Report on Traffic Density Estimation

Shrey J. Patel
2019CS10400

Rahul Chhabra
2019CS11016

March 2021

1 Introduction

- In this assignment, we implemented a program to calculate the density of traffic from an input video using OpenCV in C++. Our main approach was to process the video frame-by-frame and classify the elements on the frame into stationary(static) and moving(dynamic) objects. The stationary objects excluding the background, which is the road in our case, constitutes for the queued traffic and therefore amounts to the queue density. While the moving objects constitute the dynamic density.
- We have used **background subtraction** to calculate both queue and dynamic densities. We have used the initial frame as our background frame for the whole video. According to the discussion above, queue density of the current frame can be obtained by subtracting the background frame from the current frame, while the dynamic density can be obtained from the difference of consecutive frames.
- Note that we have considered all frames to be in grayscale. Additionally, we have set a threshold on the pixel values of the difference images so that they contain only white or black pixels, thus making it easier for us to calculate densities. Consequently, the densities can be calculated as the ratio of area covered by white pixels to the total area of the frame. For example, for queue density, any pixel having the value above 50 will be considered as a white pixel while for dynamic density, any pixel with value above 20 will be considered as a white pixel.
- In the calculation of dynamic density, the difference of consecutive frames only displays the outline of the moving object, and not the whole object, thus the value we obtain is very less as compared to the actual value. So we used an inbuilt function called Gaussian blur to approximately trace the whole object using the outline i.e. blurring the object.
- And lastly, we noticed that calculating densities using background subtraction instead of optical flow causes fluctuations(noise) in density values of consecutive frames. We reduced this noise by implementing a noise

suppression feature in which we extrapolate the density values of the last frame if they differ by a large amount.

After implementing all the features mentioned above, we obtained the density values for all frames and plotted them against frame number. The graph we got was:

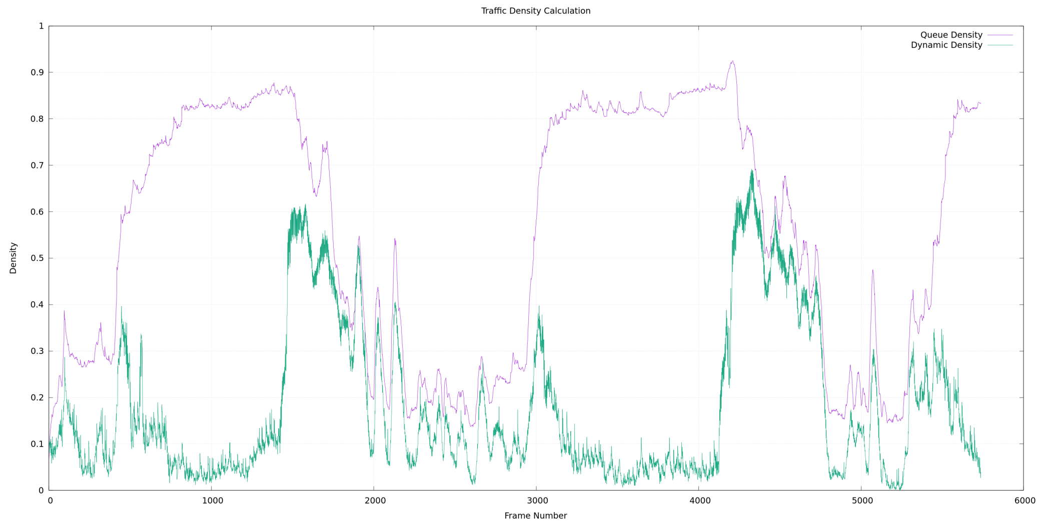

Figure 1: Original

The functions and procedures we used in this part take a lot of time because they process the frame pixel by pixel, and because a normal 15 FPS video has a lot of frames, the total execution time of the whole video can be very high, maybe even higher than the run time of the video. Till now, our utility has been to obtain the density values with as high accuracy as possible, but this requires a lot of execution time which may not be favourable in many practical scenarios. So, we can modify our original code to execute the video much faster while trading off some accuracy.

2 Metrics

Before we explain the implementation of various alternative method, we define the metrics/parameters we have used to grade/rate these methods:

2.1 Run-time Metric

We have considered the execution time of the whole input video as our runtime metric. For determining the execution time, we have used an in-built C++ library called *chrono*.

2.2 Utility Metric

Just as before, we consider the correctness or accuracy as our utility. So, the utility function we have defined compares the different methods with different parameters based on their accuracy.

The value of the utility function of a given method with a given parameter is the inverse of mean absolute error.

Utility value = $\sum_{n=0}^{size\_of\_data-1} |data[i] - origData[i]|/size\_of\_data$

where origData is the array of the original data we obtained without using any runtime-improvement methods and data is the array of current data.
More accurate the method is, lesser will be its error with respect to the original, and so more will be the value of the utility function.

3 Methods

3.1 Sub-sampling of frames

- Since consecutive frames may not differ much, we can take frames at a particular interval by skipping some frames in between, thus reducing the execution time. We implemented this simply by ignoring frames except those which occur at regular intervals of parameter.
- However, in doing so we are missing the changes which may happen in those skipped frames, which results in loss of information, thus trading off accuracy for execution speed.
- But, we can still manage to capture the required trend or pattern of the density values by taking optimal value of the number of frames to be skipped. By optimal, we mean that neither x should be very large that it might lead to a significant loss of information nor x should be too low so that there is a considerable increase in run time.

Parameter: x = Length of interval(in frames) after which the frames are to be processed or (Number of frames to be skipped + 1)

3.2 Changing Resolution

- Another issue we mentioned before that was causing the execution time to increase was that every frame was processed pixel by pixel, since images are stored as 2D matrices in C++. The resolution of the image determines the number of pixels and therefore the size of the 2D matrix. So, we can reduce the execution time by decreasing the number of pixels of the image by changing the resolution.

- But this method also suffers from the same drawbacks as the last method, in that decreasing the number of pixels to improve run time would reuslt in loss of pixels which are potentially important for some details in the image. Thus, decreasing the resolution may result in loss of information and thus this method also trades off utility for better run time.

- And similar to sub-sampling, it should be possible to come up with an optimal resolution which gives a better runtime while still maintaining a reasonable amount of accuracy.

- But we found this method quite slower than sub-sampling, and we think the reason for this lies in our implementation. To change the resolution, we are resizing every frame before processing it, which itself is a fairly time consuming operation, and thus the time saved by processing lesser number of pixels is somewhat cancelled by the extra time required in the resize operation.

Parameters: The required resolution value in the form X,Y where X = width of image and Y = height of image

3.3 Threading

Till now, our main idea behind achieving better runtime was that the time taken is proportional to the amount of work done, and thus better runtime could be achieved only by skipping some work (probably unnecessary or redundant). This is because we considered our workflow to be linear and sequential, i.e. one task can be done only after another task is completed.

But, in parallel computing, we can improve the execution time without skipping any task, because in this, instead of waiting a task to be completed, two independent tasks can be handled simultaneously. The CPU consists of various threads/cores which can work simultaneously for the same task, either by dividing the task into smaller sub-tasks which are then handled separately or handling different parts of the same task.
Parameter: x = Number of threads

3.3.1 Spatial Threading

- We can split the task of processing a particular frame spatially across threads so that each thread works separately on a specific part of the current frame and the final values can be obtained from some suitable combination of individual values obtained from these parts.

- We have implemented this threading by dividing every frame into horizontal strips(sub-frames) whose amount is equal to the number of threads, and then processing each sub-frame separately using pthreads. The net value of the queue density can be obtained as the ratio of summation of the areas of white pixels in each of these sub-frames to the total area of the frame.

- The main advantage of spatial threading is that, because different parts of the same frame are getting processed at the same time, a single frame is processed faster(by a factor equal to the number of threads) and thus the net execution time improves.
- But it may happen that the total number of pixel are not exactly divisible by the number of threads used, in which case we need to take care of boundary pixels so that no part of the image is left unprocessed.
- Another drawback of this threading arises because of the usage of Gaussian blur. This inbuilt function doesn't work on pixels which are present at the periphery of the frame. This will give inaccurate results in the case of spatial threading, because we are dividing our frame into smaller sub-frames, all of which are processed separately. So, when we use Gaussian blur on each of these sub-frames, their boundary pixels, (which are internal pixels for the original frame), will not be processed, causing a loss of information which is proportional to the number of threads used.
- This issue may be resolved either by removing Gaussian blur or by setting an optimal kernel size of matrix used in blurring so that the loss in accuracy is minimum, while still improving runtime.

3.3.2 Temporal Threading

- This method also uses pthreads to divide work among different threads but here instead of dividing a frame into different splits, we distribute consecutive frames among different threads and let them get processed independently.
- This threading is similar to sub-sampling in terms of runtime but in threading, instead of ignoring the intermediate frames, we allocate these frames to different threads, so that all of these frames can be executed in the time which was originally required to process a single frame. So, there is no loss of information in temporal threading.
- This method saves a lot of runtime, but because multiple threads are running at the same time, more resources(in terms of memory/space) are used up in executing the same task. And so the runtime may also depend upon the temperature of the machine.
- Another catch in this method is that the frames may get scrambled i.e. jumbled because each thread requires different times to process a single frame, which can't be determined prior to allocating the frames. As a result, the frames get scrambled(jumbled) after execution as the threads which require lesser time to finish processing a frame will return their result before the ones which take more time, even though the frame which takes more time appears first in the video. Thus the sequential workflow is not maintained.

- This issue can be resolved by indexing each of the frames on the basis of the order in which they appear in the video, and at the end of the video, the scrambled output can be sorted using this index.

4 Analysis

Here we provide the tables and graphs for the test cases as well as the utility-runtime trade-off analysis for each of the methods described above.

Note that for the purpose of analysis, we have removed the dynamic density calculation from the original file i.e. video.cpp

*Note that in each method, the first test case matches with the parameter of the original image, i.e. x=1 for sub-sampling, 1920*1080 in resolution, and x=1 in threading to check whether these methods work correctly for base cases. And as we expect, their output density values very closely resemble the original values(except in temporal threading) and so due to very less error, their utility values are very large compared to other testcases, so we have excluded them in graphs of methods 1 and 2 because of range inconsistency*

Runtime of original code = 266 seconds

4.1 Method-1: Sub-sampling

4.1.1 Testcases

x	Runtime(in sec)	Utility
1	264	60188.12
5	160	229
10	147	99.95
15	139	8.27
20	134	3.25

- As evident from the table above increasing the length of interval of frames to be skipped decreases the runtime by a significant amount.

- But this causes a loss of information which increases with the increase in parameter x, thus decreasing the accuracy and so the utility value.

- Consider the jump in utility values as well as runtimes from x=1 to x=5, where x=1 yields a very high utility value as it is expected to be almost equal to the original value, while in x=5, the utility value drastically decreases indicating a loss in accuracy for a much better runtime.

  Note that the inf value of utility in case of x=1 suggests zero error in output, which is to be expected since x=1 means no frame is skipped.

4.1.2 Graphs

- The graphs for x=1,5 and 10 are fairly similar to the original graph with the accuracy decreasing from x=1 to x=5.

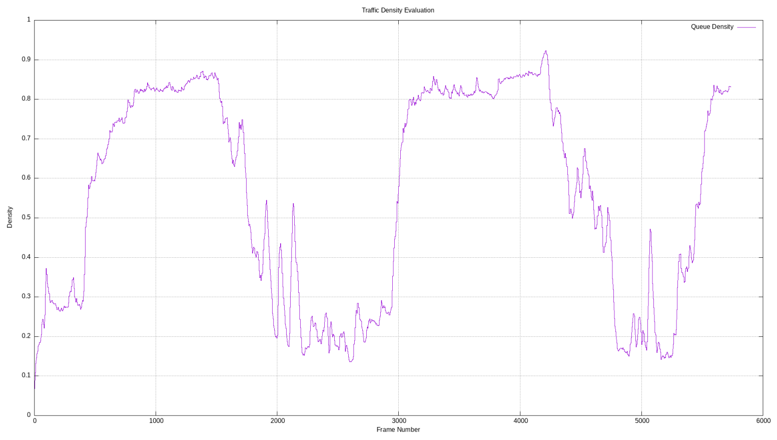

Figure 2: Density vs Frame graph for x=5

- But in case of x=15, a lot of information is missed because we are skipping 14 frames in every interval. Due to which the graph for x=15 is highly inaccurate and combined with noise suppression, the graph contains regions of constant density for long periods. For instance, the frames 400 through 1800 have the same density values which is not correct. So, increasing runtime comes at a cost of decrease in accuracy.

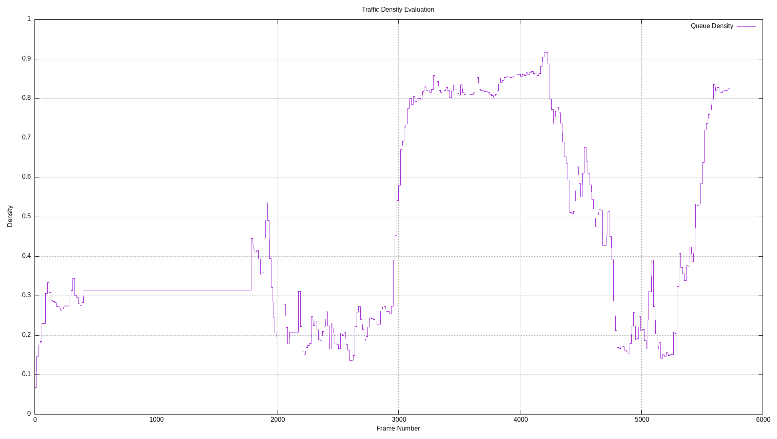

Figure 3: Density vs Frame graph for x=15

4.1.3 Trade-off Analysis

As discussed above, as runtime decreases, the error in output values increase and so does the value of utility function. So the utility runtime graph is an increasing graph.

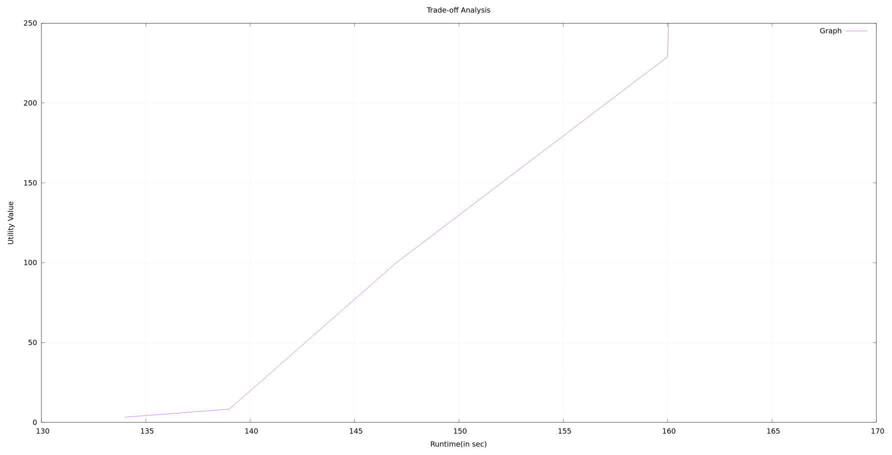

Figure 4: Utility-Runtime plot for Sub-sampling

4.2 Method-2: Changing Resolution

4.2.1 Testcases

Resolution	Runtime(in sec)	Utility
1920*1080	274	60193.80
1366*768	238	27.44
1280*720	241	23.24
800*600	244	20.59
300*400	243	12.74

- The resolution is the measure of the number of pixels to be processed per frame. So, decreasing the resolution should naturally result in decrease in runtime, and this trend is roughly visible in the table as the original resolution(1920*1080) has much higher runtime than other resolutions.
- But in this method, besides processing every frame pixel by pixel, we also need to resize each frame to a frame of desired resolution, which increases execution time. So, farther the resolution from the original value, more is the time required for the resize operation of each frame. So the run time doesn't decrease constantly on decreasing the resolution.

4.2.2 Graphs

All the cases have almost the same graph of queue density values as the original method, and all of them differ only in their absolute values.

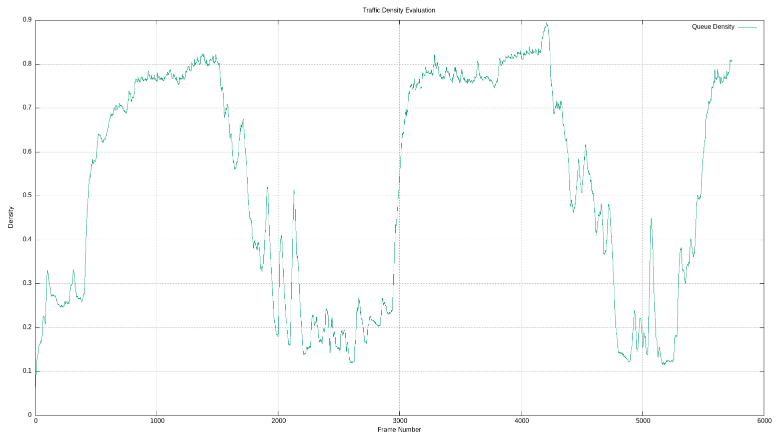

Figure 5: Density vs Frame graph for 1366*768

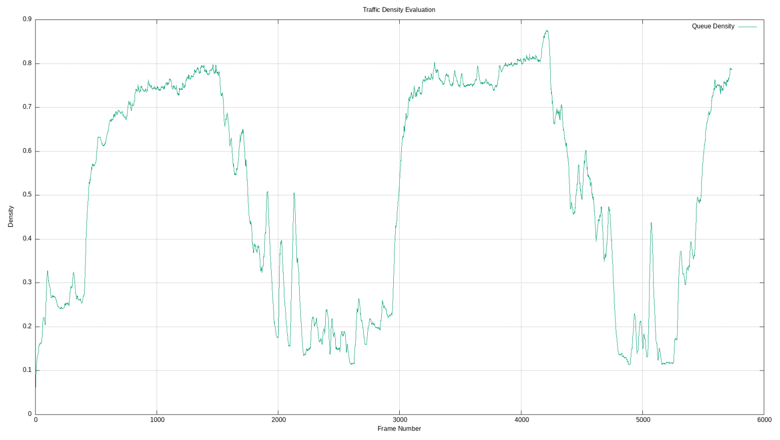

Figure 6: Density vs Frame graph for 800*600

4.2.3 Trade-off Analysis

- As mentioned above, the farther the resolution is from the original resolution, higher will be the time required in the resizing of frames, but the utility value will constantly decrease because of loss of accuracy. While decreasing the number of pixels decreases run time.

- These contrasting factors result in an optimal resolution, where both the utility value as well as the runtime are satisfactory.

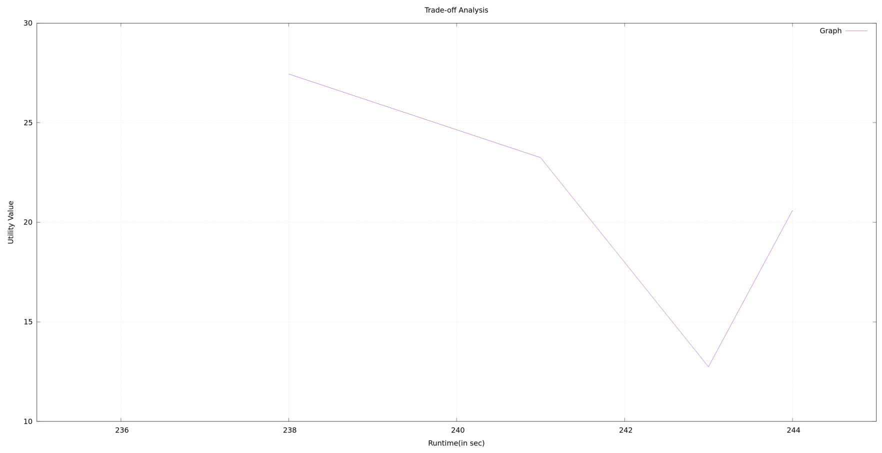

Figure 7: Utility-Runtime plot for different resolutions

- From the graph, we may speculate that the resolution corresponding to the runtime of 238 secs (1366*768) also has the highest utility value and the smallest runtime among other resolutions. So, the optimal resolution should be close to 1366*768.

4.3 Method-3: Spatial Threading

4.3.1 Testcases

x	Runtime	Utility
1	267	2810.94
2	267	779.13
4	230	416.54
8	227	189.84
16	224	90.21

- Increasing the number of threads should decrease the run time which can be seen from the table.

- Also as we discussed before, splitting a frame into separate frames leads to inaccuracy, mainly because of the usage of Gaussian blur. So we expect the utility values to decrease with increase in number of threads.

4.3.2 Graphs

Again same as in method 2, the graphs are very similar to the original case, and differ only in absolute values.

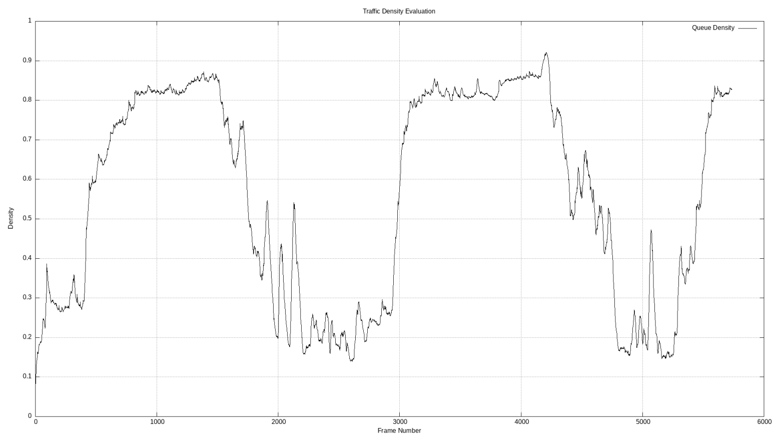

Figure 8: Density vs Frame graph for 4 spatial threads

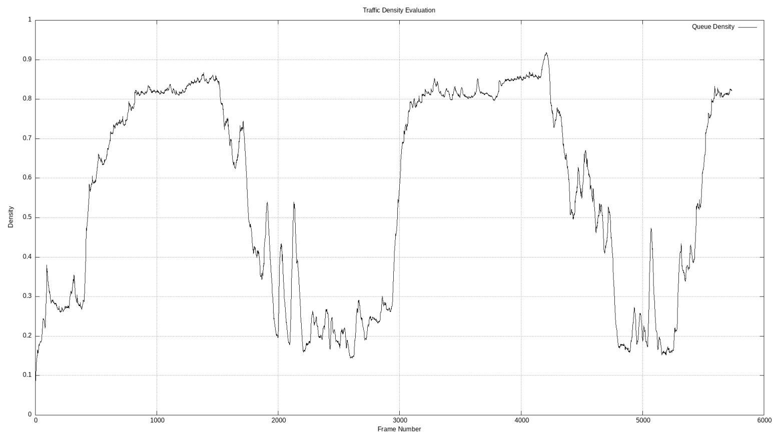

Figure 9: Density vs Frame graph for 8 spatial threads

4.3.3 Trade-off Analysis

The trade-off analysis is also very straight-forward in this method, because the utility values decrease as we try to achieve better runtimes.

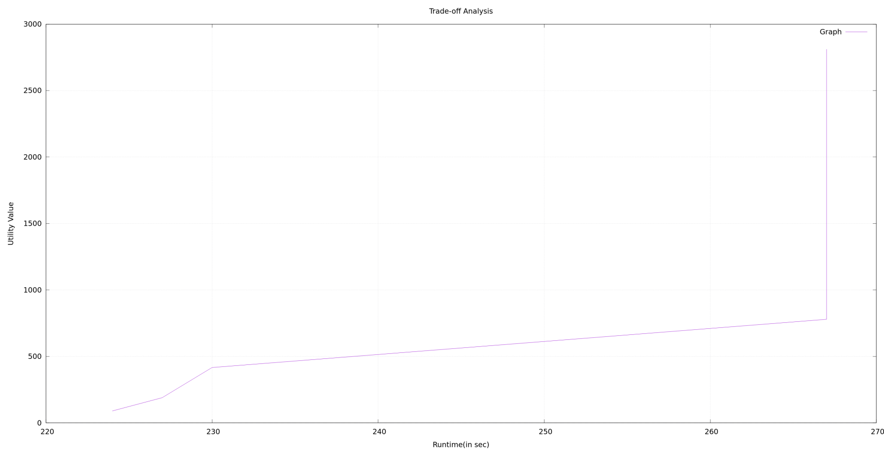

Figure 10: Utility-Runtime plot for spatial threading

Note the high utility value when only one thread is used, which is similar to the original case.

4.4 Method-4: Temporal Threading

4.4.1 Testcases

x	Runtime	Utility
1	181	2653.77
2	175	2814.67
4	178	2674.77
8	173	2602.20
16	176	2727.56

- We observed that the runtime decreases by a significant amount due to temporal threading, as expected.
- But temporal threading is better than spatial threading in this case because of comparatively high utility values, as dividing threads temporally doesn't cause as much inaccuracy as dividing threads into splits.

4.4.2 Graphs

Here also, the graphs are almost identical for different values of threads used and all of them are similar to original values.

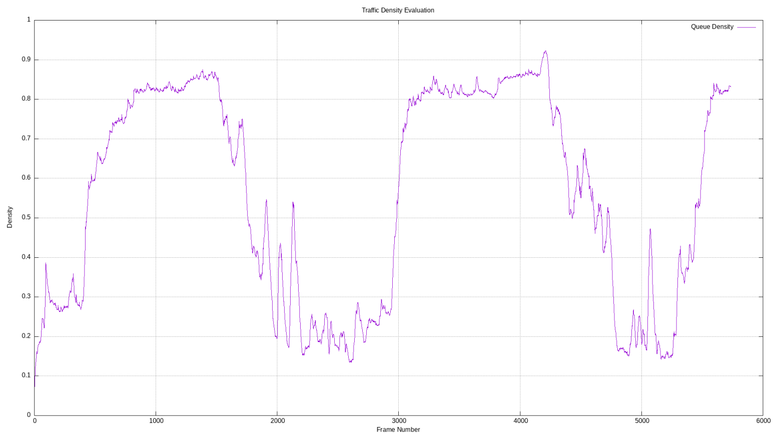

Figure 11: Density vs Frame graph for 8 temporal threads

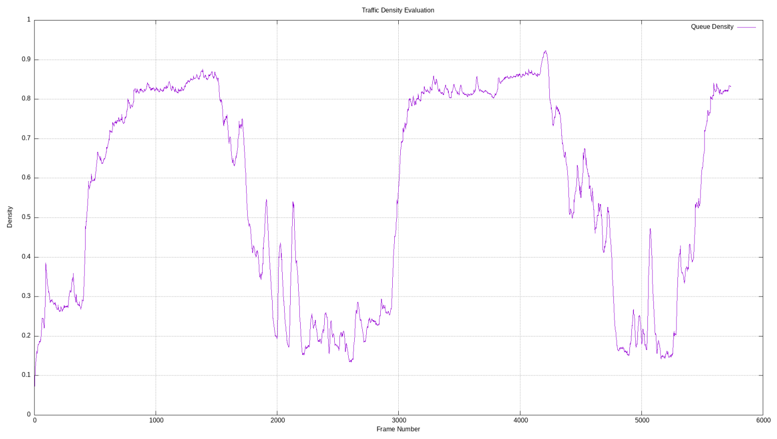

Figure 12: Density vs Frame graph for 16 temporal threads

4.4.3 Trade-off Analysis

The utility values decrease because of a loss of accuracy as compared to the original image but still these values are higher than those obtained in other methods, because the code implementation of temporal threading is such that a whole frame is processed by a single thread, so there is no loss of information and it also doesn't face the issues which are encountered in spatial threading.

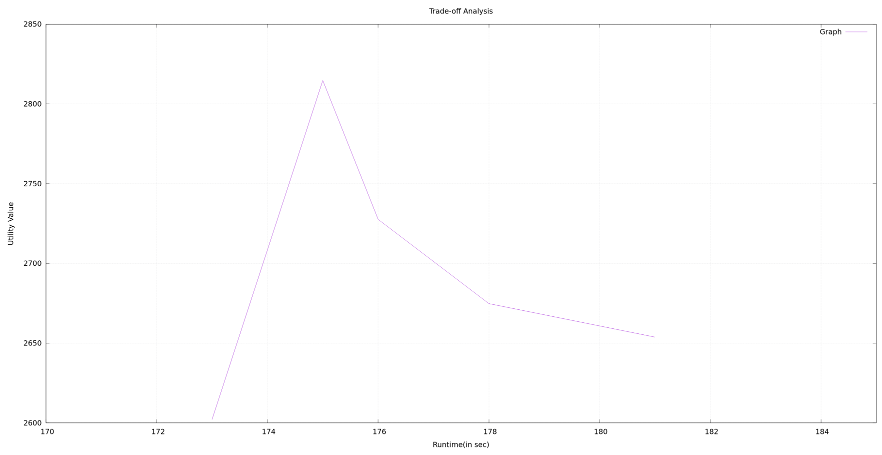

Figure 13: Utility-Runtime plot for temporal threading

5 CPU Analysis

- Besides the factors mentioned above in all the methods, there are many other factors that affect the runtime of any process. For example, we observed that we obtained different runtimes for the same test case when the laptop was on charge as compared to when it was not on charge.
- Another factor which may affect the execution time of different processes of CPU is the temperature of the machine, which is generally dependant upon the number of resources(in terms of memory) which is used by any process in execution.
- We have tried to keep track of CPU usage during the execution of the above methods using an in-built Ubuntu application System Monitor.
- We noticed that in the absence of threading, i.e. in original method, in sub-sampling and in changing resolutions, only a single thread is predominantly active in computation/processing at a time. This can be seen from the figure below:

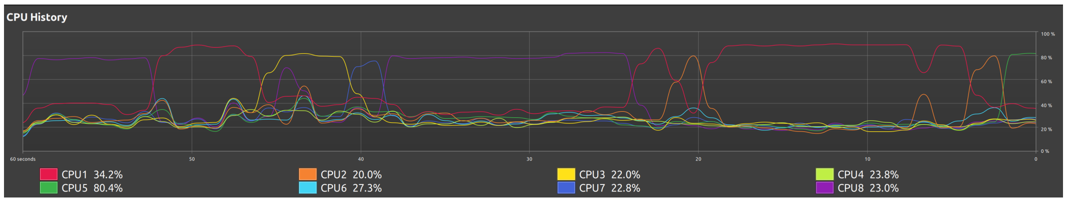

Figure 14: CPU usage in absence of multi-threading

From the figure, only one thread is actively working at 60-80 percent, while all the other threads are relatively idle. But this doesn't mean that the same thread is involved during the execution of the whole program. As we can see from the figure, the thread keeps changing throughout the execution(characterised by different colours), but at a moment, only one thread is active.

- While in methods where multiple threads were used, we observed that during the course of execution, more than one CPU cores are active i.e. working at 60-80 percent at a time, indicating that the computation is indeed parallel.

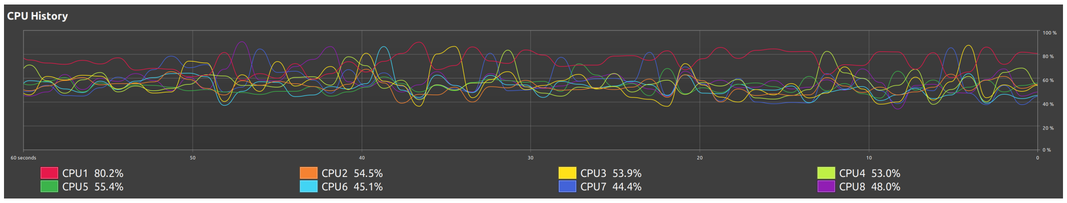

Figure 15: CPU usage in multi-threading