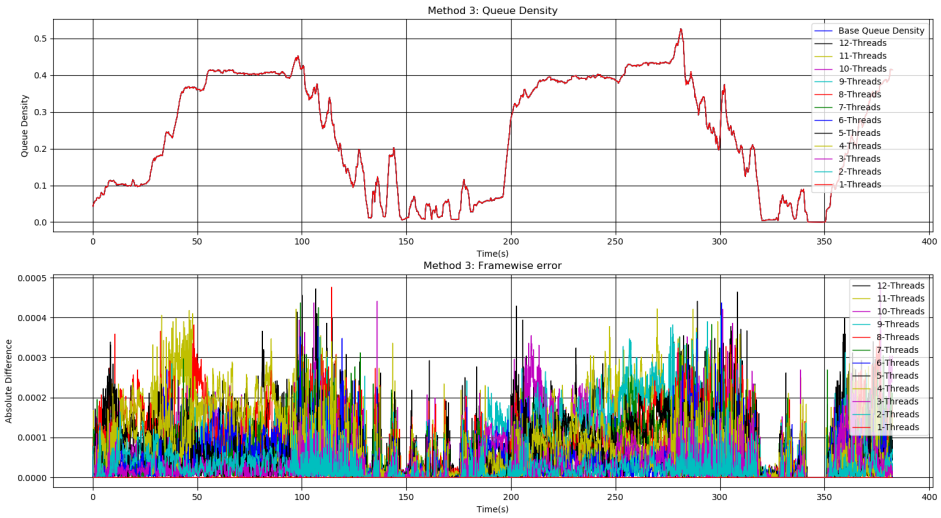

Figure 6: Method3: Density & Error

Following inferences can be made from the above graph:

1. As number of threads increases error increases (roughly). This is because, cropping opens the door for error from boundaries pixels Boundary pixels might not get added in evaluation of difference or can get added twice.

2. For given number of threads during the interval of frames where queue density have less value, absolute error is also less, and where queue density is high, absolute error is also high and increases, this can be explained by the fact that low queue density means less vehicles and hence less white pixels to be checked, so higher chance that pixels missed are black.

Now that we have done the framewise analysis, lets see the big picture, how the time and error varies as we increase the number of threads.
Following inferences can be made from the above graph:

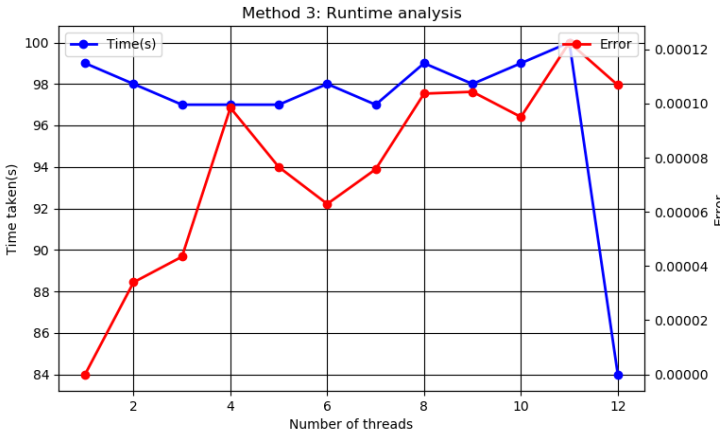

Figure 7: Method3: Runtime

1. Roughly speaking error increases as we increase number of threads (due to increase in boundaries) but it shows strange behaviour sometimes, like it decreases from 4 to 6 and from 9 to 10. This is because error also depends on how crowded the boundary points are, that depends on way of cropping the frame and video itself. Also error range is very low as compared to upper two methods.

2. Time taken to run the whole code decreases initailly then it starts increasing but strangely at 12 threads it just shoots down by a significant amount. First behaviour can be explained by the fact that computing things paralley decreases time but it too has a saturation point because it is no bug task to be done by so many threads, after a point gain is only in milliseconds by threads but time taken to crop the image increases as we increase number of partitions, that increases overall time. Behavious at 12 threads is a bit strange, only explanation we can give is that we performed analysis on 12 threads first (on fresh processor) then in decreasing sequence to others. We know that it is just a wild guess but we that is what we can provide about this behaviour.

3. Comparing with baseline, there is not much of a gain.

We did this all to use our resources more effectively, so lets see if we were able to increase the CPU usage or not.
Following graph clearly shows that as we increase number of threads CPU usage also increases (though it also decreases a bit at 2 to 3) till 10 threads then it decreases stating that afetr a point we are just wasting the threads not that much are needed.
Clearly memory usage also increases as we increase the number of threads since

at any iteration we are storing n number of frames in different threads and there parallely cropping it to desired dimensions and prfoeming background subtraction.

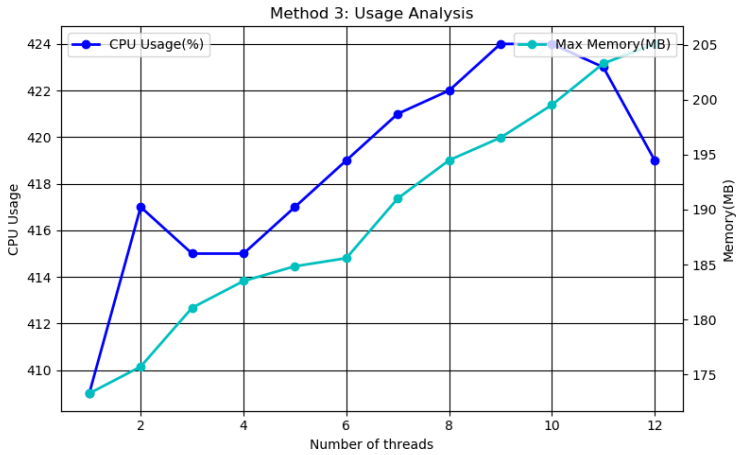

Figure 8: Method3: Usage

NOTE: Since for single frame background subtraction takes very less amount of time and we are creating pthreads for that less amount of time, we also tried to implement this method in a different way, in which we initially cropped each frame of video then stored them in an array and creayed the pthread once only and performed analysis on those stored cropped frames. That obviously shoots up the memory usage but CPU usage also did not gave accepting results.

Note that memory is as high in GBs.

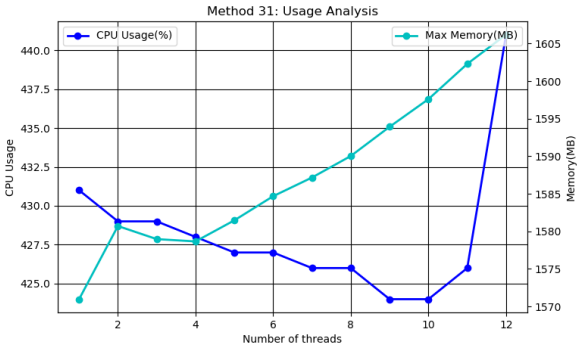

Figure 9: Method3(a): Usage

But loss in memory results in gain of time, we were able to gain 18s from baseline. Error had the same behaviour as previous one.

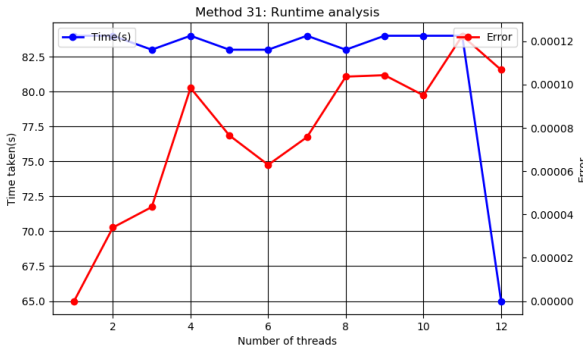

Figure 10: Method3(a): Runtime

1.5 Method 4

In this method we had to in parallel analyse some frames using pthreads. We did this by reading and storing n number of threads and then creating n threads and passing stored threads to them and then perform analysis in parallel.

- **Parameter:** Number of threads

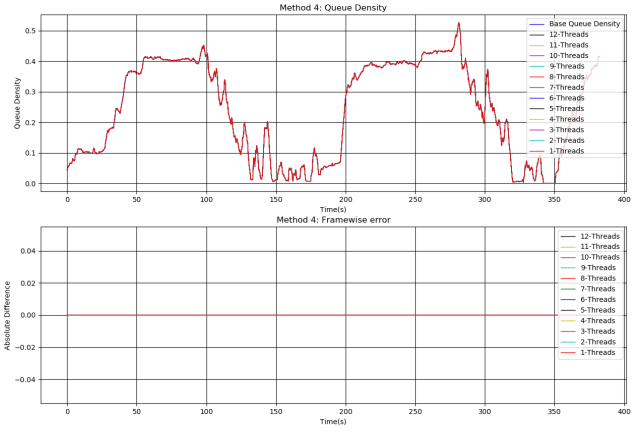

Figure 11: Method4: Density & Error

Inferences that one can made from the above graph is that Density value matches with baseline for all the frames hence error is 0. That is as expected since we are not changing anything in analysis process, we are just doing it in parallel.

Now that we have done the framewise analysis, lets see the big picture, how the time and error varies as we increase the number of threads.

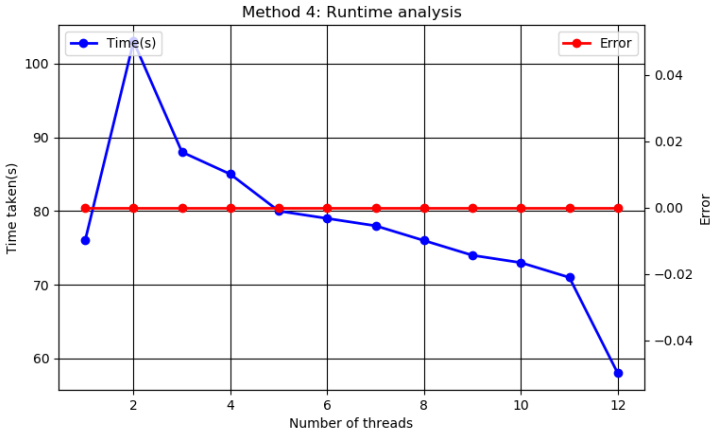

Figure 12: Method4: Runtime

Following inferences can be made from the above graph:

1. Error is zero for all values of parameter.
2. Time increases initially from 1 to 2 and then decreases as we increase number of threads. Second behaviour is as expected since we are performing same task in parallel, hence time decreases, initial increment in time can be because of preparatory tasks (like storing frames for giving them to threads, storing the results and getting them back for printing etc.)
3. W have gained significant amount of time from baseline (around 25s) that too without any error.

We did this all to use our resources more effectively, so lets see if we were able to increase the CPU usage or not.

Following graph shows that we indeed used our CPU more effectively as we increase number of threads (except from 1 to 2, where it decreases that is somewhat logical since time too increased in this part).
Also due to storing of consecutive frames memory usage also increases uniformly as number of threads increases same as part 3, but with some offset from that because here we are also storing the output of queue density of each frame and printing it at the end.

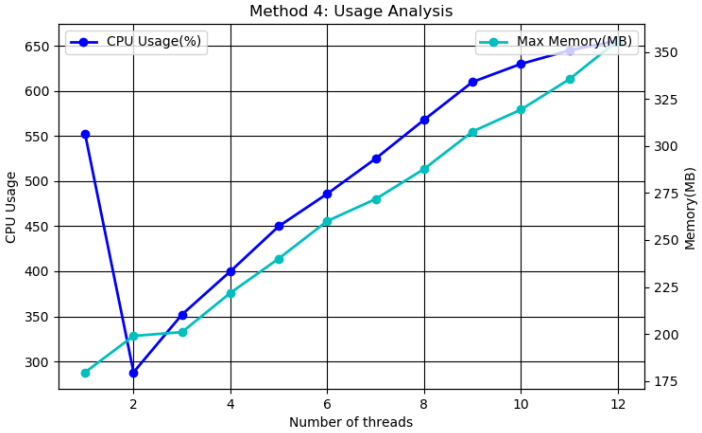

Figure 13: Method4: Usage

For this method also we made an other variant where we create pthread single time only, by copying the video number of threads times and giving that to each thread and each thread does analysis of different frame from each video in parallel.
Here is the runtime result for that:

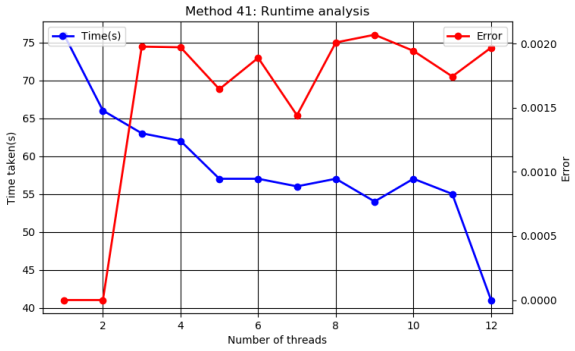

Figure 14: Method 4(a): Runtime

Following inferences can be made from the above graph:

1. Time decreases as number of threads increases. (same reason as earlier).
2. We are able to get a lot of gain in time (more than method 4) by an amount of 42s.

3. Error shows strange behaviour, this is because during the setting of starting frame of videos of different threads, synchronization of frames gets affected (for some thread it is ahead of ideal frame by 1, for some it is 2 and for some it is in sync). We did our best to remove this ambiguity but not able to do so. But error is not as huge as method 1 or 2 it is very minute.

Above graph shows everything good about this method, but we are getting this much gain in time by the trade-off of memory, by storing video multiple times we are taking a lot of memory, Following graph shows it very well.

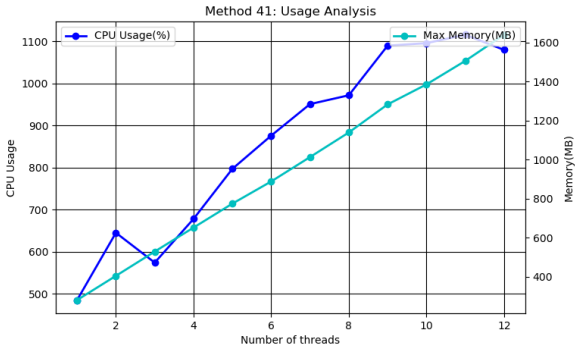

Figure 15: Method 4(a): Usage

This graph also shows that using this method we were able to get the best usage of our CPU, for 12 threads it is as close as 1200%.

1.6 Optical Flow

We have in implemented two approaches to calculating the dynamic density. We have analysed these two methods.

- **Dense Optical Flow:** We calculated the dynamic density using the calcOpticalFlowFarneback function of the opencv library which basically track all the points in the two frames. In our implementation we have skipped 3 frames and have calculated the optical flow of points in frame at an interval of 3. This gives us a dynamic density which contains some amount of noise. This noise is due to the fact that we are tracking all sets of points rather than some specific features . In order to reduce the noise we have averaged the values in two consecutive frames to get us a more accurate result. The time taken by this function is 241s and the CPU usage is 250 percent.The memory usage was 159292 kbs.

- **Sparse Optical Flow:** We calculated the dynamic density using the calcOpticalFlowPyrLK function of the opencv library which tracks some

specific features in the frame. In our implementation we have used the points which are counted in queue as the features to track. Here too we have skipped 3 frames in between for better time. There isn't any issues of noise in this approach as we are tracking very specific points and thus chances of noise is very low. The time taken by this function is around 128s and CPU usage is 486 percent.The memory used was 157188 kbs.

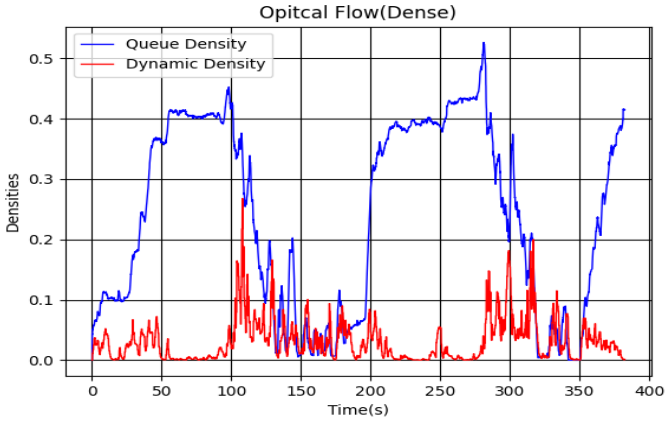

Figure 16: Opitcal Flow(Dense)

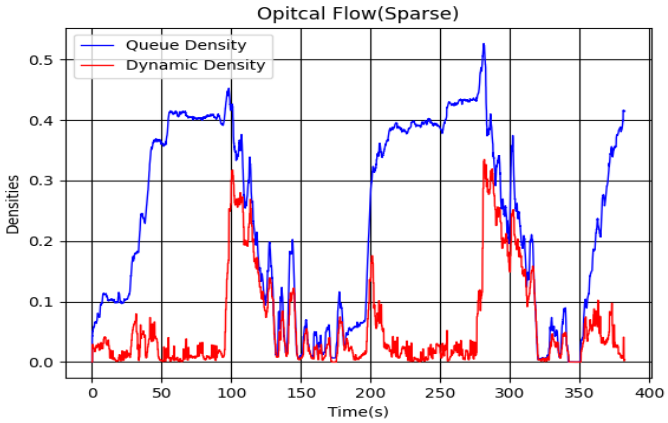

Figure 17: Opitcal Flow(Sparse)

1.7 Conclusion

1. During the whole process most of the time is taken in chnaging homography of the frame (more than 50%), so the methods that incorporate with homography, performs well in time i.e. in method 1 we are doing less homography change, in method 2 we are doing homography on shorter image, in method 4 we we are doing homography in parallel, hence we get significant gain there but in method 3 we are not changing anything for homography hence we are not getting much gain there.

2. As we go for using our CPU more and more effectively (method 31 and 41) we are paying off in our memory) so we need to find a middle way of both.

3. Best method of all is method 4 where we are gaining time by minimal memory loss and without any error.