Contents

Chapter 1

Metrics

1.1 Runtime Metric:

The metric used to analyse and compare runtime of a method for a given parameter is simple and straightforward, i.e. the **total elapsed time** (in **s**). This elapsed time is also referred to as the wall-clock time, which is basically the total real world time that a program (or a task) takes to complete its execution. For our analysis, the **task** is to estimate queue and moving density on the benchmark (video). So, the elapsed time **only** includes the time the program takes to complete traffic density estimation (both queue and moving).
We have measured the wall-clock time for different methods and parameters using chrono library. Code template used:

```
// header files
#include <chrono>
using namespace std::chrono;
// code
// code
auto start = high_resolution_clock::now();
// task_run<method, param>(video)
// queue and moving density estimation
auto end = high_resolution_clock::now();
float time = duration_cast<milliseconds>(end - start).count()/1000.0;
// remaining code
```

According to C++ reference manual, **high_resolution_clock** is the clock with the shortest tick period available. Therefore, we have used this clock in our code for **accurate** and **precise** results. Also, in order to make our analysis more tolerant to lucky/unlucky program runs (in terms of wall-clock time), we report runtime of the task (for a given method and parameter) as the **average** wall-clock time of the task over **five runs**.
Note: We give sufficient sleep time between successive runs to avoid heating up of the processor.

Final metric reported: For a given method and parameter -

$$\textbf{Runtime (in s)} = \frac{\text{Total elapsed time } <\textbf{task}> \text{ over five runs (in } \textbf{s})}{\textbf{5.0}} \tag{1.1}$$

1.2 Utility Metric:

The utility metric is a little more complex as far its definition is concerned. To define it, we first need to define the notion of ***associated frame***. This frame is defined a bit differently for method 1 (sub-sampling frames), whereas, the definiton is the same for the other three methods.

1.2.1 Method 1:

In our baseline code, we are processing every third frame of the video. This means that starting from frame number 1, we process frames numbered 4, 7, 10, 13, .., 5737 **(total number of frames in the video = 5737)**. Now, in this method, we process every **x** frame, where **x** is the method parameter (integer). So, we basically process frames numbered 1 + x, 1 + 2x, 1 + 3x, .. (not exceeding 5737), assuming first frame is numbered 1.
So, for every $\boldsymbol{a} \in \{4, 7, .., 5737\}$, I define its corresponding *associated frame number* as $\boldsymbol{b}$, such that $\boldsymbol{b}$ is the greatest element of the set $\boldsymbol{\{y = 1 + nx : (n \in N) \wedge (nx \leq 5736) \wedge (y \leq a)\}}$, and the frame numbered $\boldsymbol{b}$ is defined as the ***associated frame***.

1.2.2 Method 2, 3, 4:

In these methods, we are processing the same set of frames, i.e. frames numbered {4, 7, .., 5737}, therefore, the *associated frame number* is the same as the given frame number, and the ***associated frame*** is the same as the **given frame**.

1.2.3 Metric definition:

Let **QueueX[y]** and **MovingX[y]** represent the queue and moving density values determined at frame number y using method X. **X** can take two values:- B and P. **B** denotes baseline method, and **P** denotes parametric method (which we need to analyse). Also, for a given frame number $\boldsymbol{x \in \{4, 7, .., 5737\}}$, let assoc[x] denote its *assosciated frame number*, which is defined above. We define two error metrics as follows-

$$\mathbf{QueueErrorP} = \sqrt{\frac{\sum_{i=1}^{i=1912}(\mathbf{QueueB[1+3i]} - \mathbf{QueueP}[\text{assoc}[1 + 3i]])^2}{\mathbf{1912}}} \tag{1.2}$$

and,

$$\mathbf{MovingErrorP} = \sqrt{\frac{\sum_{i=1}^{i=1912}(\mathbf{MovingB[1+3i]} - \mathbf{MovingP}[\text{assoc}[1 + 3i]])^2}{\mathbf{1912}}} \tag{1.3}$$

Here **P** is indicative of the method and parameter used for density estimation. Also, since we have not determined queue and moving density values for frame number 1 in any of the methods/baseline, we define $\mathbf{QueueX[1]} = \mathbf{MovingX[1]} = \mathbf{0}$.

Now, we define our utility metrics as follows-

$$\textbf{QueueUtilityP (in util)} = \frac{\textbf{10}}{\textbf{0.1} + \textbf{QueueErrorP}} \tag{1.4}$$

and,

$$\textbf{MovingUtilityP (in util)} = \frac{\textbf{10}}{\textbf{0.1} + \textbf{MovingErrorP}} \tag{1.5}$$

Here, **util** is the unit in which we measure the utility of a parametric method. It is clear from the definition of the utility metric that when P = B, then QueueUtilityB = MovingUtilityB = **100 util**. Also, for all other parametric methods P, non-negative **QueueErrorP** and **MovingErrorP** necessarily imply that the utility will always be **less than** or **equal** to **100 util** (Queue or Moving). Hence, this metric gives us a performance measure out of 100 which is kind of **relative** to the **baseline performance** (100 util).

Chapter 2

Parametric Methods:

We have implemented all the four methods that were listed on the course website, and have analysed both queue and moving utility/runtime. The methods have been described below along with the parameters that they take:-

2.1 Method 1: Sub-sampling frames

In this method, we take in a single parameter **x** that indicates to the function that every x^{th} frame of the video must be processed, starting from frame number 1. In other words, only frames with frame number coming from the set $\{1 + nx : nx \leq 5736 \wedge n \in \{1, 2, 3, ...\}\}$ will be processed for queue/moving density estimation. So, if the input parameter is set to $\boldsymbol{x = 3}$, then it basically implies the baseline code, as we process every third frame in the original code.
Constraints on x: (x is a positive integer) $\wedge$ (x $\leq$ 5736)
The utility and runtime metric have already been defined for this method in the previous chapter. Also, the processing strategy remains similar to the one used in baseline code, the only difference being that we are now processing every x^{th} frame instead of every 3^{rd} frame.

2.2 Method 2: Reduce resolution

In this method, we take in two parameters **A** and **B** that indicates to the function that the resolution of every frame of the video must first be changed to **A x B**, and then it must be processed for queue/moving density estimation. The resolution of the image used for processing in baseline code was **329 x 779**. So, if the input parameters are A = 329 and B = 779, then it basically implies the baseline code, as we process images which are of the same resolution.
Note: This method is independent of method 1, and we still process every 3^{rd} frame as we did in our baseline procedure.
Constraints on A and B: (A is a positive integer) $\wedge$ (B is a positive integer)

2.3 Method 3: Spatial split (pthread)

In this method, we take in a single parameter **n** that indicates to the function the number of pthreads that must be used in parallel to do the processing and traffic density estimation. This method splits each frame into **n** *rectangular strips* (horizontal dimension is the same as the original image, but vertical dimension (height) is divided almost equally into **n** parts). Each rectangular strip is passed to a different thread, and all the threads process their frame portion in parallel.

The total traffic density (for a frame) is then estimated by just adding up the area values obtained from different threads, and dividing them by the total image area.
Since, we have increased the number of workers (pthread), we expect a decrease in the runtime measure, but at the same time, little work and too many workers can also lead to chaos. So, we also expect that this decrease in runtime will not be uniform as we increase the value of **n**.
Constraints on n: (n is a positive integer) $\wedge$ (n $\leq$ 16)

2.4 Method 4: Temporal split (pthread)

In this method, we take in a single parameter **n** that indicates to the function the number of pthreads that must be used in parallel to do the processing and traffic density estimation. This method splits the entire video into **n** smaller videos of almost equal duration (or equivalently frame count), and passes these videos to different threads so that they can process them in parallel. After processing, individual data files are merged together (appended in order) to get the total traffic density estimation. Since, the duration of the video along with frame-by-frame processing (transforming, cropping etc.) eats up a major chunk of execution time (as well as wall-clock time), running videos of smaller duration on different threads should lead to a significant decrease in the runtime of the task. Also, since different threads are just processing different sections of the same video and at the same FPS, it is also natural to expect that the utility measure will be the same as the baseline utility (i.e. 100 util). Hence, this method is expected to have good utility-runtime tradeoffs, with reasonable parameters.
Constraints on n: (n is a positive integer) $\wedge$ (n $\leq$ 16)

Note: The utility and runtime metric have already been defined for method numbers 2, 3 and 4 in the previous chapter. Also, each method is independent of one-another, and there is no mixing amongst them, like temporal with spatial, temporal with reduced resolution etc.

Chapter 3

Utility-Runtime Analysis

In this section, we describe the data that we obtained by running different methods with different parameters, and justify the utility and runtime values that we got for each one of them.

3.1 Method 1

Parameter values used for data collection: **x = 2, 3, 5, 7, 9, 11, 13 and 15**. Following values of utility and runtime were determined using the corresponding metric:

Parameter (x)	Queue Utility (in util)	Moving Utility (in util)	Runtime (in s)
2	88.7826	73.1395	32.9584
3	100.0	100.0	27.2424
5	78.9499	71.0651	21.8984
7	68.7207	59.6505	19.5268
9	62.9703	53.6923	18.3814
11	57.6563	49.2512	17.6578
13	54.0681	46.2773	17.1626
15	51.646	43.9963	16.7838

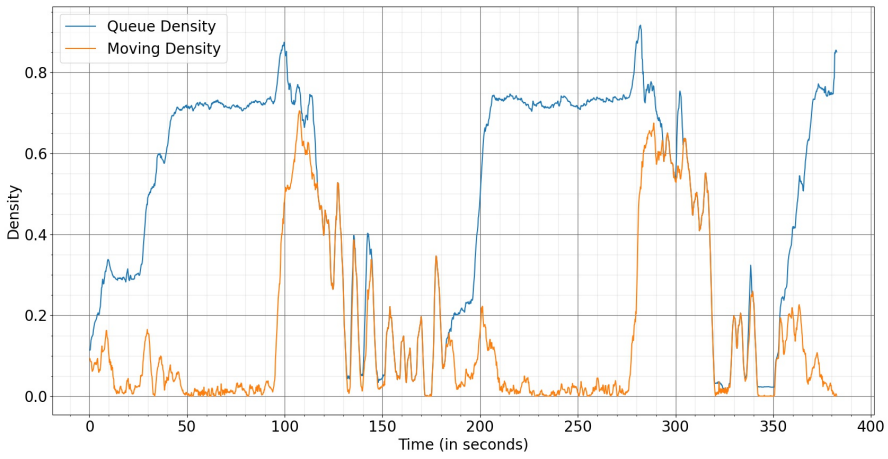
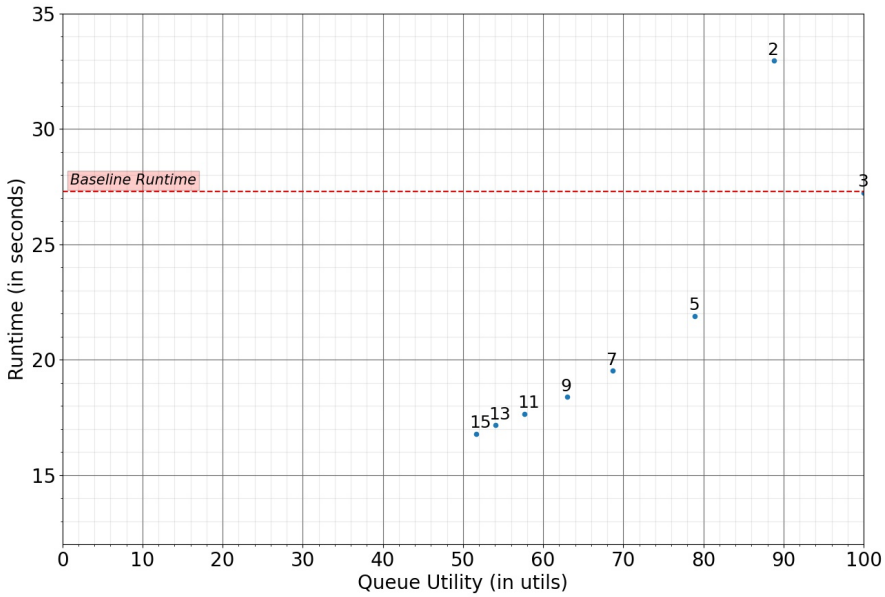
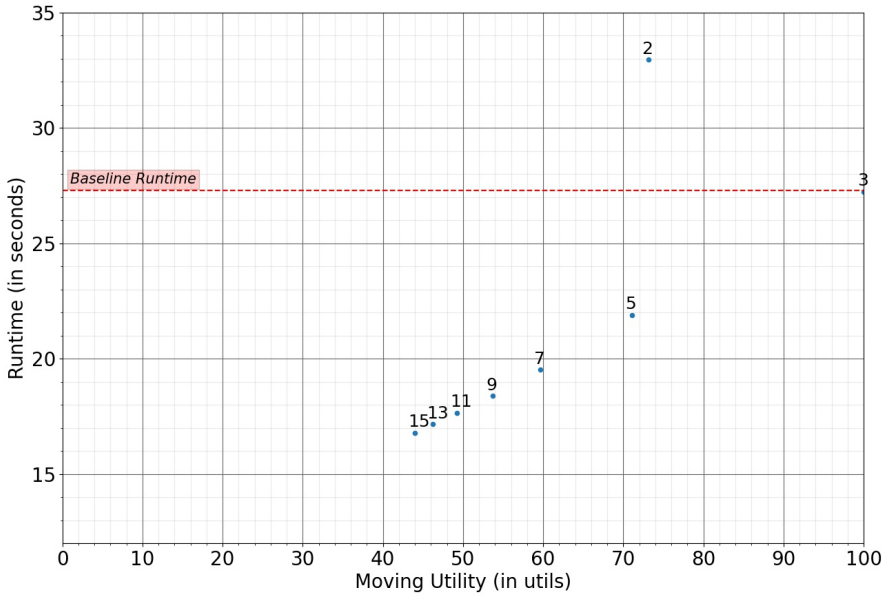


Figure 3.1: Traffic density estimation graph for x = 3 (Utility = 100 util)

Parameter values have been marked above the points



(a)



(b)

Figure 3.2: (a) Queue Utility vs Runtime (b) Moving Utility vs Runtime

From Figure 3.2, it is clear that both the utility and runtime of the task decreases as we increase the value of the parameter (**x**). On the other hand, if we decrease the value of **x** to less than 3, then there is an increase in runtime and decrease in utility. This is not the kind of tradeoff that we want, as we are compromising both the factors.

3.1.1 Runtime Analysis:

For parameter values greater than 3, we are dropping more frames as compared to baseline procedure. Due to this reason, the total runtime of the task decreases, as now we are processing less number of frames. For e.g. we were processing 1912 frames in the baseline procedure, but for $x = 5$, we process only 1147 frames, which is 765 frames less than baseline. Also, each frame takes significant time in processing- transforming, cropping, subtraction and density calculation. Therefore, less frames to process necessarily imply less wall-clock time or runtime. So, as the value of **x** increases, we expect a decrease in runtime and that is what we get from our analysis. Similarly, if the parameter value is less than 3, we are processing more number of frames than baseline, hence the runtime should increase. This increase can be seen in the graph for $\boldsymbol{x = 2}$ (Figure 3.2)
Note: The decrease in runtime will not be uniform, i.e. we cannot lower down runtime arbitrarily by just increasing the value of **x**, as there are some fixed overheads that will always be there no matter what the parameter is. These include- going through each and every frame of the video and processing background and first frames. Since, the video itself is of more than 6 min in duration, time spent in these activities cannot be considered to be negligible, and hence will always be there (unaffected).

3.1.2 Utility Analysis:

As far as utility values are considered, there is going to be a significant decrease in utility as the value of **x** increases. This is because we are dropping a lot more frames and consequently,losing preciseness. In other words, for some intermediate frame (in baseline), we use the most recent frame available till that frame for the given parameter, which may or may not reflect the traffic situation at current frame, hence leading to errors. This error (or decrease in utility) is more **pronounced** when we are calculating **moving density**, as moving density is sensitive to motion of vehicles at that frame and it is something that varies drastically from one frame to another. For e.g, static vehicles may remain static for a long duration of time (red light), in such a case the queue density will remain nearly the same over a set of frames, however, moving traffic varies from one frame to another, hence, we expect greater errors (lesser utility) in such a case. From Figure 3.2, this logic can be easily verified for both queue and moving utilities.

3.2 Method 2:

Parameter values used for data collection: **(A, B) = (66, 156), (82, 195), (110, 260), (165, 779), (165, 390), (329, 390), (329, 779) and (395, 935)**. Following values of utility and runtime were determined using the corresponding metric:

Parameter (A, B)	Queue Utility (in util)	Moving Utility (in util)	Runtime (in s)
(66, 156)	48.9847	52.8501	21.923
(82, 195)	52.1538	55.9369	22.0583
(110, 260)	58.7298	62.7446	22.4446
(165, 390)	69.8735	74.868	23.4863
(165, 779)	79.6387	83.0482	24.9
(329, 390)	83.2374	86.7301	25.807
(329, 779)	100.0	100.0	26.891
(395, 935)	91.1818	92.9473	29.2753

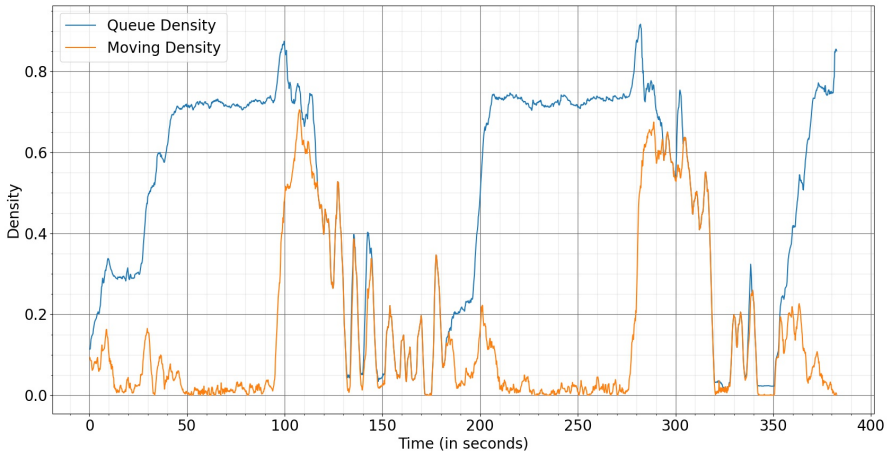
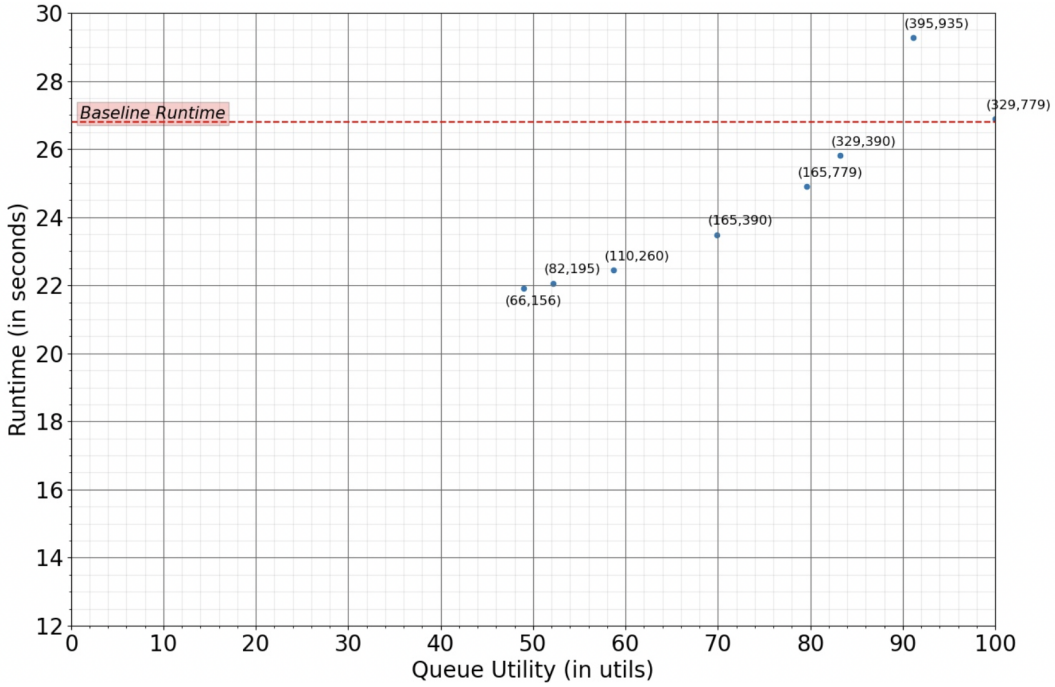
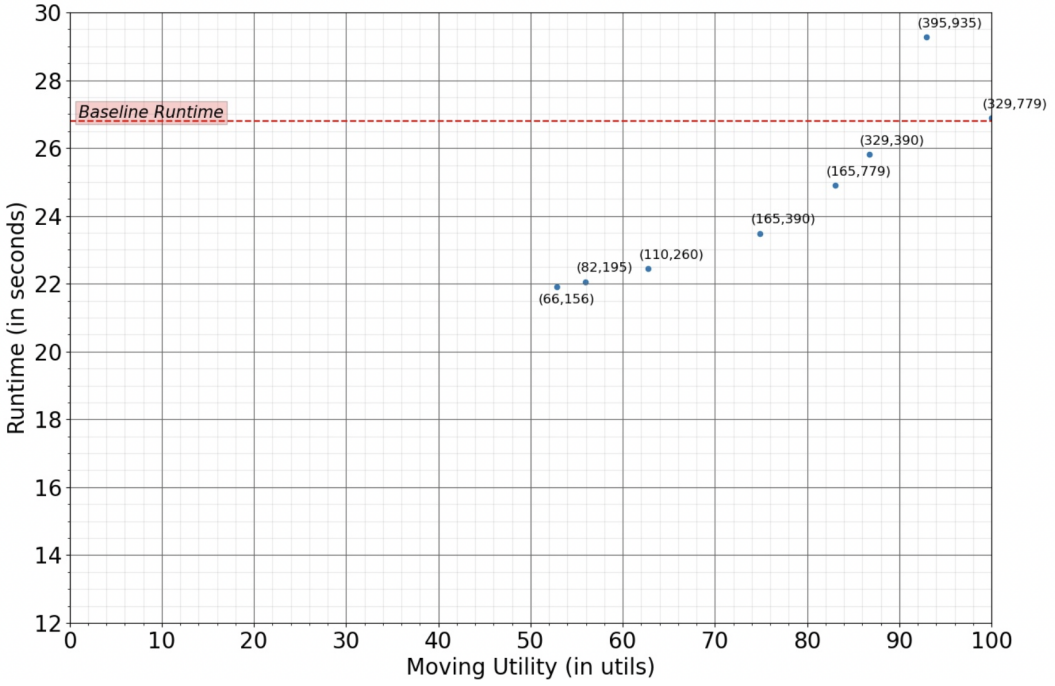


Figure 3.3: Traffic density estimation graph for (A, B) = (329, 779) (Utility = 100 util)

Parameter values have been marked above/below the points



(a)



(b)

Figure 3.4: (a) Queue Utility vs Runtime (b) Moving Utility vs Runtime

From Figure 3.4, it is clear that both the utility and runtime of the task decreases as we decrease the value of the parameter **A** or **B**. On the other hand, if we increase the value of both **A** and **B** to more than (329, 779), then there is an increase in runtime (as we are increasing total pixel count) and decrease in utility. This is not the kind of tradeoff that we want, as we are compromising both the factors.

3.2.1 Runtime Analysis:

Reduced resolution means less pixels to process in image subtraction and area calculation. This reduces the total runtime. For eg. Baseline testing has the resolution (329, 779) with 256,291 pixels in each frame. For a resolution of (165, 390) each frame has 64,350 pixels which is approximately one-forth the initial pixel count. Less pixels reduce processing time which imply less runtime. So, as the resolution reduces, we expect a decrease in runtime, which we observed in our analysis above. Similarly, for an increased resolution (interpolated) the runtime increases, as we are increasing the pixel count. This increase can be seen in the graph for **(A, B) = (395, 935)**. **Note**: The decrease in runtime will not be uniform, i.e. we cannot lower down runtime arbitrarily by just decreasing the resolution, as there are some fixed overheads that will always be there no-matter what the parameter is. These include going through each and every frame of the video and then warping and cropping every third frame. Since, the video itself is of more than 6 min in duration, time spent in these activities cannot be considered to be negligible, and hence will always be there (unaffected).

3.2.2 Utility Analysis:

Utility decreases significantly due to loss of information in lower resolution frames. We believe that lower resolution increases the estimated traffic area which consequently reduces the utility. This increase in traffic estimation is possibly due to the fact that we have used erode and dilate functions in our image processing, which use pixel squares of size 5 and 21 respectively. These dimensions were used keeping in mind the original resolution of the image, therefore, in case of lower resolution frames, these functions produce erroneous white pixels (as the pixel square has not been scaled down appropriately). Similarly, when we increase resolution (A, B) = (395, 935), we get that the traffic area is reduced, as now less white pixels are being generated (as the pixel squares have not been scaled up). Also, the utility drop is more in case of queue density as compared to moving density. This can be easily explained, as on an average, the white pixel count in queue frame is greater than that in moving frame. Due to this reason, the erroneous white pixels that are generated in case of queue frame are greater than that in case of moving frame (as erode/dilate depend on neighbouring pixels). Hence, queue utility is less than moving utility in general. Further we noted that, reducing the pixel count to half along X and Y direction lead to similar change in utility values (and runtime) indicating information is equally important along both the axial directions. For eg. reducing pixels along x axis to half (165, 779) and along y axis to half (329, 390) have similar utilities and runtime values.

3.3 Method 3:

Parameter values used for data collection: **n = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 and 16**. Following values of utility and runtime were determined using the corresponding metric:

Parameter (n)	Queue Utility (in util)	Moving Utility (in util)	Runtime (in s)
1	100.0	100.0	27.124
2	98.6312	99.1886	25.864
3	97.6539	98.5049	24.117
4	96.2885	97.9245	23.023
5	95.5566	97.2253	23.059
6	94.5196	96.6488	23.308
7	93.0373	96.0493	22.746
8	92.1478	95.5113	22.836
9	91.3768	94.7905	22.786
10	90.0227	94.2905	22.816
11	88.9658	93.7235	22.697
12	88.6217	93.1349	23.042
13	87.3925	92.5481	23.342
14	86.4374	92.0159	22.723
15	86.5132	91.4361	23.52
16	84.8643	90.9827	26.116

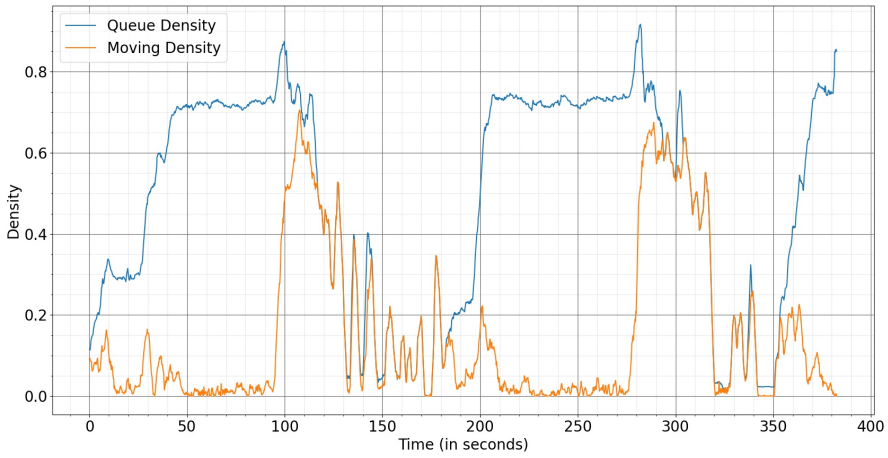
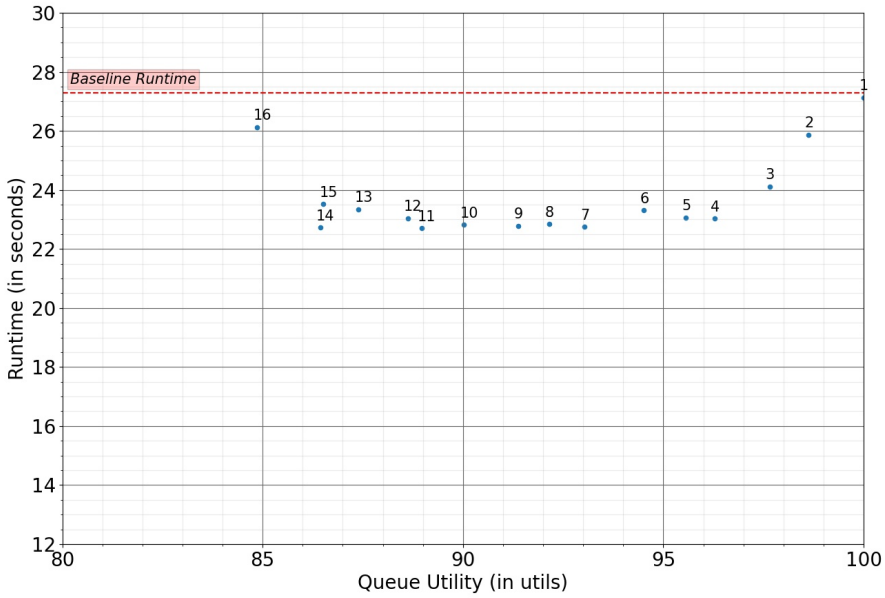
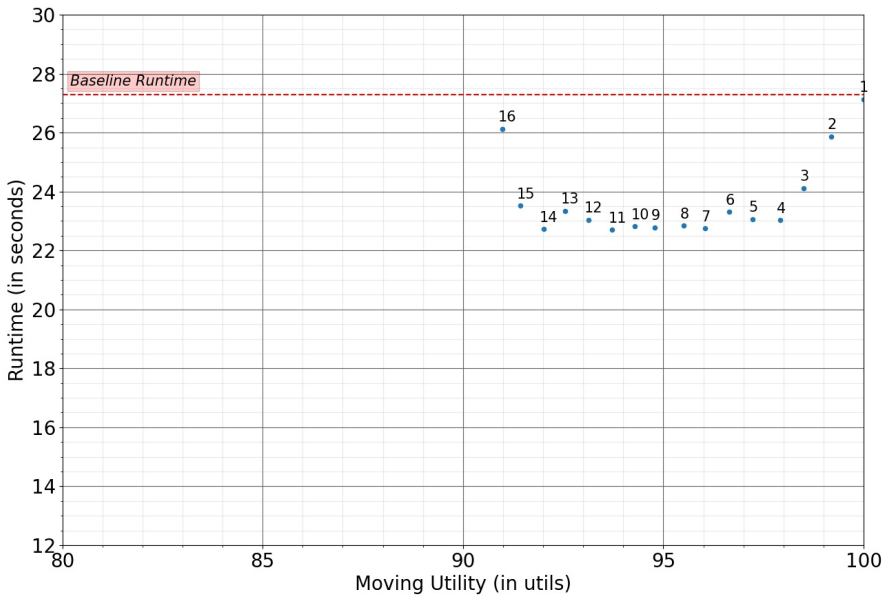


Figure 3.5: Traffic density estimation graph for n = 1 (Utility = 100 util)

Parameter values have been marked above the points



(a)



(b)

Figure 3.6: (a) Queue Utility vs Runtime (b) Moving Utility vs Runtime

3.3.1 Runtime Analysis:

We are dividing each frame of the video into **n** smaller rectangular strips of almost equal height and passing each strip to a different thread. This way, we expect the overall runtime of the task to decrease as we increase the value of **n** (because threads run in parallel). This trend is easily seen in Figure 3.6 in which the runtime is decreasing as the number of pthreads is increasing from 1 to 4. From n = 4 to n = 6, the runtime of the method remains almost the same, and then decreases when n = 7. Again from n = 7 to n = 11, the runtime of the method remains almost the same (within 1 place of decimal). We can explain this behaviour by noting that there are some overheads present in creating and joining threads. So, as we increase the number of threads, and decrease the portion of every frame that a thread processes, we also increase the creation and joining time. The joining time is the main factor here, as we join the threads in a fixed order (using for loop), but that may not be the exact same order in which the threads finish their processing. Hence, there is also this wait time that comes into the picture, as we might be waiting for a thread to finish its job even when there are some other threads which are already done with their job. So, the expected decrease in runtime is almost completely compensated by this increase in overhead time. Increasing the number of pthreads ($\mathbf{n \geq 12}$) further leads to an increase in runtime. Hence, we analysed utility/runtime only upto $\mathbf{n = 16}$ threads. This runtime increase is probably due to the increased time spent in pthread creation and pthread join. Apart from these overheads, there is another factor which is crucial in this analysis, and that is the hardware used for testing. Hardware used for testing: Macbook Air with 8 CPU cores. So, it is not surprising that we are getting high CPU utilization when $\mathbf{n = 8}$ (1 thread per core) and numbers around it, like 7, 9, 10, 11. Also, as **n** increases, the overhead time also increases, as now the number of threads are more than the number of CPU cores on which they can run. Further, we note that the time spent by 1 pthread is lower than the baseline runtime. This is somehow related to the fact that pthreads have a more optimised CPU utilisation in general, which we confirmed by using **top** command on Ubuntu.

3.3.2 Utility Analysis:

In spatial split, we are splitting each frame into **n** equal height (almost) *rectangular strips*, and processing them in parallel. Here, **n** denotes the number of pthreads which will be used. Processing a part of a frame involves image subtraction and area calculation. However, the density estimation also depends on neighbouring pixels, as we have used erode and dilate functions in image processing. Hence, as we increase the number of rectangular strips, we also increase the number of pixels present at the boundary. Hence, the effective number of white pixels that we get after splitting is different than the original amount. For most cases, the effective white pixel area is slightly less than the baseline. This is because, some pixels which were earlier compensated for by dilation which followed erosion, are now not compensated at all, since they are present on the boundary and hence, may not have neighbouring white pixels. Therefore, the density estimation is not exact, which leads to a decrease in utility. As expected, the utility (both queue/moving) decreases as we use more pthreads as we are increasing boundary pixel counts with each split. Also, we observe that the utility in case of queue density estimation decreases much more rapidly

than in case of moving density estimation. This is mainly because there is a high queue density for the majority of the time (due to red light). Hence, expected chances of a stationary vehicle to be present at a rectangle boundary is much more than the corresponding expected chance of a moving vehicle. Hence, erroneous pixel loss in case of queue density estimation is much more than that in case of moving density estimation. Consequently, queue utility is less than moving utility. As expected, this difference in utility increases as we increase the number of rectangular strips (and consequently, boundary pixels).

3.4 Method 4:

Parameter values used for data collection: **n = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 and 16**. Following values of utility and runtime were determined using the corresponding metric:

Parameter (n)	Queue Utility (in util)	Moving Utility (in util)	Runtime (in s)
1	100.0	100.0	25.785
2	100.0	100.0	23.3422
3	100.0	100.0	18.9656
4	100.0	100.0	15.8196
5	100.0	100.0	15.7036
6	100.0	100.0	15.1544
7	100.0	100.0	15.1332
8	100.0	100.0	15.005
9	100.0	100.0	15.022
10	100.0	100.0	16.372
11	100.0	100.0	16.911
12	100.0	100.0	17.561
13	100.0	100.0	18.027
14	100.0	100.0	18.158
15	100.0	100.0	18.413
16	100.0	100.0	18.384

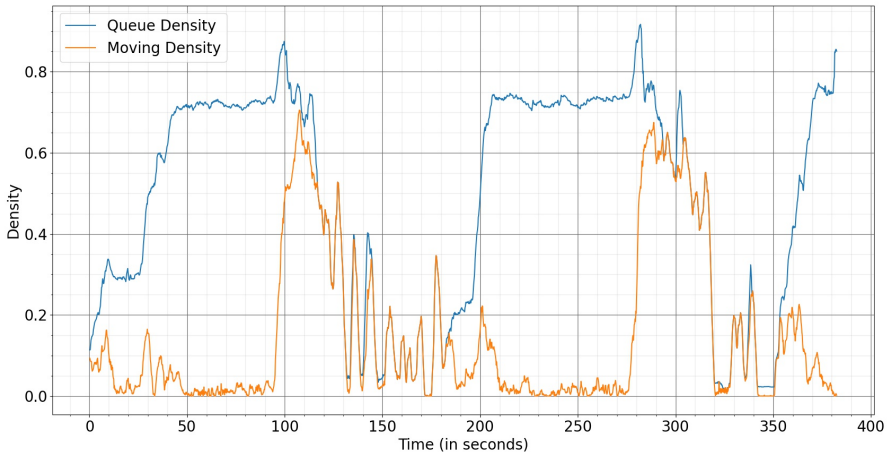
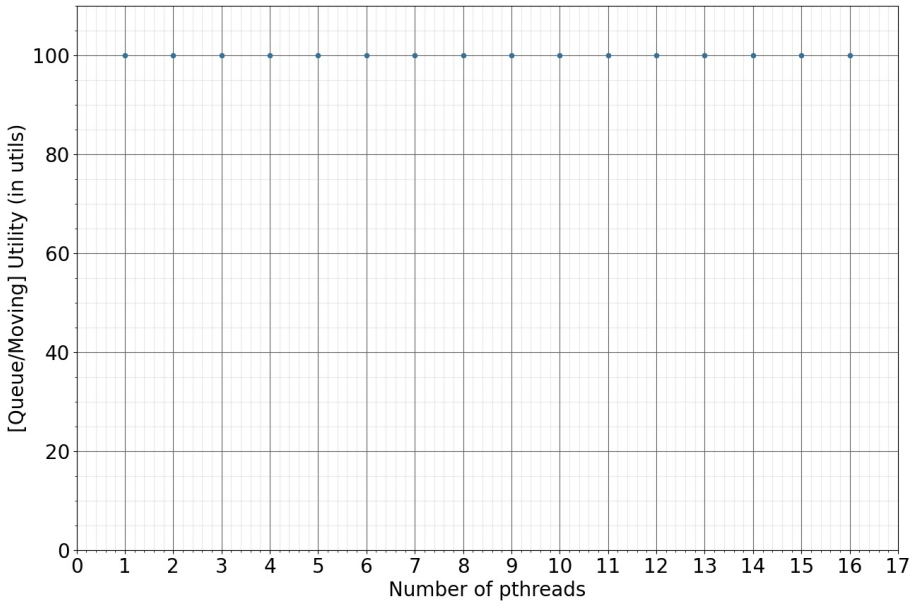
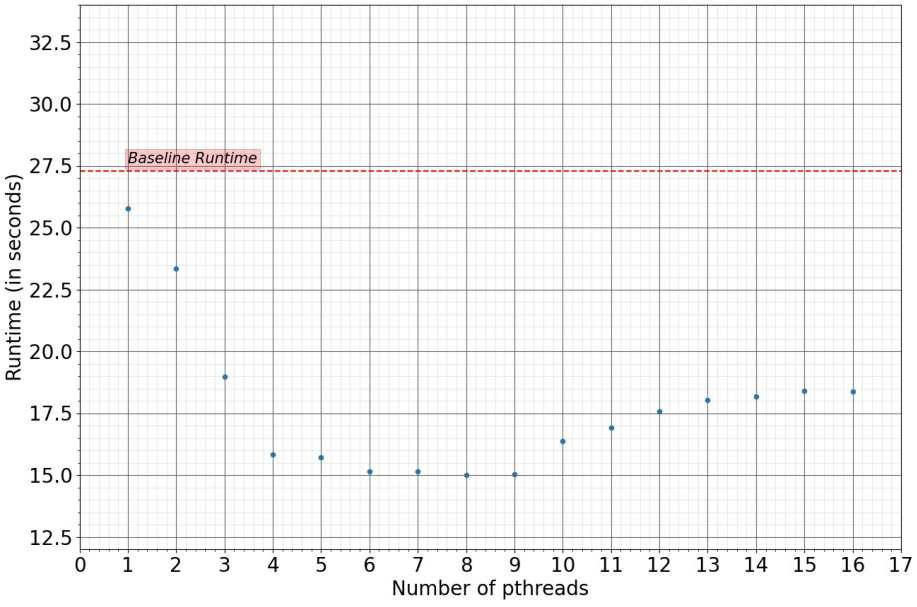


Figure 3.7: Traffic density estimation graph for n = 1 (Utility = 100 util)



(a)



(b)

Figure 3.8: (a) [Queue/Moving] Utility vs Number of pthreads (b) Runtime vs Number of pthreads

3.4.1 Runtime Analysis:

We are dividing the entire video into **n** videos of smaller duration and passing each sub-video to a different thread. This way, we expect the overall runtime of the task to decrease as we increase the value of **n** (because threads run in parallel). This trend is easily seen in Figure 3.8(b) in which the runtime is decreasing as the number of pthreads is increasing from 1 to 6. From n = 6 to n = 9, the runtime of the method remains almost the same, achieving minimum value at **n = 8**. We can explain this behaviour by noting that there are some overheads present in creating and joining threads. So, as we increase the number of threads, and decrease the video duration, we also increase the creation and joining time. The joining time is the main factor here, as we join the threads in a fixed order (using for loop), but that may not be the exact same order in which the threads finish their processing. Hence, there is also this wait time that comes into the picture, as we might be waiting for a thread to finish its job even when there are some other threads which are already done with their job. So, the expected decrease in runtime is almost completely compensated by this increase in overhead time. Increasing the number of pthreads (**n $\geq$ 10**) further leads to an increase in runtime. Hence, we analysed utility/runtime only upto **n = 16** threads. This runtime increase is probably due to the increased time spent in pthread creation and pthread join. The saturation effect is explained by the balance between multi-threaded gains and creation-join losses. Apart from these overheads, there is another factor which is crucial in this analysis, and that is the hardware used for testing. Hardware used for testing: Macbook Air with 8 CPU cores. So, it is not surprising that we are getting high CPU utilization when **n = 8** (1 thread per core). Also, as **n** increases, the overhead time also increases, as now the number of threads are more than the number of CPU cores on which they can run. Further, we note that the time spent by 1 pthread is lower than the baseline runtime. This is somehow related to the fact that pthreads have a more optimised CPU utilisation in general, which we confirmed by using **top** command on Ubuntu.

3.4.2 Utility Analysis:

The video is simply split up into smaller duration for each pthread. Hence, we are still processing the same set of frames and at the same FPS. Also, the image resolution is the same and there are no spatial splits. Hence, the method is exactly the same as baseline, except the fact that the work is now divided equally among **n** threads. This results in utility of **100 utils** for every value of the parameter (both queue and moving).

3.5 Interpreting Utility values:

Interpreting runtime values is pretty straightforward as they reflect the real-world time. So faster a code, the better it is. However, interpreting the utility metric that we have defined may not be that straightforward as it doesn't relate to any concrete real-world entity, except the fact that more is the error, lesser is the utility. But, what the metric doesn't tell is that what values of utility are acceptable, what are poor, what is the expected error associated with a utility value, and so on. Also, it may so happen that the traffic estimation is very good for majority of the duration, but only varies drastically in some regions, which brings down the utility value as a whole. Therefore, in order to make the metric more concrete, we mention some common details regarding the metric which can be used while making an informed decision:

Utility Range (in util)	Avg. Absolute Error (per frame point)	Performance	Graph Access
90-100	0.005	Excellent	Not required
80-90	0.018	Very Good	Not required
60-80	0.043	Good	Optional
50-60	0.081	Average	Optional
40-50	0.122	Below average	Required
0-40	0.400	Poor	Required

This table can be used as a reference to judge whether or not a particular parametric method is producing good results or not as far as absolute values of density are concerned. One can also opt to just see the graph for himself/herself and make a decision, just to be on the safe side. Graphs which compare queue/moving density of a parametric method with baseline method are given in the **Analysis** folder for reference.

Chapter 4

Conclusion: Trade-offs

In each of the methods described above (except method 4), we observed a trade off between utility and runtime, i.e., if we decrease the runtime of a method, we also decrease its utility. So, we can make some informed decision regarding what parameters we should be using for optimal execution of our program. However, what we consider as optimal, itself will vary from one application to another. Hence, to conclude something, we need to have some kind of a definition. For this report, **we consider parameter P to be optimal for a given method, if its utility is greater than or equal to 60 utils and it has the least runtime among all parameters that satisy the former.** Below, we have stated the optimal parameters for each of the method (1-3) based on this definition:

Method 1- Sub-sampling Frames: **x = 9** (Only Queue density)
. **x = 5** (Both Queue and Moving density)
Method 2- Reduce resolution: **(A, B) = (110, 260)** (Only Moving density)
. **(A, B) = (165, 390)** (Both Queue and Moving density)
Method 3- Spatial split (pthread): **n = 11** (n = 7, 8, 9, 10 also produced similar results)
. (Both Queue and Moving density)

As far as method 4 is concerned, there are actually no utility-runtime trade-offs, as the utility is **100 util** for each parameter value (after taking care of boundary conditions). Hence, **optimal parameter will be the one with the least runtime.** So, optimal parameter:
Method 4- Temporal split (pthread): **n = 8** (n = 6, 7, 9 also produced similar results)
. (Both Queue and Moving density)

Note: Our experimentation is not exhaustive, and we have not considered all the possible parameter values. We have simply analysed utility-runtime trade-offs for a fixed set of parameters, and tried to reason out their values by means of plotting graphs and understanding multi-threaded behaviour.

Chapter 5

Appendix

5.1 Background Subtraction vs Dense Optical Flow

In Task 1(b), we estimated moving traffic density on the given stretch of road using both dense optical flow (Farneback's algorithm) and background subtraction (involves image subtraction and processing). The moving density comparison plot obtained is shown below:

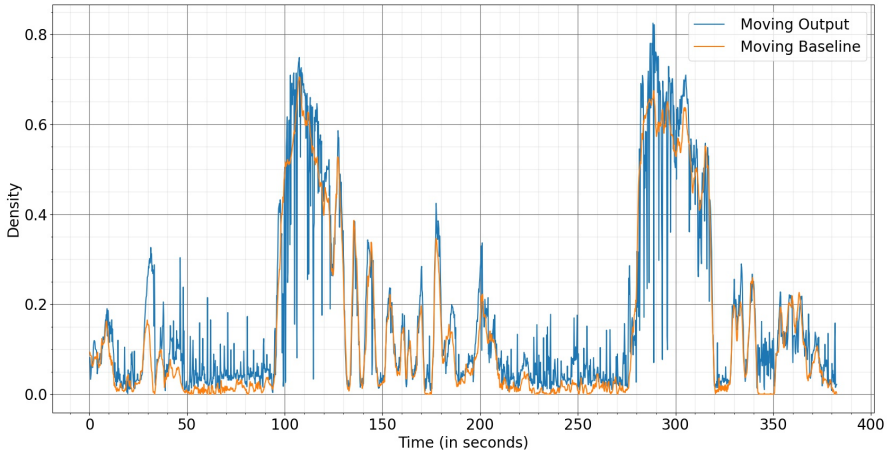


Figure 5.1: Dense Optical Flow (blue), Background subtraction (orange)

From the figure, it is clear that the overall shape of the density estimation plot is similar for both techniques, however, there is significant noise in the plot obtained using dense optical flow. Also, note that in our dense optical flow, we did do noise removal using **erode** function. Still, there exists significant noise. Utility and Runtime values for dense optical flow are given below:
Moving Utility - 56.2053 util; **Runtime -** 95.807 s (baseline runtime is $\sim$ **27 s**)
From these values, it is clear that the runtime of dense optical flow is more than three times than that of background subtraction (baseline). At the same time, its utility is not that good either because of noise. Hence, there is **no utility-runtime tradeoff** in this method (optical flow), as utility has decreased significantly and the runtime increased to more than three times. **Therefore, we used background subtraction instead of dense optical flow in Task 1(b).**