Traffic Density Estimation

Aman Verma, Ishaan Singh

2019CS50419, 2019CS10359

Introduction

In this subtask of Traffic Density Estimation, we have analyzed and done a utility-runtime tradeoff analysis for various methods and parameters. This report is divided into three sections, namely metrics(covers the parameters for juding the program), methods (covers the various methods along with their parameters) and a trade-off analysis (along with the required tables and graphs).

1 Metrics

Due to fewer number of cores, the metrics used in this assignment are utility, parameters passed to the various methods(which in an ideal scenario with no hardware constraints, directly determine runtime), and runtime. Note: We mix and match these metrics for trade-off analysis plots.

The metrics used have been defined below:

1.1 Utility

We have computed a quantity *static_error* to measure the error in static density from the baseline, and a quantity *dynamic_error* to measure the error in dynamic density from the baseline. These errors are RMS values. utilities have been calculated as follows:

$static\_utility = \frac{1}{static\_error}$ and $dynamic\_utility = \frac{1}{dynamic\_error}$ RMS value is computed over the various queue density values over the processed frames. The formula for RMS error is given below (which is static_error or dynamic_error here)

$$RMS\_error = \sqrt[2]{\frac{\sum_{i=1}^{n}(x_i - y_i)^2}{n}}$$

Here, n refers to the total number of frames processed in the baseline method, and x_i, y_i are the values of appropriate densities of the baseline method and the method(with its parameter) for which the error is being computed.

1.2 Runtime

Runtime is an important metric to determine how well a program performs. We define the runtime of a method as follows: The time taken by the method to process the benchmark and produce the final density output per frame, given the parameters as input. Due to the availability of lesser number of cores, we have also used parameter as an important metric. Given no hardware constraints, parameters passed to methods directly determine the runtime and hence seem like a logical choice to pursue. Varying the parameters makes the execution slower/faster.

2 Methods

We have implemented the following methods:

- Method 0: The baseline method - Implements background subtraction and optical flow to calculate ideal values for static and dynamic densities.
- Methods 1 & 2: These methods aim to reduce the work to be performed by losing out on some data. This speeds up the runtime while causing a deviation from the ideal values.

  - Method 1: Skipping Frames allows speeding up the process of analysing the video due to lesser number of frames to be processed. However, lesser number of frames causes a loss of data.
  - Method 2: Resizing frames reduces their resolution. This tends to speed up the analysis process as fewer pixels have to be processed. This too leads to loss of data.

- Methods 3 & 4: These methods aim to make the CPU work more efficiently, by involving all the cores present in the processor. This allows reduction in runtime without affecting the quality of the output.
  - Method 3: Spatial splitting of work. This is achieved by splitting each frame into various parts and applying background subtraction and optical flow on each of these splits separately. This speeds up the process as each thread has to act on fewer pixels.
  - Method 4: Temporal splitting of work. This is achieved by splitting the video into multiple mini-videos and assigning each thread one of these mini-videos to process. This speeds up the process as each thread has to work on lesser number of frames.

- Method 5: We applied the dense and sparse implementations of the optical flow algorithm to calculate the dynamic densities. The sparse algorithm applies heuristics to detect some feature points and calculates the dynamic nature of only these points. This reduces the runtime.

Method	Parameters	Details
0	None	Baseline method - accepts no parameters, calculates the ideal value of static and dynamic densities
1	int x	skip determines the number of frames to skip, a skip of x implies that after processing the N^{th} frame, the $N + x^{th}$ frame will be processed
2	int X, int Y	Reduces the video's resolution from 1920×1088 to $X \times Y$
3	int num_threads	Splits the frames spatially in a manner to divide the work equally among num_threads number of threads
4	int num_threads	Splits the video in a manner to divide it into num_threads number of videos with equal number of frames to be processed by each thread
5	bool sparse	A boolean value determines if the sparse or dense optical flow algorithm is to be run

Table 1: Parameters of methods

3 Trade-Off Analysis

We have analysed the trade-offs for each method separately. For various methods, we have analysed the runtime-parameter and utility-runtime behaviours as and when applicable/necessary.

The static and dynamic density values calculated by the baseline method are shown in the plot below.

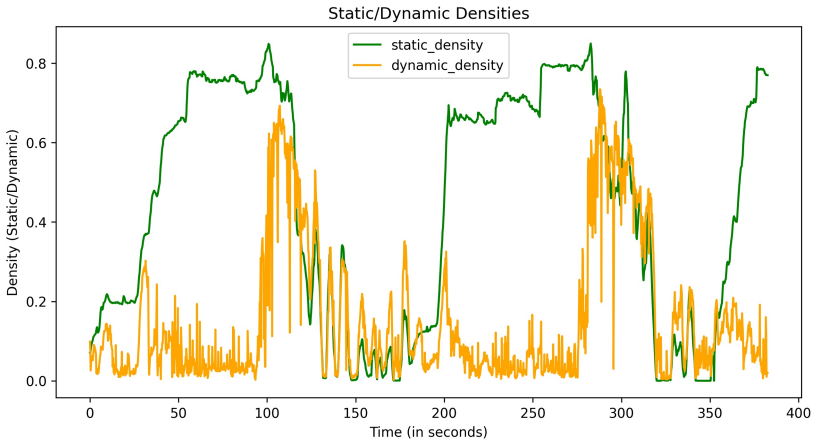

Figure 1: Static and Dynamic Densities vs Time Elapsed

The values of static and dynamic densities as shown in the above method are used for error/utility calculations in our assignment. The runtime for the baseline method is 300s which is also used in our analysis.

3.1 Method 1

In method 1, we have skip_factor as the parameter, which says that 1 out of every skip_factor frames is processed. It is logical to assume that as the number of frames skipped increases, the error will go up and this will bring down the utility of the program. However, this will reduce the runtime of the program significantly since only a fraction of the frames will have to be processed. This will reduce the work that the processor has to perform and is an effective method for reducing run-time.

We have run the program for all values of skip_factor from 1 through 10.
The results are as follows:

skip_factor	static_error	dynamic_error	runtime(s)
1	0	0	300
2	0.00746264	0.0526974	170
3	0.0128076	0.0668927	118
4	0.0173086	0.0787929	103
5	0.02136	0.0820292	81
6	0.0271868	0.084773	76
7	0.0295947	0.0911608	71
8	0.0345959	0.0974428	71
9	0.0398184	0.10339	70
10	0.0416917	0.101696	64

Table 2: Metrics obtained on running method 1 with various parameters

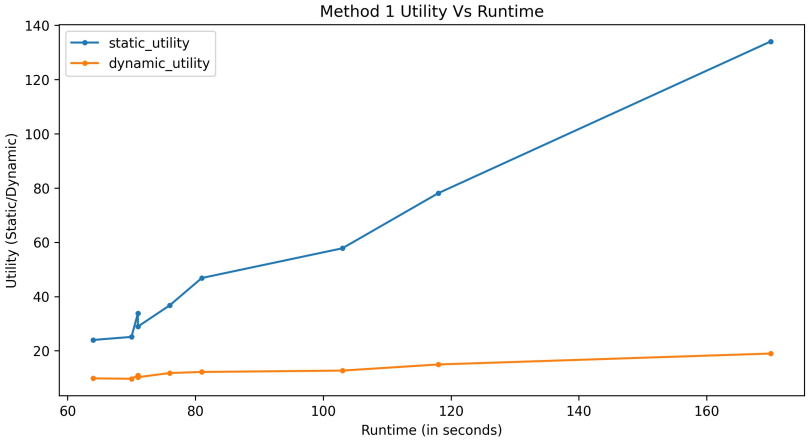

Figure 2: Utility vs Runtime

The plot in Figure 2 clearly indicates that with the increase in run-time both static and dynamic utilities increase. This result is expected since processing more frames allows more accurate results however causes larger runtimes.

The trend indicates a drastic change in the utility of static density, however dynamic utility and hence, the dynamic error are not affected much even with an increase in skip-factor.

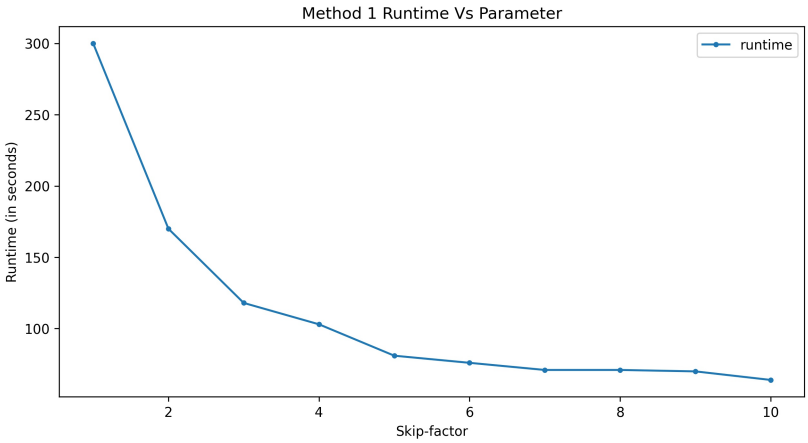

Figure 3: Runtime vs Skip

The plot in Figure **??** shows how the runtime reduces as the skip is increased. Increasing skip results in the processing of a smaller number of frames and thus reduces runtime. The runtime follows an almost inverse trend with the skip factor. The deviation from an ideal inverse is caused due to a certain amount of time required to skip a frame as each frame is encountered sequentially in a VideoCapture object of opencv. The plot tends to saturate around the 60ms runtime mark, which is an exceptional improvement given only a small RMS error of 0.1 in overall values of static and dynamic densities. Thus there is a possibility of improving the runtime by as much as 5 times without much loss of the trends.

The base time to process the frames as seen from Table 2 is 300 seconds. Choosing a skip factor like 2 or 3 only causes a 2% error, while decreasing runtime by around 50%.

frame skipping produces relatively low dynamic_utility which remains more or less constant for different value of skip.

3.2 Method 2

In method 2 we accept 2 parameters, the width and the height of the frame. The frame is then resized to the given values. This reduces the resolution of the frames. Lower resolution improves run-time as fewer number of pixels have to be processed. However, dropping pixels causes a loss of data, which produces inaccuracies in the results. So, the overall parameter which affects the runtime and the utilities is the area or the number of pixels in the resized frame.

Table 3 depicts the values obtained on running the code on various widths and heights. For the purpose of this analysis, we have maintained the aspect ratio of the video while performing the resizing.

width	height	static_error	dynamic_error	runtime(s)
1728	979	0.0581065	0.0229653	333
1536	870	0.0904935	0.0350097	277
1344	762	0.127784	0.0483547	235
1152	653	0.147663	0.0695984	195
960	544	0.146646	0.0969382	167
768	435	0.225168	0.107532	137
576	326	0.28472	0.154942	111
384	218	0.371496	0.203018	86
192	109	0.532251	0.332621	70

Table 3: Metrics obtained on running method 2 with various parameters

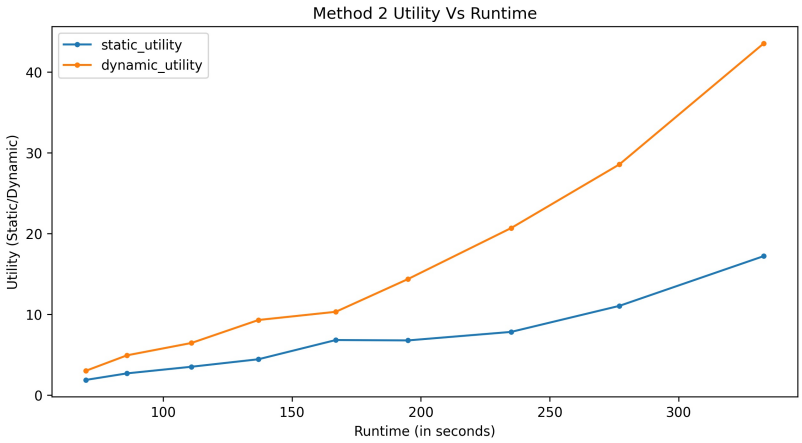

Figure 4: Utility vs Runtime

The plot in 4 Depicts that the static, as well as the dynamic density increases with the increase in runtime as expected.

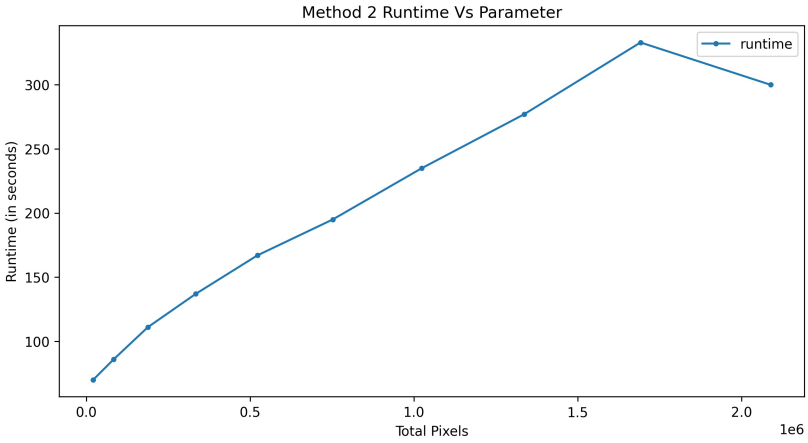

Figure 5: Runtime vs Parameter

The plot in 5 shows the increase in runtime as the total number of pixels to be processed increases. As one would anticipate, the runtime follows a linear relation with respect to the number of pixels to be processed per frame.

3.3 Method 3

Method 3 brings a separate kind of analysis for us to perform. Here, rather than decreasing the quality of analysis, we aim to maximise the use of the processor's capability by splitting work into independent pieces and assigned to independent threads. This is done in method 3 by splitting the frames spatially into various pieces. This assigns independent parts of the frames to each thread to process and allows the processor to function efficiently. Here, the deviation in static density is extremely low as can be seen from the table below.

Ideal results for runtime are not obtained in the analysis due to the availability of only 2 cores on the test machine.

num_threads	static_error	dynamic_error	runtime(s)
1	0	0	300
2	4.28455e-07	0.0411877	275
3	3.12813e-07	0.0547974	324
4	3.31794e-07	0.0693995	371
5	2.70796e-07	0.0763881	443
6	2.82931e-07	0.0818528	506
7	2.57715e-07	0.0984694	576

Table 4: Metrics obtained on running method 3 with various parameters

Although we have included the graph for utility vs parameter and utility vs runtime for method 3 in our model results, a detailed analysis has not been done since the values for utilities are pretty high (since we just split our work spatially). In case of dynamic utility, there is one observation that that it increases as the number of threads increases. This is because in dense optical flow calculations, for each of the pixels at the edge of a sub-rectangle of the frame, an error is introduced since we partition the frame spatially and this error increases as the number of threads / partitions increases.

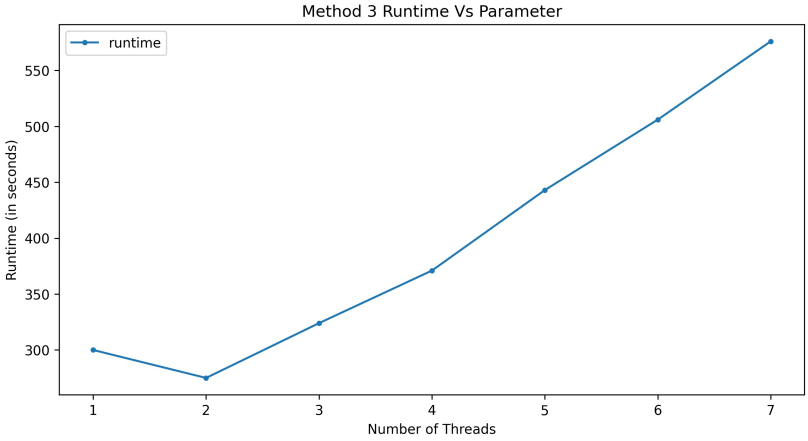

Figure 6: Runtime vs Parameter

Contrary to our expectation, in Figure 6 runtime increases as the number of threads increase. This is because of the reason stated above: that is the availability of only 2 cores on the host machine. However, there is another contributing factor. In this analysis, the splitting is not completely independent as even if the static density and dynamic density calculations require processing over a small number of pixels, we need to perform identical operation for applying homography and cropping to all of the frames. This increases the amount of redundant computations and thus does not create much room for optimisation. In this situation, the overhead of creating and managing multiple threads exceeds the benefit of splitting work across independent threads. Therefore the runtime increases as the number of threads increases.

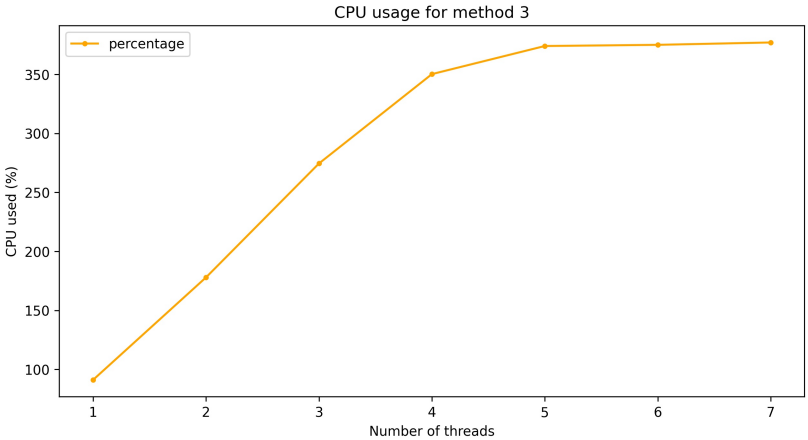

Figure 7: CPU Usage

Here in Figure 7, we can first see that as the number of threads increases, the percentage of CPU used increases linearly and then it begins to saturate. Initially, we are able to divide the task into parallel subtasks across various threads properly and hence more CPU is used. However, due to the limitation of number of cores of our systems (which are 2), addition of more and more threads does not help in breaking our computation parallelly effectively and relatively lesser CPU is used.

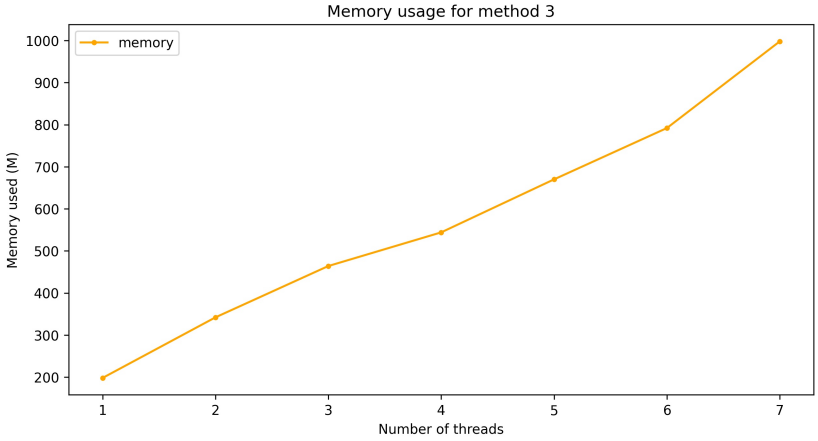

Figure 8: Memory Usage

Here in figure 8, we can see that as the number of threads increases, memory usage increases approximately linearly. This can be attributed to the fact that each creation of each thread requires a fairly constant amount of memory allocation in our computer.

3.4 Method 4

Similar to method 3, here we split the work temporally. This is done by splitting the benchmark into small parts and giving each thread a part to process. This eliminates any repetion of work across threads and so runtime is optimised efficiently with even a large number of threads.

num_threads	static_error	dynamic_error	runtime(s)
1	0	0	300
2	0.00444818	0.0264425	194
3	0.00424213	0.0312232	174
4	0.00468027	0.0342674	163
5	0.00503726	0.0341636	162
6	0.00520873	0.0431371	162
7	0.00520928	0.0409633	163

Table 5: Metrics obtained on running method 4 with various parameters

Again, we do not analyse the Utility vs Runtime or Utility vs Parameter plots. However, we do see that the static utility is pretty high, and the static_error is of the order of 10^{-3}. While, the dynamic_error is still low, it is higher as compared to the static case. It increases with number of threads again due to the boundary conditions - the optical flow algorithm loses information at the boundary of these mini-videos.

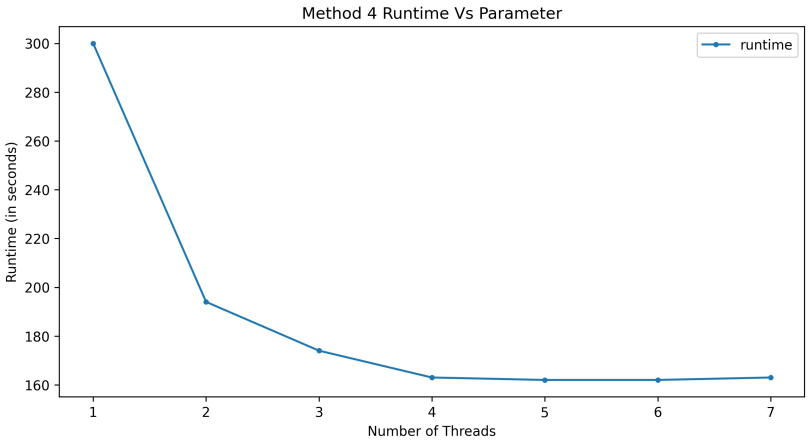

Figure 9: Runtime vs Parameter

Here is is evident from the plot that the runtime decreases as the number of parameters increase. This shows an inverse nature as expected. This is because the work is divided equally among a number of threads and this optimises the functioning of the processor. However, as more and more threads are involved, a saturation is reached due to the limitation of the cores of our system (which are two).

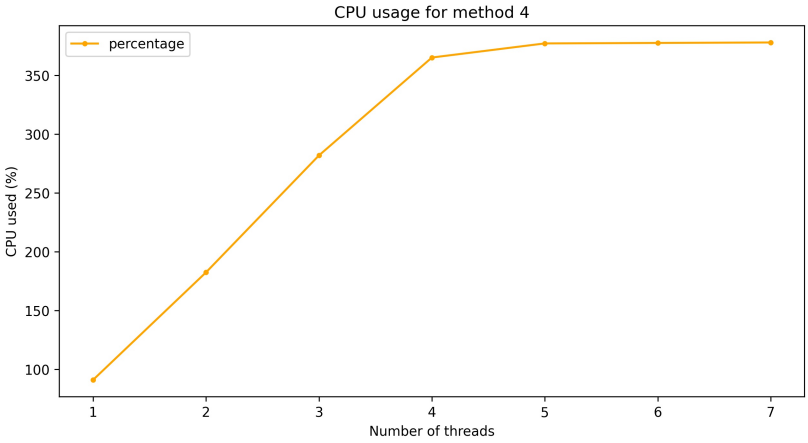

Figure 10: CPU Usage

As the number of threads increases, the CPU usage increases linearly and then saturates. Initially, we are able to divide the task into parallel subtasks across various threads properly and hence more CPU is used. However, due to the limitation of number of cores of our systems (which are 2), addition of more and more threads does not help in breaking our computation parallelly effectively and relatively lesser CPU is used.

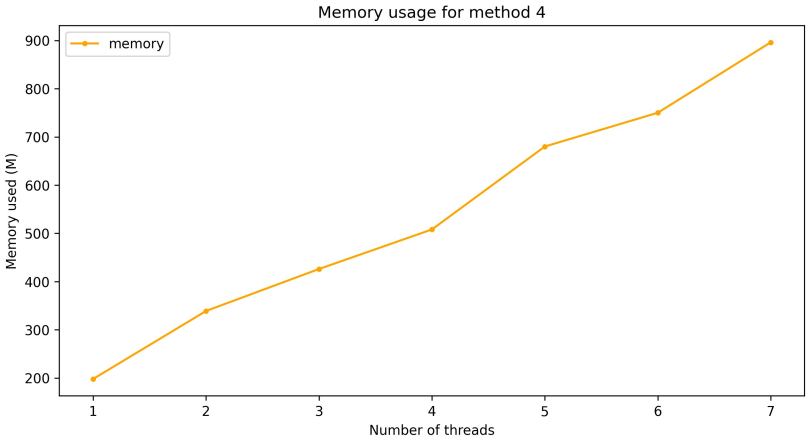

Figure 11: Memory Usage

Here in figure 11, we can see that as the number of threads increases, memory usage increases approximately linearly. This can be attributed to the fact that each creation of each thread requires a fairly constant amount of memory allocation in our computer.

3.5 Method 5

Optical Flow Algorithm	dynamic_error	runtime(s)
dense	0	300
sparse	0.121669	200

Table 6: Sparse vs Dense

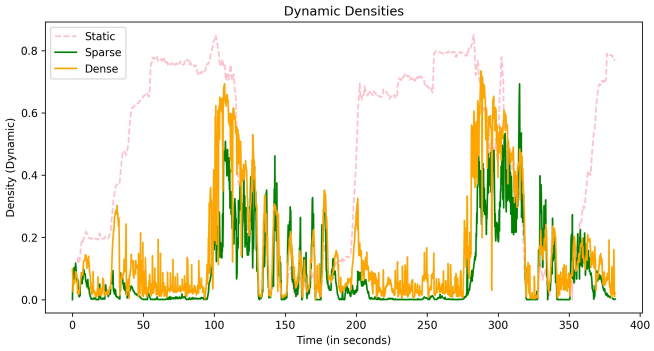

Figure 12: Sparse vs Dense Optical Flow

This plot shows that the Dense Optical Flow Algorithm is able to capture more features as compared to the Sparse Algorithm.
On comparing the plot with the video, it is seen that because the sparse algorithm selects a few feature points and determines dynamic density using those, the time taken to execute the program over all frames is reduces drastically. However this comes with a drawback. At many instances, the movement of a single vehicle or motorcycle is totally missed. Only, when a significant number of pixels show movement is the sparse algorithm able to capture the flow. Both the sparse and dynamic optical flow algorithm are able to track motion of vehicles around the light's color change as clearly indicated by the plot above. This explains the large value of the error of the sparse optical flow algorithm.

Thus, the sparse algorithm is better if one wishes to track bulk movement of traffic due to it's lower runtime, whereas, the dense algorithm provides much more accurate results for smaller changes.

3.6 Comparative Analysis

Overall, we observe that method 1 provides really fast runtimes(60-100s) while providing a decent utility.

Method 2 also provides great runtimes, however the utility drops quite a bit as compared to Method 1 as the resolution of the video decreases. Thus Method 1 seems like a better alternative.

Thus, skipping frames decreases runtime significantly while minimising the deviation from original values. Multithreading allows us to split work across frames. From our analysis, it was clear that method 3 required better hardware to support more threads due to more amount of redundant computation. Method 4 is better since it does not involve overheads of splitting and applying homography multiple times on a single frame. Thus method 4 reduces the runtime quite significantly (runtime is reduced to nearly half after using 4 threads) , although, it does give some errors accounting to the edge factors.
Method 5 (the bonus method) which uses sparse optical flow for dynamic density calculations clearly reduces the runtime as the algorithm selects only some feature points for flow computation instead of all the pixels. However, since it selects lesser number of points, we get some errors as compared to the dense optical flow.

4 Conclusion

In this assignment, first of all we learnt about using various openCv functions and libraries to process videos, and learnt how to crop and warp various video frames. Using these tools, we calculated static and dynamic queue densities on a given road intersection. Through this assignment, we also became familiar with the concepts of baseline, benchmark, methods and parameters. Moreover, we learnt about multi-threading and analysed its behaviour using various methods and then through CPU and memory usage. Through the implementation of various methods across different parameters, we were able to perform a utility-runtime or runtime-parameter analysis which was an enriching experience. Moreover, we became quite familiar with other auxillary skills/tools such as plotting graphs using python 3. Hence, overall, this assignment was a very good technical learning experience.