COP290 Task 1 Sub-task 3

Utility Run Time Trade-off Analysis

T Abishek & Gaurav Jain
2019CS10407 & 2019CS10349

2068 Words 1st April, 2021

1 Introduction

In task 1, sub-task 2, we had implemented code to calculate queue density (density of all vehicles queued (either standing or moving) which are waiting for red signal in the straight stretch of road going towards north) and dynamic density (density of those vehicles which are not standing but moving in that same stretch).
In this sub-task, our aim is to modify our previous attempt using various methods to optimize the code for various objectives, like better run time, while also not sacrificing the accuracy of the task. We aim to find (subjectively), what method and parameters provides the best trade-off between better run-time and good accuracy for this task of computing queue density.

2 Metrics

We mainly care about our run-time and accuracy and thus, initially we have to define these 2 metrics before analysing them.

- **Run-time** is defined as the time difference between the start time and end time of the program which calculates and outputs the queue density, taking in the traffic video as input.
- **Utility** is defined as the difference between the average queue density (calculated in sub-task 2) and the root mean square of queue density difference calculated at all frames divided by average queue density. Mathematically,

$$Utility = \frac{AvgQueueDensity - RMSError}{AvgQueueDensity}$$

where,

$$RMSError = \sqrt{\frac{(NewQueueDensity - OriginalQueueDensity)^2}{No.of Frames}}$$

3 Methods

We implement various methods to reduce run time and have good accuracy and analyse which methods with its respective parameters, have what kind of trade-offs between run-time and accuracy while also comparing it against the baseline of the initial code's run-time and output *(without skipping frames)*. Higher accuracy implies that the output is more closely matching the initial code's output.

3.1 Method 1

Sub-sampling frames: Instead of processing every frame present in the video, we skip **x** no. of frames between every frame that is processed to decrease the run-time. The parameter in this method is **x**, the no. of frames skipped. Based on **x**, utility and run time metric are calculated and the trade-off between utility and run-time is analyzed.

3.2 Method 2

Sub-sampling resolution: Instead of processing every frame at 400×1000 resolution, we apply a scaling factor *y* and process the frames at $(400 * y) \times (1000 * y)$ resolution. This decreased the computation required and thus decreases run-time. The parameter in this method is *y*, the scaling factor applied. Based on *y*, utility and run time metric are calculated and the trade-off between utility and run-time is analyzed.

3.3 Method 3

In this method, after applying homography on all frame, each frame is split into *N* horizontal strips and passed to *N* different pthreads. Each thread then processes the frame for the queue density. The parameter in this method is *N*, the number of threads. Based on *N*, utility metric and run time metric are calculated and the trade-off between utility and run time is analyzed.

3.4 Method 4

In this method, work is split temporally across threads (application level pthreads), by giving consecutive frames to different threads for processing. The parameter in this method is *N*, the number of threads. Based on *N*, utility metric and run time metric are calculated and the trade-off between utility and run time is analyzed.

4 Trade-off Analysis

4.1 Method 1

The results of the analysis after using Method 1 for trade-off analysis:

	x=1	x=2	x=3	x=4	x=5	x=6	x=7	x=8	x=9	x=10
Run Time	78.93	52.58	44.75	38.65	36.24	34.34	33.00	32.27	31.32	30.95
Utility	1.000	0.992	0.989	0.984	0.980	0.976	0.971	0.968	0.964	0.960

Table 1. Utility v/s Run time Analysis using Method 1.

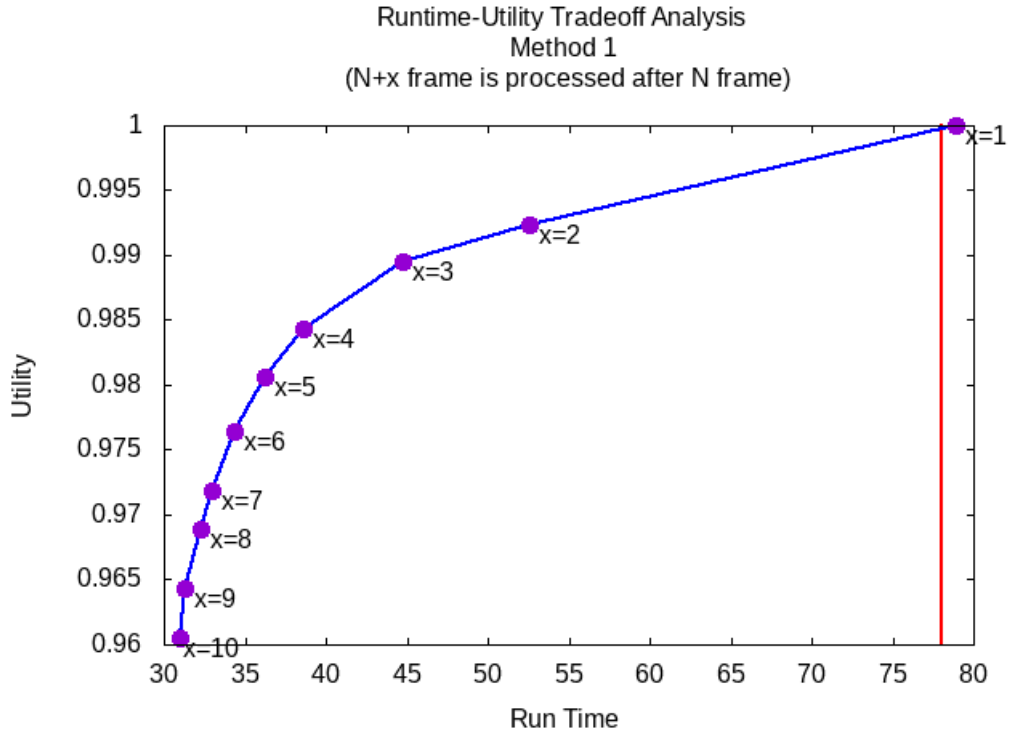

Fig. 1. Utility v/s Run time Analysis using Method 1.

Observations:

- The red line indicates the baseline performance. As we can see from the graph, run-time and utility matches baseline when x=1, which is an expected result as no frames are skipped while computing, which is the same as baseline.
- And as we increase the no. of skipped frames between every frame that is computed, the no. of frames that are being processed to compute queue density reduces which leads to a reduction in run-time, and this trend matches with the result obtained. The utility too decreases as we increase the no. of skipped frames, as there is less data to work with while computing queue density as more frames are skipped, which leads to less accuracy in the output and thus more error.
- As we can see from the graph, utility starts dropping sharply from x=3 with more reduction in utility but comparably not a substantial reduction in run-time and thus it would be the best choice when implementing this method. The utility being at 0.989 further solidifies this choice as the error is really negligible.

4.2 Method 2

In method 2, we reduce the resolution of the projected and cropped frame which is used for calculating the queue density using Background Subtraction. The results of this analysis are given below:

	y=0.05	y=0.20	y=0.35	y=0.50	y=0.65	y=0.80	y=0.95	y=1.0
Run Time	45.33	51.84	56.85	62.39	67.05	71.56	75.17	78.28
Utility	0.982	0.996	0.998	0.998	0.999	0.999	0.999	1.0

Table 2. Utility v/s Run time Analysis using Method 2.

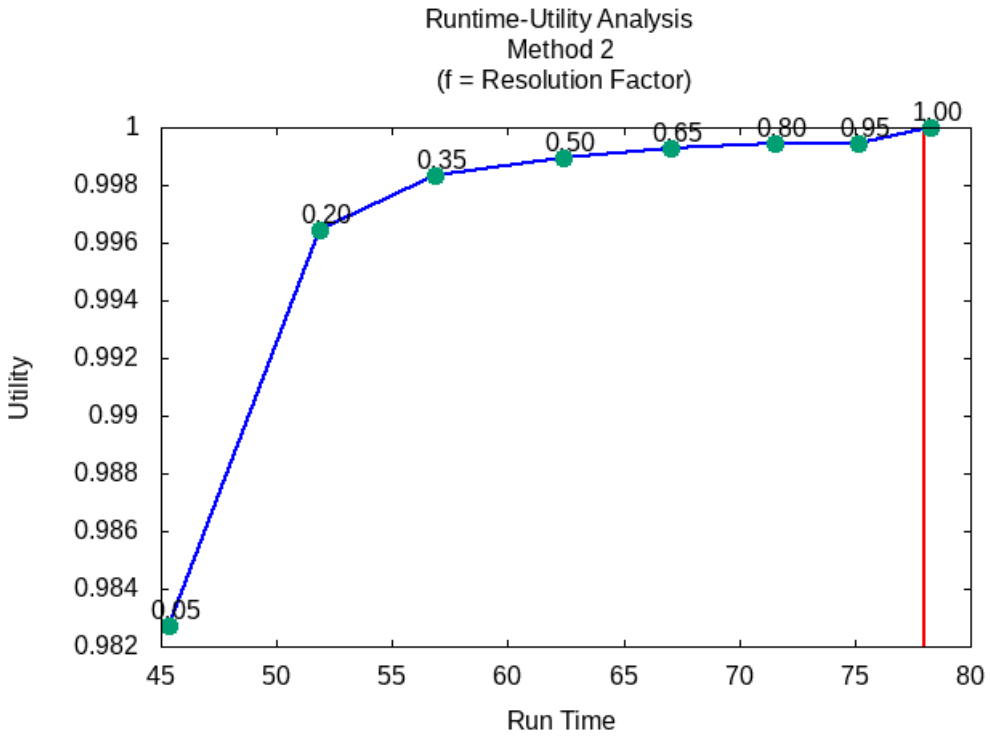

Fig. 2. Utility v/s Run time Analysis using Method 2.

Observations:

- The red line indicates the baseline performance. As, we can see from the graph, run-time and utility matches the baseline when scaling factor is 1 (same resolution as baseline), which is an expected result.
- And as we reduce the scaling factor, the resolution of the image that is being processed also decreases reducing the computation required and thus leads to lower and lower run-time as we reduce the scaling factor which matches with the result. The utility too decreases as we reduce the scaling factor because as the resolution decreases, there is less data to work with while computing queue density which leads to less accuracy of the output data and thus more error.
- As we can see from the graph, utility drops sharply without much reduction in run-time when scaling factor reduces from 0.20, and thus this would be the best choice when implementing this method on this machine. The error is also negligible as the utility is 0.996 which further solidifies our choice.

4.3 Method 3

The results of the analysis after applying method 3 in which frames are split into N horizontal strips where N represents the number of threads used for computation and each strip is processed by a thread. $N = 1$ is same as the baseline case.

	N=2	N=3	N=4	N=5	N=6	N=7	N=8
Run Time	84.32	81.37	79.41	76.81	75.11	76.99	77.76
Utility	0.999	0.998	0.999	0.999	0.995	0.993	0.999

Table 3. Utility v/s Run time Analysis using Method 3.

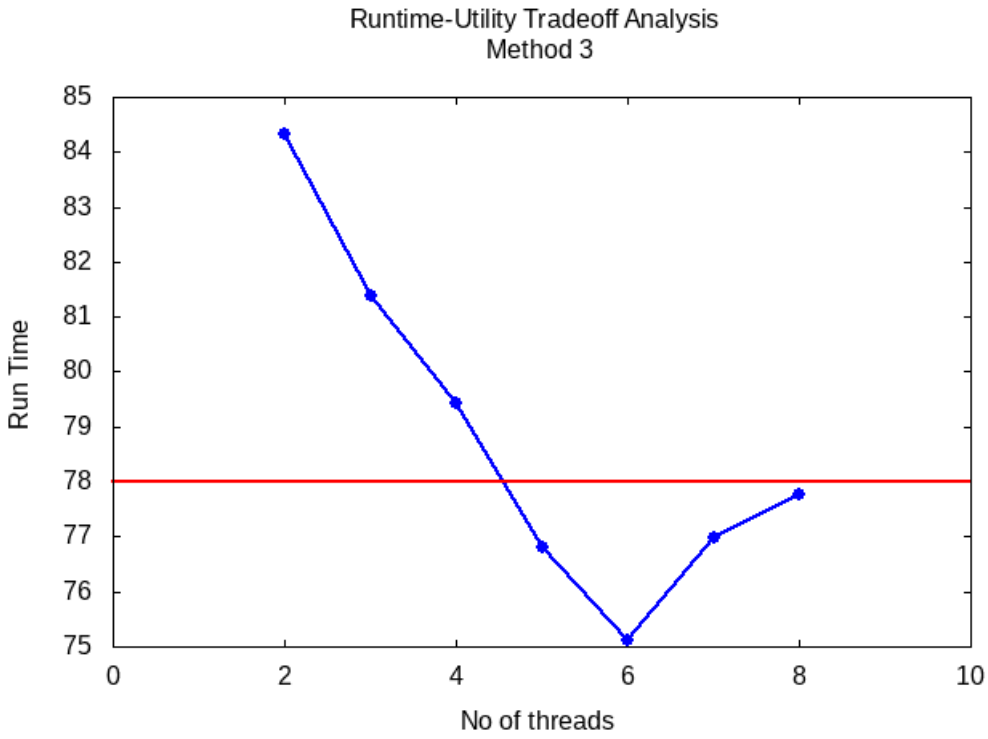

Fig. 3. Utility v/s Run time Analysis using Method 3.

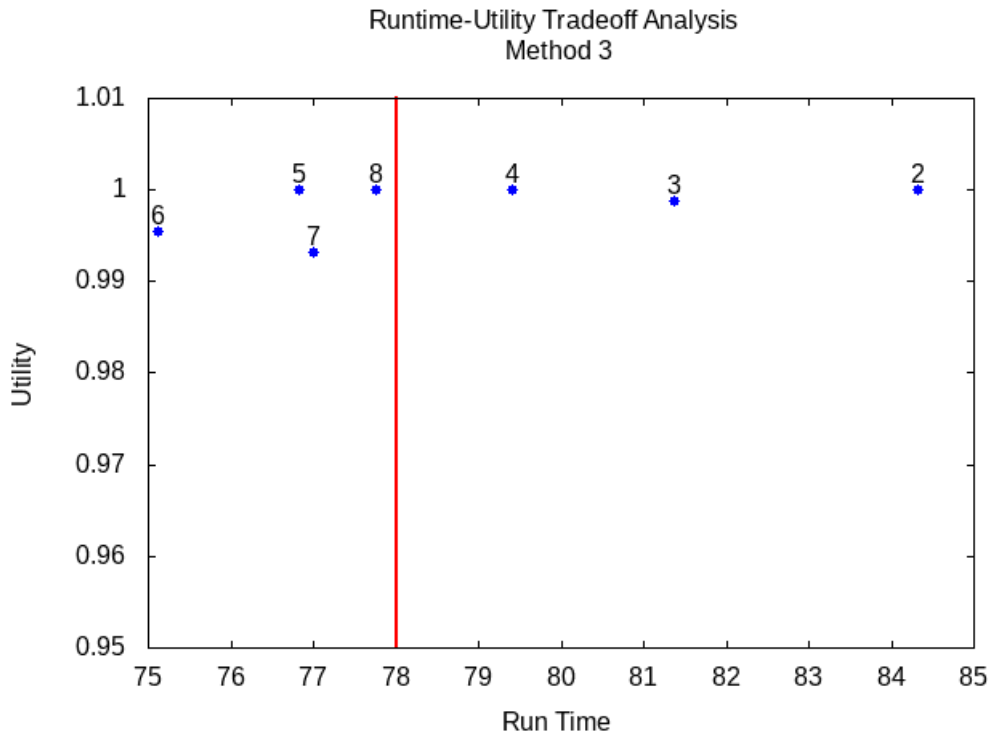

Fig. 4. Utility v/s Run time Analysis using Method 3.

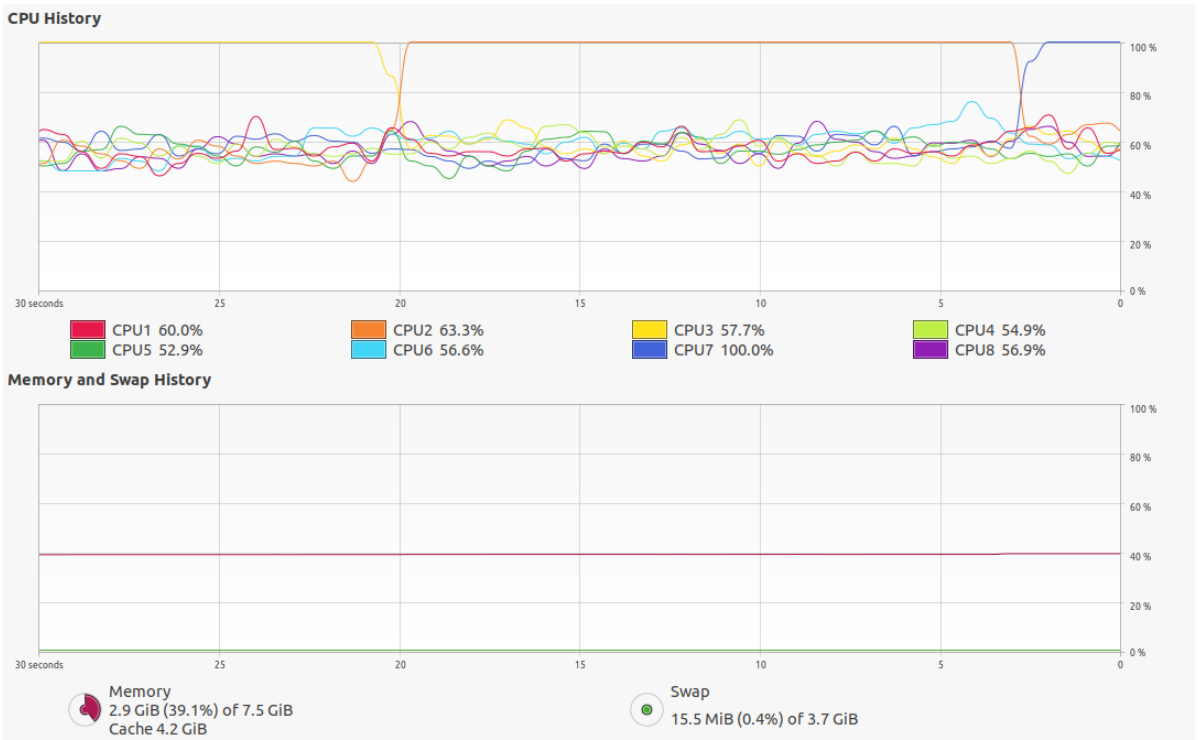

Fig. 5. CPU usage of baseline.

ibus-daemon	lenovo	0	1244	1.4 MiB	384.0 KiB	4.0 KiB	N/A	N/A	Normal
ibus-engine-simple	lenovo	0	1405	628.0 KiB	16.0 KiB	N/A	N/A	N/A	Normal
ibus-extension-gtk3	lenovo	0	1249	10.8 MiB	760.0 KiB	N/A	N/A	N/A	Normal
ibus-memconf	lenovo	0	1248	632.0 KiB	24.0 KiB	N/A	N/A	N/A	Normal
ibus-portal	lenovo	0	1255	656.0 KiB	92.0 KiB	N/A	N/A	N/A	Normal
ibus-x11	lenovo	0	1251	5.9 MiB	100.0 KiB	N/A	N/A	N/A	Normal
nautilus	lenovo	0	259157	32.2 MiB	N/A	148.0 KiB	N/A	N/A	Normal
output	lenovo	0	388335	330.0 MiB	N/A	64.0 KiB	N/A	21.3 KiB/s	Normal
pulseaudio	lenovo	0	910	4.3 MiB	3.9 MiB	12.0 KiB	N/A	N/A	Very High
(sd-pam)	lenovo	0	901	2.9 MiB	N/A	N/A	N/A	N/A	Normal
snap-store	lenovo	0	1502	292.4 MiB	35.9 MiB	10.1 MiB	N/A	N/A	Normal
ssh-agent	lenovo	0	1147	456.0 KiB	N/A	N/A	N/A	N/A	Normal
systemd	lenovo	0	900	1.7 MiB	1.2 GiB	2.4 GiB	N/A	134.7 KiB/s	Normal

Fig. 6. Memory usage of baseline.

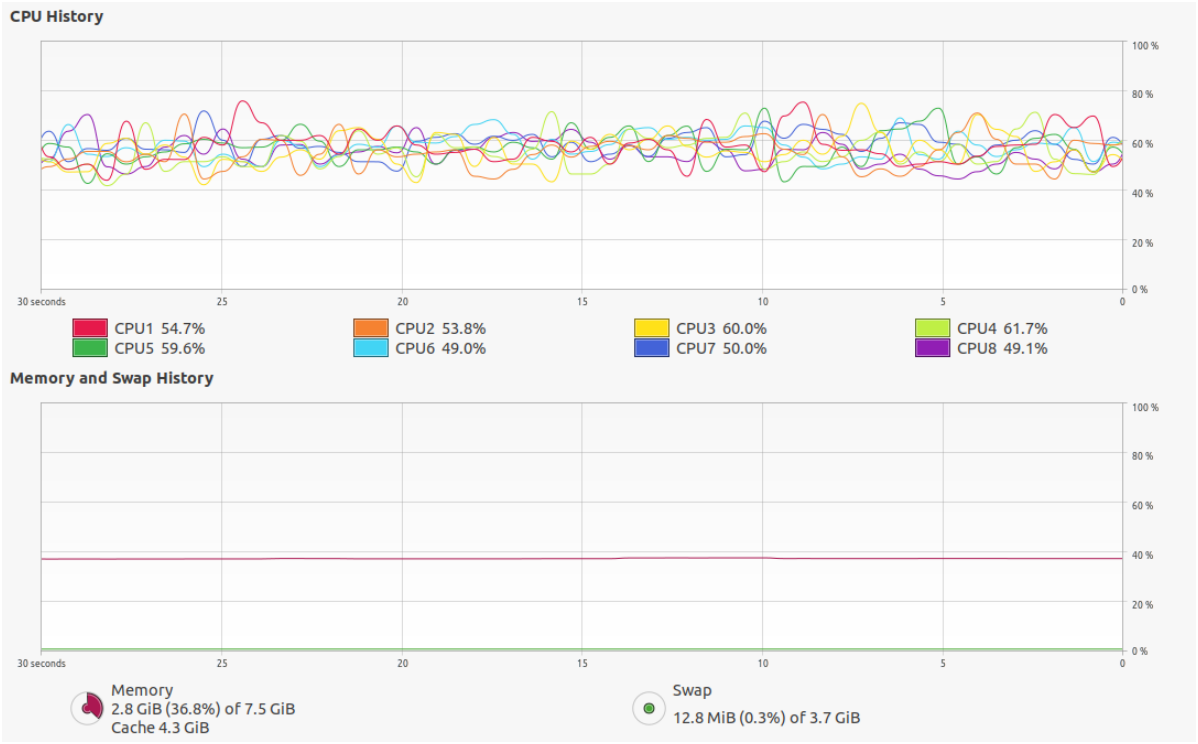

Fig. 7. CPU usage of method 3 with N = 2.

ibus-daemon	lenovo	0	1244	1.4 MiB	384.0 KiB	4.0 KiB	N/A	N/A	Normal
ibus-engine-simple	lenovo	0	1405	628.0 KiB	16.0 KiB	N/A	N/A	N/A	Normal
ibus-extension-gtk3	lenovo	0	1249	10.8 MiB	760.0 KiB	N/A	N/A	N/A	Normal
ibus-memconf	lenovo	0	1248	632.0 KiB	24.0 KiB	N/A	N/A	N/A	Normal
ibus-portal	lenovo	0	1255	656.0 KiB	92.0 KiB	N/A	N/A	N/A	Normal
ibus-x11	lenovo	0	1251	5.9 MiB	100.0 KiB	N/A	N/A	N/A	Normal
nautilus	lenovo	0	259157	25.0 MiB	N/A	148.0 KiB	N/A	N/A	Normal
output	lenovo	49	363650	142.2 MiB	N/A	N/A	N/A	N/A	Normal
pulseaudio	lenovo	0	910	4.7 MiB	3.9 MiB	12.0 KiB	N/A	N/A	Very High
rygel	lenovo	0	351810	11.1 MiB	48.0 KiB	6.3 MiB	N/A	N/A	Normal
(sd-pam)	lenovo	0	901	2.9 MiB	N/A	N/A	N/A	N/A	Normal
snap-store	lenovo	0	1502	292.4 MiB	35.9 MiB	10.1 MiB	N/A	N/A	Normal
ssh-agent	lenovo	0	1147	456.0 KiB	N/A	N/A	N/A	N/A	Normal
systemd	lenovo	0	900	1.7 MiB	1.2 GiB	2.3 GiB	N/A	N/A	Normal

Fig. 8. Memory usage of method 3 with N = 2.

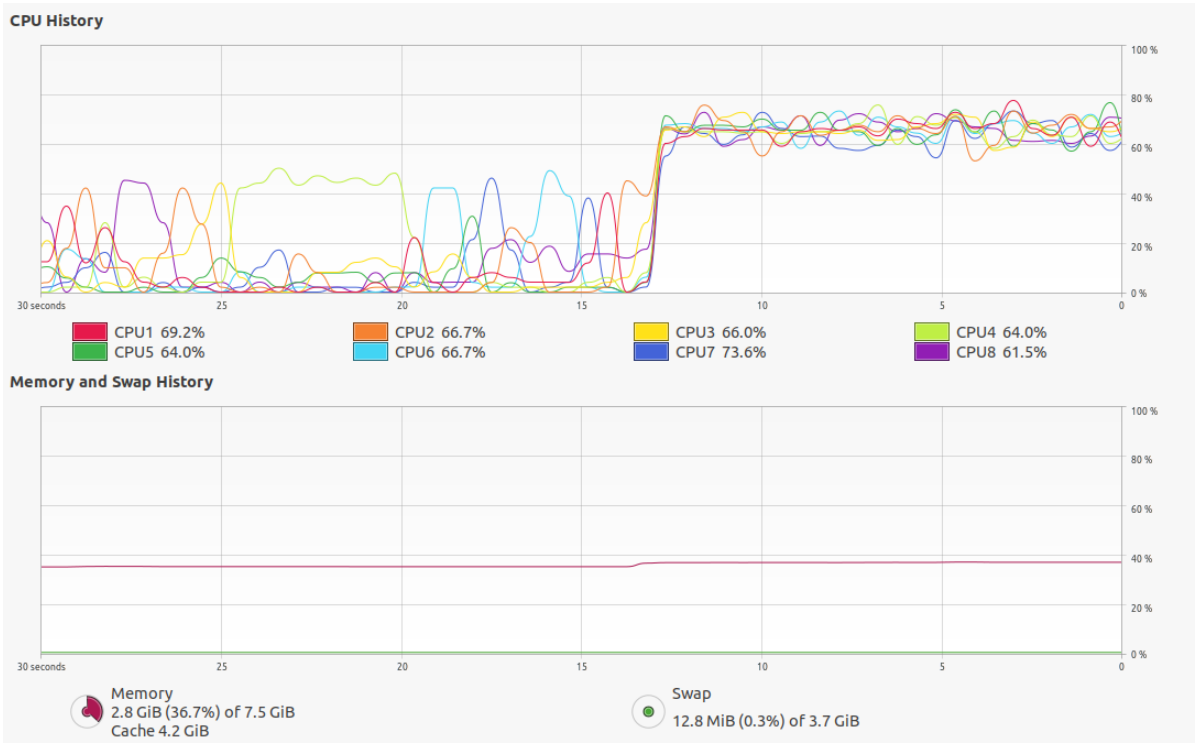

Fig. 9. CPU usage of method 3 with N = 6.

ibus-daemon	lenovo	0	1244	1.4 MiB	384.0 KiB	4.0 KiB	*N/A*	*N/A*	Normal
ibus-engine-simple	lenovo	0	1405	628.0 KiB	16.0 KiB	*N/A*	*N/A*	*N/A*	Normal
ibus-extension-gtk3	lenovo	0	1249	10.8 MiB	760.0 KiB	*N/A*	*N/A*	*N/A*	Normal
ibus-memconf	lenovo	0	1248	632.0 KiB	24.0 KiB	*N/A*	*N/A*	*N/A*	Normal
ibus-portal	lenovo	0	1255	656.0 KiB	92.0 KiB	*N/A*	*N/A*	*N/A*	Normal
ibus-x11	lenovo	0	1251	5.9 MiB	100.0 KiB	*N/A*	*N/A*	*N/A*	Normal
nautilus	lenovo	0	259157	25.0 MiB	*N/A*	148.0 KiB	*N/A*	*N/A*	Normal
output	lenovo	61	317014	140.8 MiB	*N/A*	*N/A*	*N/A*	*N/A*	Normal
pulseaudio	lenovo	0	910	4.7 MiB	3.9 MiB	12.0 KiB	*N/A*	*N/A*	Very High
(sd-pam)	lenovo	0	901	2.9 MiB	*N/A*	*N/A*	*N/A*	*N/A*	Normal
snap-store	lenovo	0	1502	292.4 MiB	35.9 MiB	10.1 MiB	*N/A*	*N/A*	Normal
ssh-agent	lenovo	0	1147	456.0 KiB	*N/A*	*N/A*	*N/A*	*N/A*	Normal
systemd	lenovo	0	900	1.7 MiB	1.2 GiB	2.3 GiB	*N/A*	*N/A*	Normal

Fig. 10. Memory usage of method 3 with N = 6.

Observations:

- The red line indicates the baseline performance. As we can see from the graph, performance gets better than baseline only when the no. of threads exceed 4. A possible reason is that, the extra overhead due to splitting the image into strips exceeds the performance gain by splitting the computation into different threads and this balance is displaced when the threads exceed 4.
- The performance only increase up to thread no. 6 and then starts decreasing again after that in the working machine. A possible reason for this is that as the threads are not all instantiated at the same time but in a for loop, the initial threads complete execution and then wait for the last threads to complete. So, the threads are not active 100% of the time and thus there is no more performance gain due to splitting into threads, but the splitting of the image still occurs which increases the overhead.
- This leads to a performance decrease. The utility of the output data doesn't differ much, and thus if power used isn't a limitation, using 6 threads would be the best choice for the current machine in this method.
- All the threads of the CPU seem to have equal load with none reaching 100% usage. This is because, for each frame, new threads are instantiated and these threads are not assigned to the same CPU thread each time and are being equally distributed with all threads of the CPU throughout the run.
- Memory usage remains constant irrespective of the no. of threads because increasing the no. of threads doesn't change the fact that only 1 frame of the video is being processed by the threads after splitting it. So, the data being stored by the program doesn't change.

4.4 Method 4

The results of the analysis after applying method 4 in which frames are processed across different threads are given below. N represents the number of threads used for computation. $N = 1$ is same as the baseline case.

	N=2	N=3	N=4	N=5	N=6	N=7	N=8
Run Time	82.34	73.52	69.29	66.56	64.97	63.78	63.12
Utility	0.999	0.999	0.999	0.999	0.999	0.999	0.995

Table 4. Utility v/s Run time Analysis using Method 4.

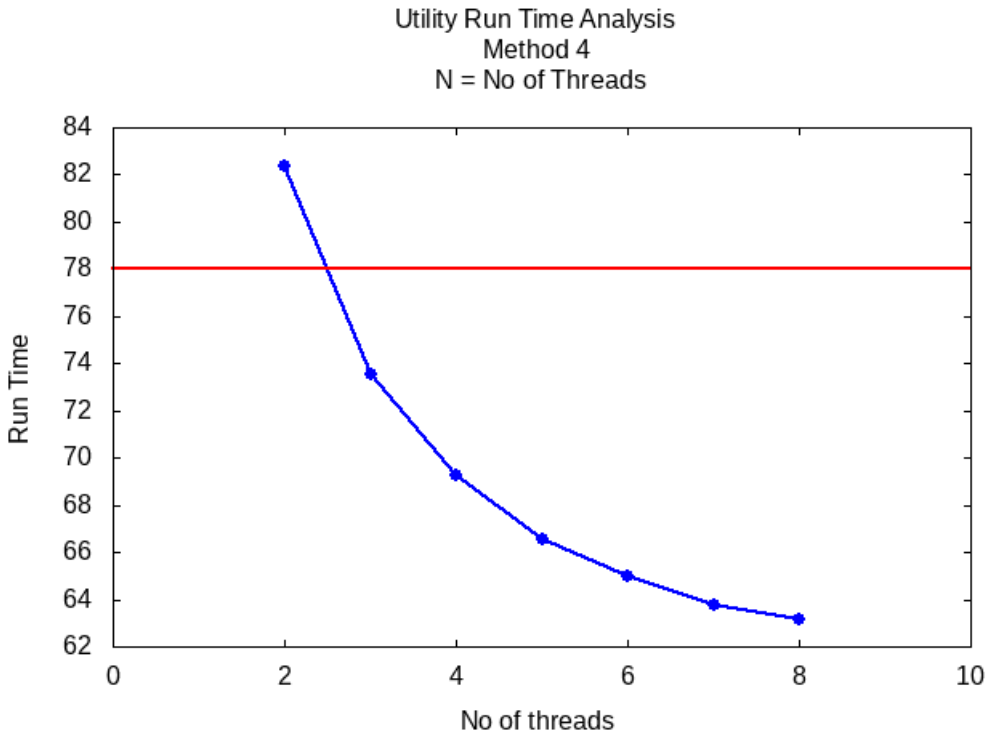

Fig. 11. Utility v/s Run time Analysis using Method 4.

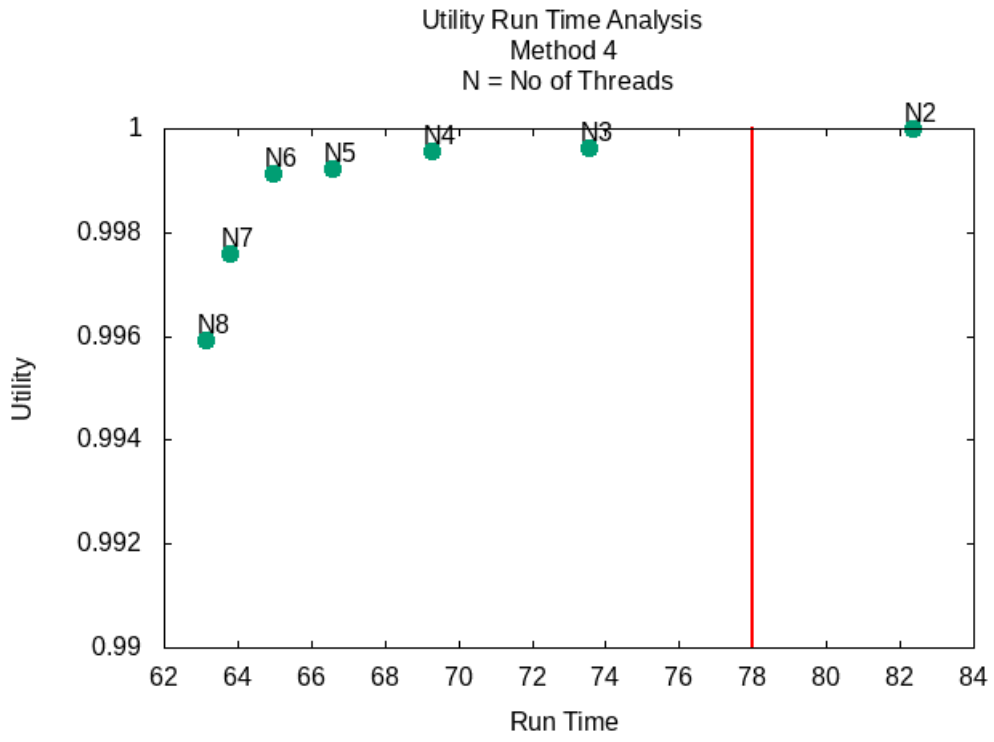

Fig. 12. Utility v/s Run time Analysis using Method 4.

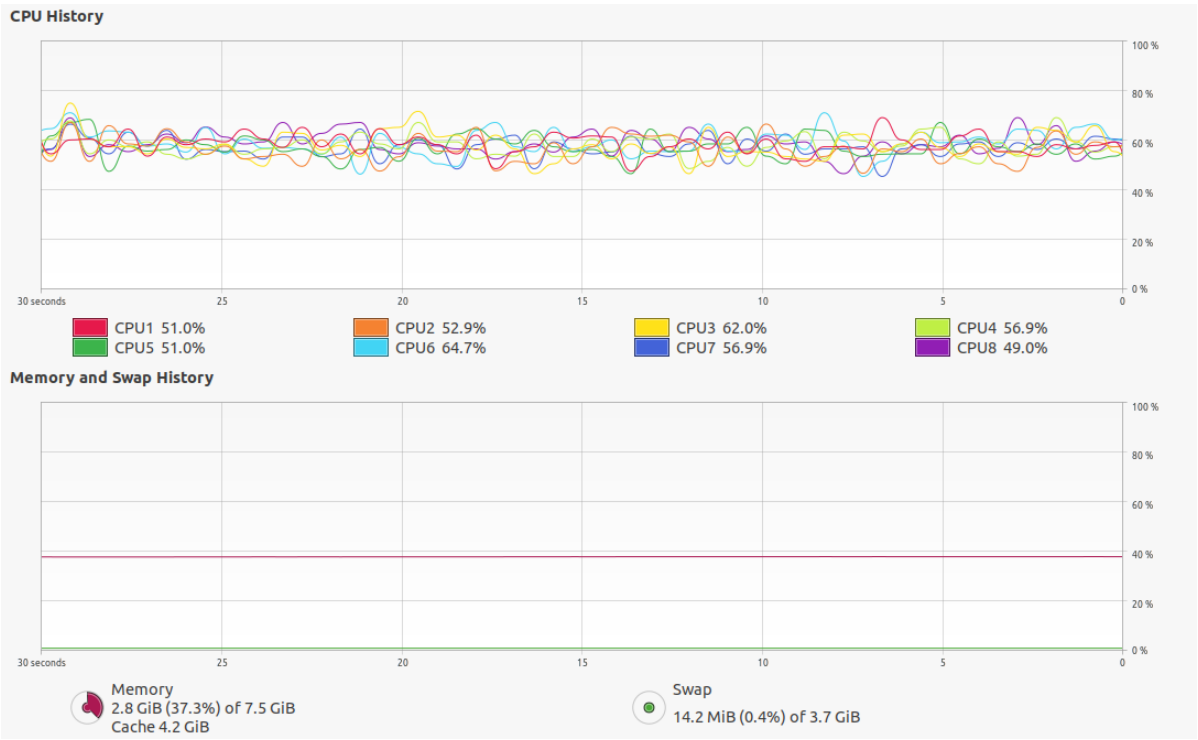

Fig. 13. CPU usage of method 4 with N = 2.

ibus-daemon	lenovo	0	1244	1.4 MiB	384.0 KiB	4.0 KiB	N/A	N/A	Normal
ibus-engine-simple	lenovo	0	1405	628.0 KiB	16.0 KiB	N/A	N/A	N/A	Normal
ibus-extension-gtk3	lenovo	0	1249	10.8 MiB	760.0 KiB	N/A	N/A	N/A	Normal
ibus-memconf	lenovo	0	1248	632.0 KiB	24.0 KiB	N/A	N/A	N/A	Normal
ibus-portal	lenovo	0	1255	656.0 KiB	92.0 KiB	N/A	N/A	N/A	Normal
ibus-x11	lenovo	0	1251	5.9 MiB	100.0 KiB	N/A	N/A	N/A	Normal
nautilus	lenovo	0	259157	28.7 MiB	N/A	148.0 KiB	N/A	N/A	Normal
output	lenovo	50	375899	170.8 MiB	N/A	N/A	N/A	N/A	Normal
pulseaudio	lenovo	0	910	4.5 MiB	3.9 MiB	12.0 KiB	N/A	N/A	Very High
rygel	lenovo	0	351810	11.1 MiB	64.0 KiB	9.1 MiB	N/A	N/A	Normal
(sd-pam)	lenovo	0	901	2.9 MiB	N/A	N/A	N/A	N/A	Normal
snap-store	lenovo	0	1502	292.4 MiB	35.9 MiB	10.1 MiB	N/A	N/A	Normal
ssh-agent	lenovo	0	1147	456.0 KiB	N/A	N/A	N/A	N/A	Normal
systemd	lenovo	0	900	1.7 MiB	1.2 GiB	2.4 GiB	N/A	N/A	Normal

Fig. 14. Memory usage of method 4 with N = 2.

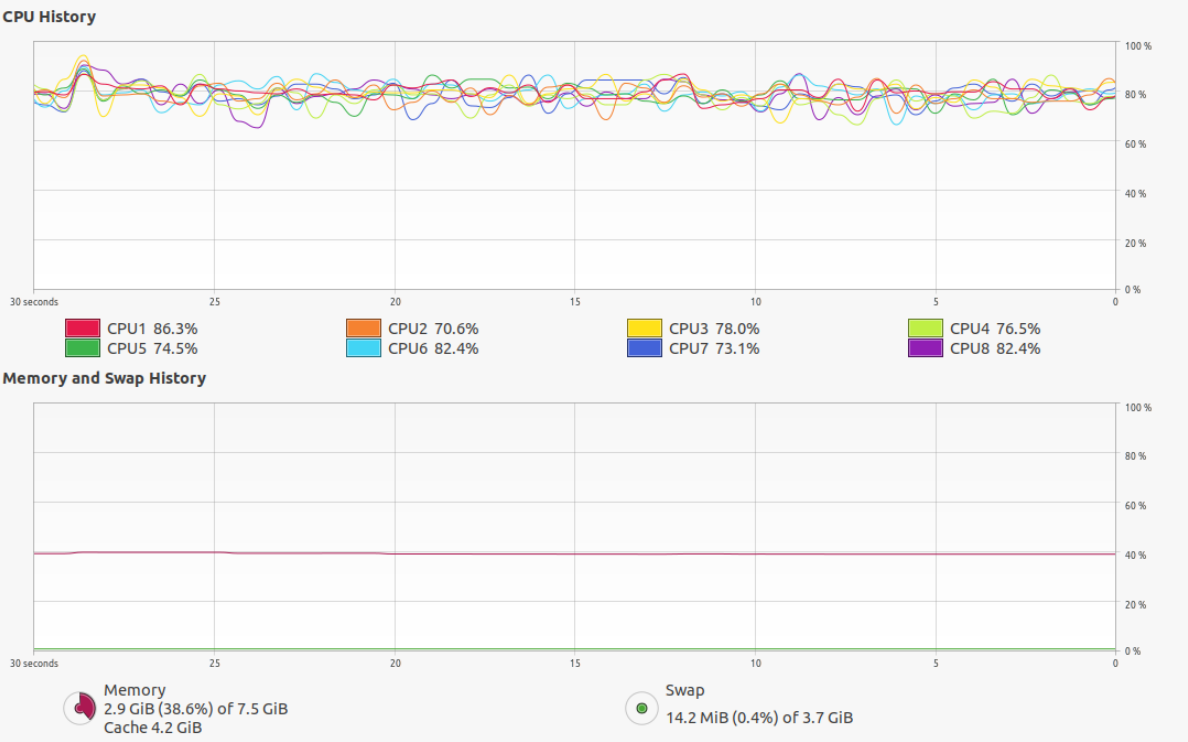

Fig. 15. CPU usage of method 4 with N = 6.

ibus-daemon	lenovo	0	1244	1.4 MiB	384.0 KiB	4.0 KiB	N/A	N/A	Normal
ibus-engine-simple	lenovo	0	1405	628.0 KiB	16.0 KiB	N/A	N/A	N/A	Normal
ibus-extension-gtk3	lenovo	0	1249	10.8 MiB	760.0 KiB	N/A	N/A	N/A	Normal
ibus-memconf	lenovo	0	1248	632.0 KiB	24.0 KiB	N/A	N/A	N/A	Normal
ibus-portal	lenovo	0	1255	656.0 KiB	92.0 KiB	N/A	N/A	N/A	Normal
ibus-x11	lenovo	0	1251	5.9 MiB	100.0 KiB	N/A	N/A	N/A	Normal
nautilus	lenovo	0	259157	28.7 MiB	N/A	148.0 KiB	N/A	N/A	Normal
output	lenovo	74	381715	288.0 MiB	N/A	N/A	N/A	N/A	Normal
pulseaudio	lenovo	0	910	4.5 MiB	3.9 MiB	12.0 KiB	N/A	N/A	Very High
(sd-pam)	lenovo	0	901	2.9 MiB	N/A	N/A	N/A	N/A	Normal
snap-store	lenovo	0	1502	292.4 MiB	35.9 MiB	10.1 MiB	N/A	N/A	Normal
ssh-agent	lenovo	0	1147	456.0 KiB	N/A	N/A	N/A	N/A	Normal
systemd	lenovo	0	900	1.7 MiB	1.2 GiB	2.4 GiB	N/A	N/A	Normal

Fig. 16. Memory usage of method 4 with N = 6.

Observations:

- The red line indicates the baseline performance. As we can see from the given 2 graphs, run-time is lower than baseline when the no. of threads exceed 2. This maybe because, the extra overhead due to implementing multiple threads exceeds the performance benefit gained by multi-threading when using 2 threads and is only compensated when the no. of threads exceed 2. And as we keep increasing the no. of threads, the run-time keeps decreasing but each step, the decrease in run-time keeps reducing, and it seems to approach a saturation point.

- A possible reason for this is that as all the threads are being instantiated sequentially in a 'for' loop, as the no. of threads keep increasing, the time difference between the initial threads being initiated and the final threads being instantiated also increases and thus the initial threads finish computing and then wait for the last threads to finish before proceeding further. Due to this, increasing the threads will just lead to the threads waiting for the other threads to complete.

- The error also increases as each thread processes fewer frames and thus leads to a difference in queue density calculated. As we can see from the graph, after thread no. 6, error increases sharply without much difference in run-time and thus using 6 threads would be the best choice on the current machine.

- All the threads of the CPU seem to have equal load with none reaching 100% usage. This is because, for every N frames, new N threads are instantiated in a 'for' loop and these threads are not assigned to the same CPU thread each time and are being equally distributed with all threads of the CPU throughout the run.

- Memory usage increases with an increase in no. of threads because more no. of threads signify more frames of the video being processed concurrently. This in turn leads to more data being stored in the memory to compute.

5 Conclusion

We have analyzed the four methods that have been implemented to reduce run-time while maintaining good utility to see what kind of trade-off is shown when adjusting the parameters of each method.

- As we can see from the results, Method 4, Method 3 and method 2 have the best utility when compared to other methods as the utility stays above 0.99 with almost all the parameters we have tested which means the error is really negligible and accuracy is really good. We wouldn't have to worry about the output being off at all while implementing these methods unless until it is a very sensitive operation which requires very good accuracy and precision. In this task, we are only trying to find pattern of traffic density in different times of the day which shouldn't require that level of accuracy.

- Method 1 reduces run-time the most, and it should be implemented if that is the most important factor. Out of the other more accurate methods, method 2 has the best reduction in run-time and should be preferred if this balance is what you prefer.

- Of the two multi-threading methods, method 4 is the better implementation as it reduces run-time substantially whereas method 3 doesn't.

- We can also implement a combination of multi-threading of methods 3 or 4 with methods 1 or 2 to reduce run-time even further while having good utility. As method 4 is a better implementation of multi-threading, we have 2 good choices of combination, 'method 1 + method 4' and 'method 2 + method 4'. For best reduction in run-time the initial combination would be better, whereas for having better accuracy but a slightly higher run-time, the latter combination would be a better choice.