Report

Aarunish Sinha, Mohit Sharma

March 2021

1 Introduction

We have implemented the 4 methods mentioned in the subtask specifications. In all the methods we have calculated the queue density and dynamic density using the `absdiff` function.
We have used `Matplotlib` for as our plotting tool for all the methods.

Baseline: Calculating the queue and dynamic density for every frame in the video without multi-threading.
Note: Compiler optimisation were set to default in all the methods.

2 Analysis

2.1 Method-1

We observed that the queue density values do not change significantly if we start skipping frames, but the dynamic densities for each frame vary significantly as we skip more number of frames.
Here is a plot showing the variation in dynamic densities for each frame for $x = 1, 6, 15$, i.e., 0 frames skipped, 5 frames skipped and 14 frames skipped.

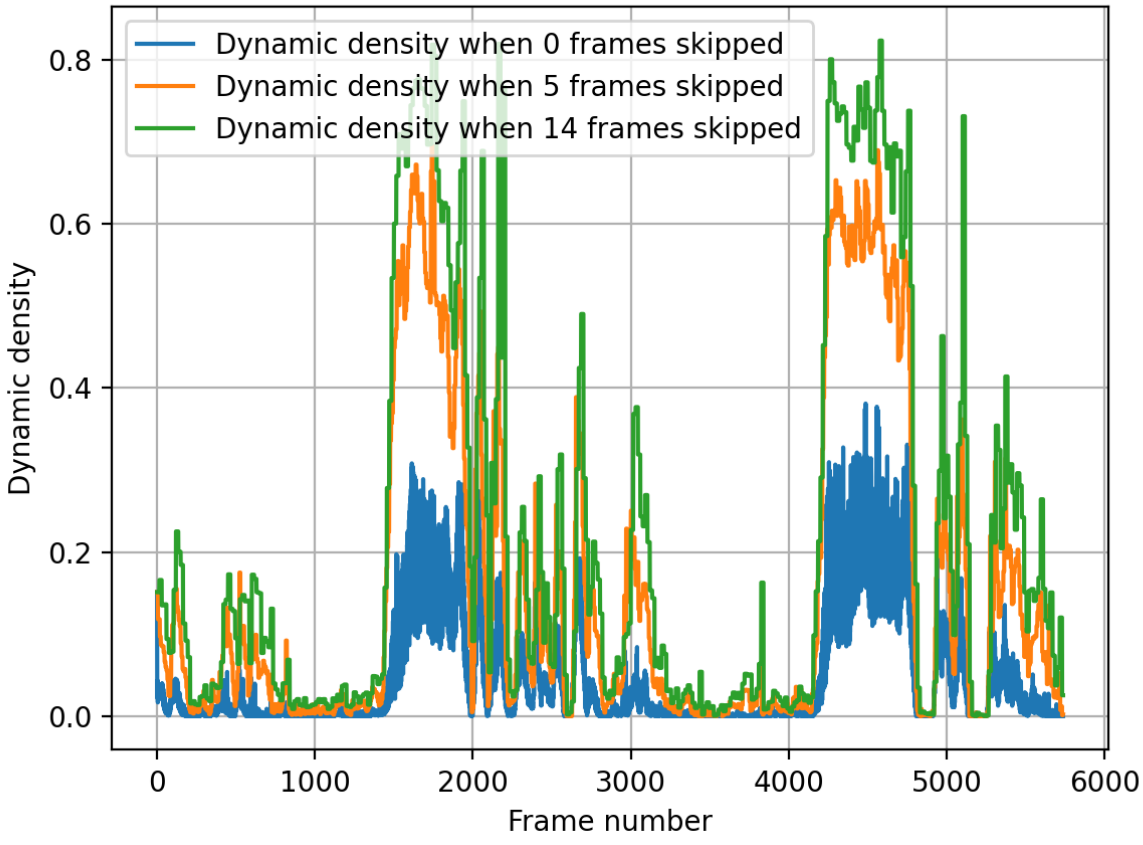
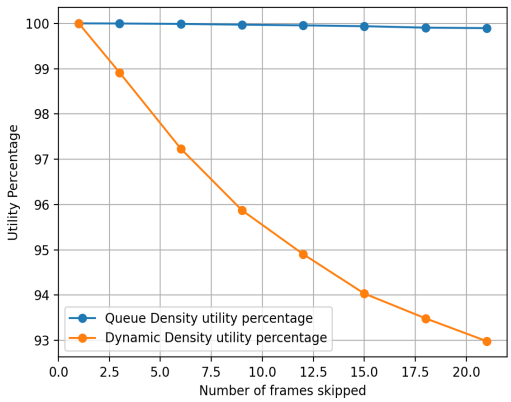


Figure 1: Dynamic Density vs Frame Number

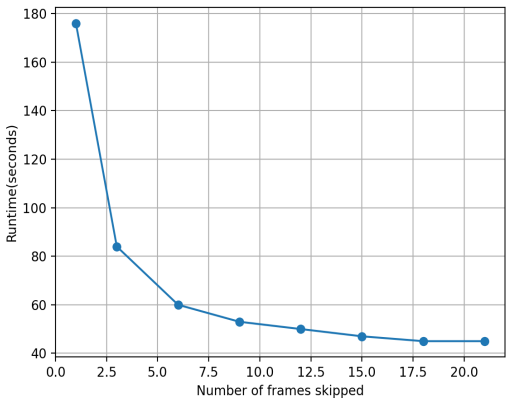
Now, moving on to utility and runtime for different values of x.
We have used squared error as a metric for measuring utility.

$$Squared\ Error\ =\ (Baseline\ Density - New\ Density)^2 \tag{1}$$

$$Utility\ =\ \frac{100}{1 + Squared\ Error} \tag{2}$$



(a) Utility Percentage vs Number of frames skipped



(b) Runtime vs Number of frames skipped

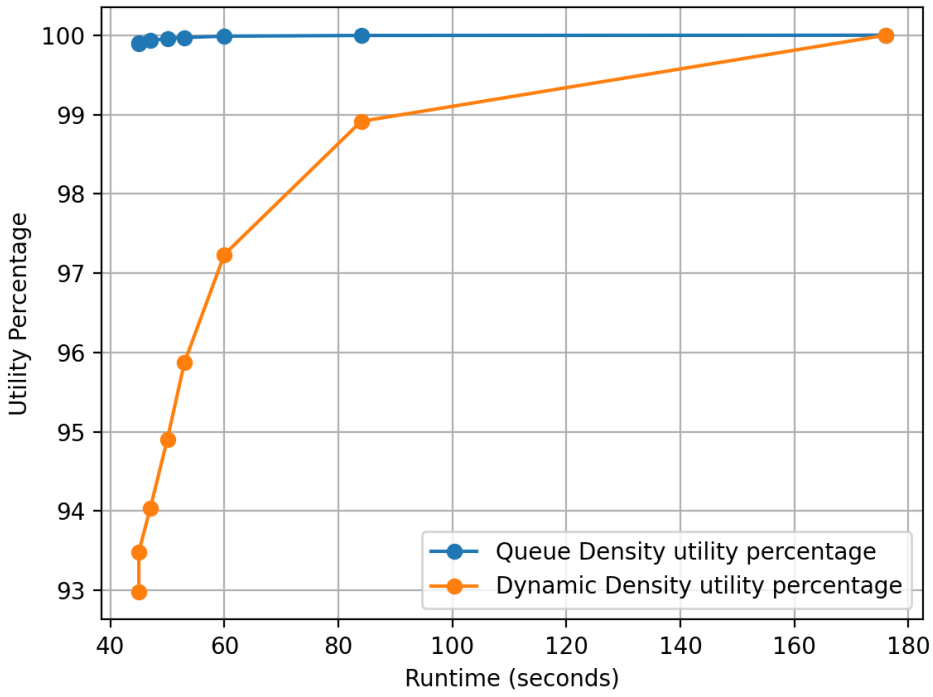


Figure 3: Utility Percentage vs Runtime

2.1.1 Output Values

Frames skipped	Queue Density Error	Dynamic Density Error
1	0.0	0.0
3	2.840917864604146e-05	0.01097480035927651
6	0.0001198180115234778	0.028442184556385353
9	0.0002880365423124875	0.04305052704767744
12	0.0004383934888585457	0.053701053821146066
15	0.0006292749930353825	0.06346216178689988
18	0.0009452962914193953	0.0696859631139873
21	0.0010530033336218205	0.07548481140880152

Number of frames skipped	Runtime(seconds)
1	176
3	84
6	60
9	53
12	50
15	47
18	45
21	45

Number of frames skipped	Queue Density Utility(%)	Dynamic Density Utility(%)
1	100.0	100.0
3	99.99715916284126	98.91443383599905
6	99.98801963431126	97.23444010917797
9	99.9712046398847	95.8726326355895
12	99.95617986157751	94.90357785764716
15	99.93711207449535	94.03249461360558
18	99.90555964497543	93.48538117569358
21	99.894810431604	92.98132241310574

Runtime(sec)	Queue Density Utility(%)	Dynamic Density Utility(%)
176	100.0	100.0
84	99.99715916284126	98.91443383599905
60	99.98801963431126	97.23444010917797
53	99.9712046398847	95.8726326355895
50	99.95617986157751	94.90357785764716
47	99.93711207449535	94.03249461360558
45	99.90555964497543	93.48538117569358
45	99.894810431604	92.98132241310574

Processing every 3^{rd} frame reduced the runtime of the program by more than half of that in *baseline*. As we skipped more and more frames the runtime decreased even further but the rate of decrement slowed down. We do not see a linear decrease in the runtime because the we are reading every frame and there is some 'overhead' that would always be there when running the program.

From analysis of the above method, we can conclude that processing every 3^{rd} frame of the video is a reasonable trade-off since the utility reduces only by 1% while calculating dynamic density and by a very negligible amount while calculating queue density.

`Note`: Runtimes are for Intel Core i5-6200U processor (2.30 GHz x 4)

2.2 Method-2

Reducing the resolution of the video affects the utility more in case of dynamic density. As decrease the resolution of the video the runtime decreases along with the utility.
Below are the graphs for *Utility vs Runtime* for queue density and dynamic density,

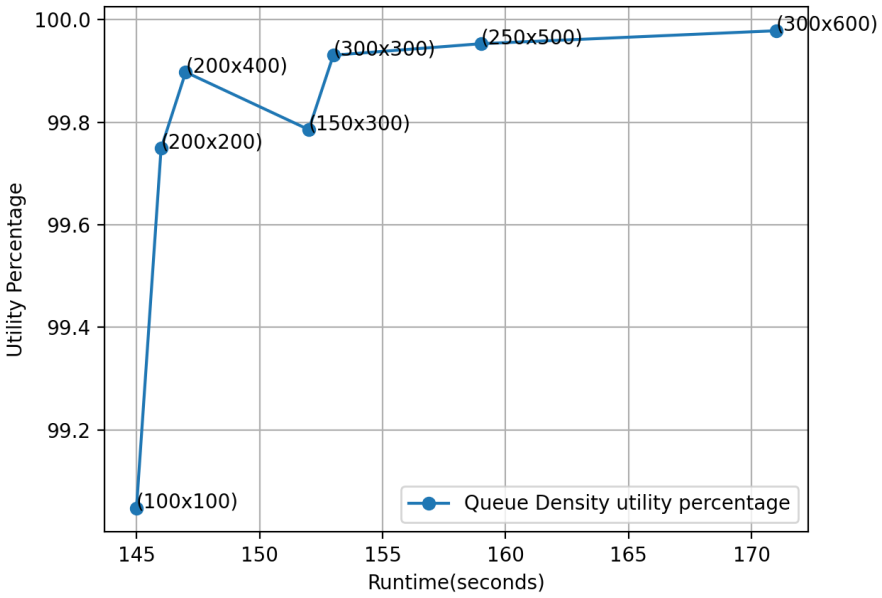


Figure 4: Utility Percentage vs Runtime for Queue Density

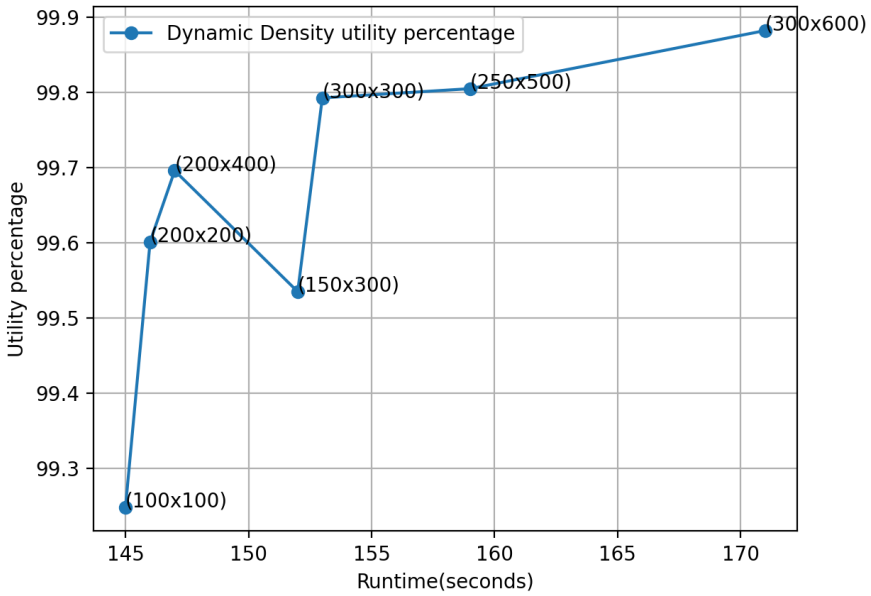


Figure 5: Utility Percentage vs Runtime for Dynamic Density

2.2.1 Output Values

Resolution	Runtime(sec)	Queue Density Error	Dynamic Density Error
100 x 100	145	0.00961153214858116	0.007568749586411405
200 x 200	146	0.002516787873053002	0.004001393213587502
200 x 400	147	0.0010262353425777076	0.003050576624151185
150 x 300	152	0.0021499448190919358	0.004670403896163429
300 x 300	153	0.0006966433013263569	0.002078881319988709
250 x 500	159	0.0004717984044166022	0.001955761004711435
300 x 600	171	0.00021855524642998015	0.0011815226556515677

Resolution	Runtime(sec)	Queue Density Utility(%)	Dynamic Density Utility(%)
100 x 100	145	99.04799699265254	99.24881060577572
200 x 200	146	99.74895304462756	99.60145541224998
200 x 400	147	99.89748167367198	99.69587010912072
150 x 300	152	99.78546675273428	99.535130737598
300 x 300	153	99.93038416727089	99.79254314617923
250 x 500	159	99.95284240843479	99.80480565302102
300 x 600	171	99.97814925095284	99.88198716926802

From our observations, reducing the resolution to as low as 100×100 does not reduce the runtimes as much as $Method - 1$ but it keeps the utility higher at the same time.

`Note`: Runtimes are for Intel Core i5-6200U processor (2.30 GHz x 4)

2.3 Method-3

In this method we first read, split and store all the frames in a `std::vector<Mat>`. Each split of a particular frame is processed by one thread and all threads run in parallel.
We have plotted the *CPU utilisation vs Runtime* graph,

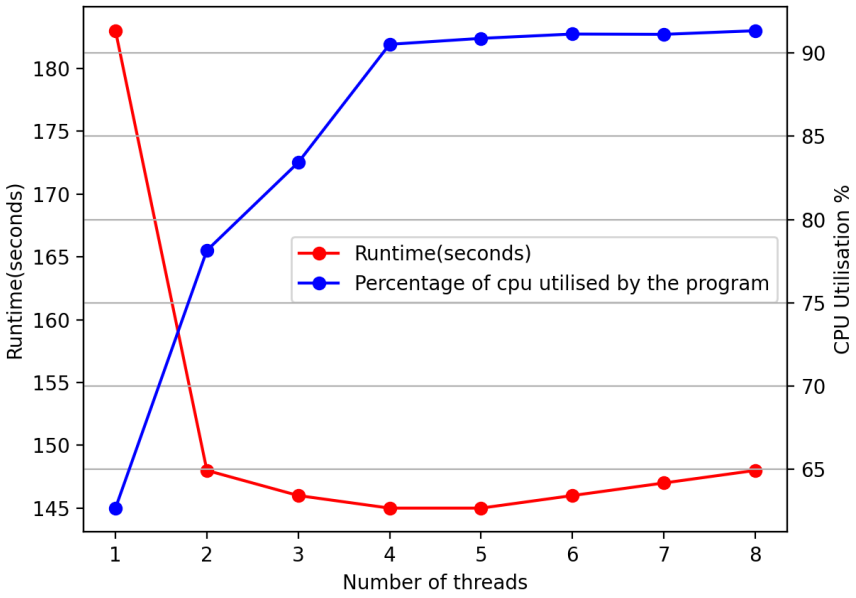
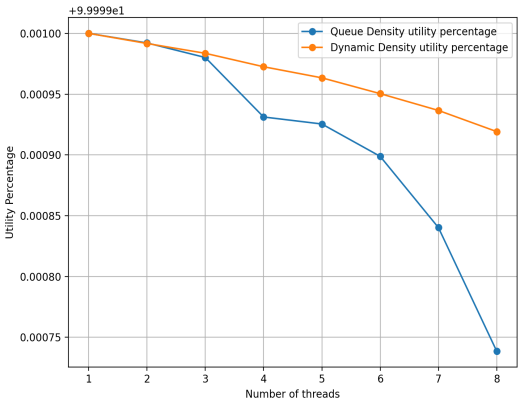
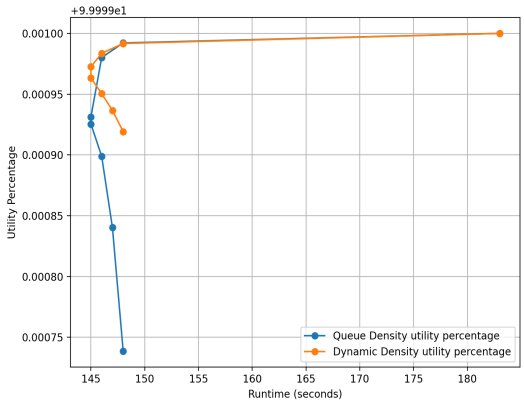


Figure 6: CPU Utilisation Percentage vs Number of threads and Runtime vs Number of threads



(a) Utility Percentage vs Number of threads



(b) Utility Percentage vs Runtime

2.3.1 Output Values

Number of threads	Runtime(s)	CPU Utilisation(%)	Queue Density Error	Dynamic Density Error
1	183	62.66046	0.0	0.0
2	148	78.14396	7.891518426703894e-08	8.316427778142135e-08
3	146	83.43075	1.9849279757713988e-07	1.6407284004607154e-07
4	145	90.52962	6.877489182596548e-07	2.7445002063008387e-07
5	145	90.88717	7.45462100739067e-07	3.657536993154096e-07
6	146	91.14683	1.0114067525400607e-06	4.957661889317304e-07
7	147	91.12881	1.5975822513486348e-06	6.339827503833789e-07
8	148	91.3468	2.6140666828796926e-06	8.078360598208544e-07

Number of threads	Runtime(sec)	Queue Density Utility(%)	Dynamic Density Utility(%)
1	183	100.0	100.0
2	148	99.9999921084822	99.99999168357293
3	146	99.99998015072418	99.99998359271869
4	145	99.99993122515548	99.99997255500546
5	145	99.9999254538455	99.99996342464344
6	146	99.99989885942703	99.99995042340568
7	147	99.99984024203009	99.99993660176516
8	148	99.99973859401504	99.99991921645928

From the above graphs we can see that the utility of for different number of threads is very close to a 100% and as we increase the number of threads the runtime decreases and the CPU utilisation increases.

`Note`: Runtimes are for Intel Core i5-6200U processor (2.30 GHz x 4)

2.4 Method-4

In this method we first read the whole video and store each frame in a `std::vector`. Each thread has it's own iterator with the initial value of `thread_num` and increases by `NUM_THREADS` for reading from the vector and processing the frames.

Since, the method executes in the exact same manner as the baseline because for n threads each thread processes the n^{th} frame starting from $thread\_num$ index in the vector and also uses the $(iter-1)^{th}$ frame for calculating the dynamic density.

Hence, we have only plotted the *CPU utilisation vs Runtime* graph,

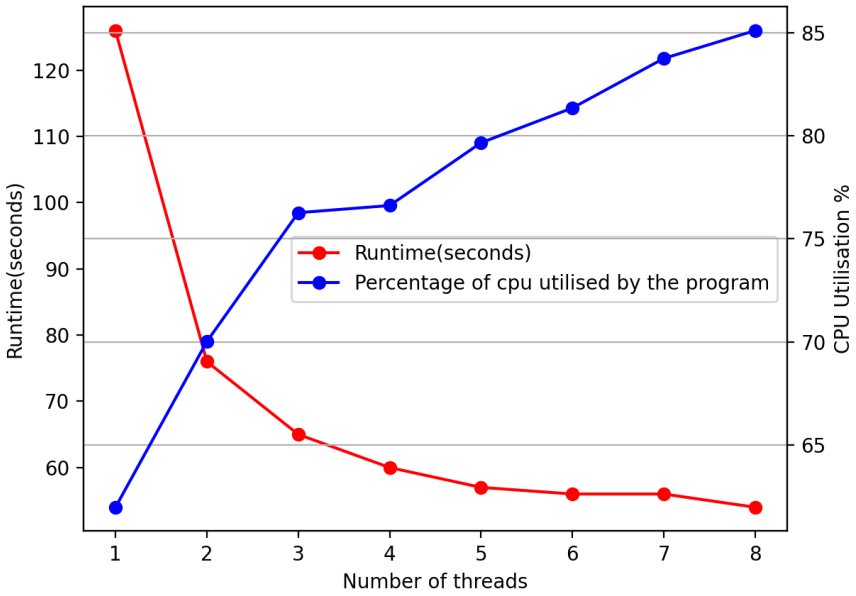


Figure 8: CPU Utilisation Percentage vs Number of threads and Runtime vs Number of threads

2.4.1 Output Values

Number of threads	Runtime(s)	CPU Utilisation(%)
1	126	61.99003859974452
2	76	70.02332798100772
3	65	76.26901814414292
4	60	76.6154087176912
5	57	79.65829618268536
6	56	81.34612523567118
7	56	83.74624496486628
8	54	85.09740021941407

In this method we do not have to do any trade-off with utility since it always remains 100%. More is the number of threads lesser will be the runtime.
Some of the issues that can be found in this method are that since using more number of threads leads to higher CPU utilisation the processor will heat up more unless it has a proper cooling mechanism in place. We are reading the whole video before we start processing and therefore there will always be an overhead in runtime for this pre-processing step.

`Note`: Runtimes are for Intel Core i9-9980H processor (2.30 GHz x 8, Turbo Boost up to 4.90GHz)

3 Some Extra Methods

3.1 Method-5

This method is basically just another implementation of $Method-3$. In this method we do not perform the pre-processing step where we read, split and store all the frames before density estimation.

Instead, each thread individually reads each frame, splits it and processes the i^{th} split (where i it the thread number). We have plotted the *CPU utilisation vs Runtime* graph,

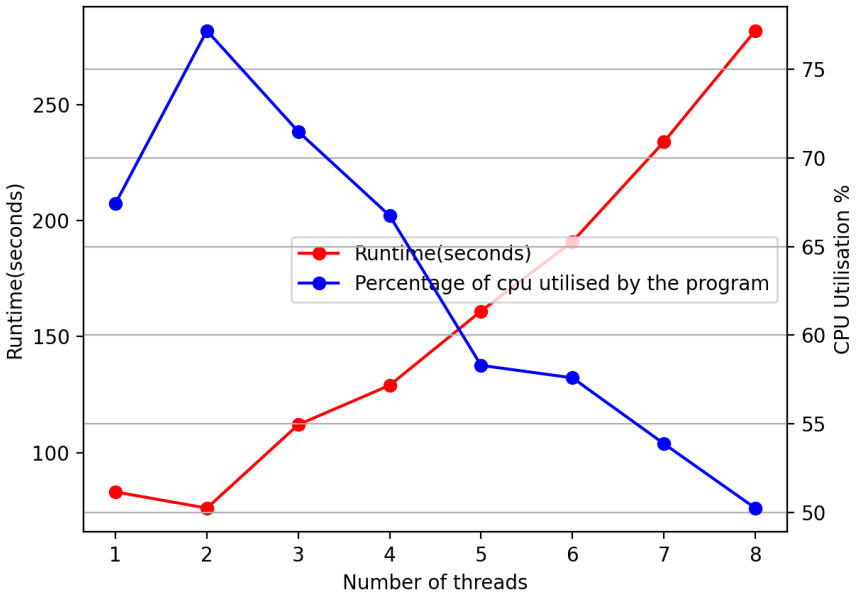
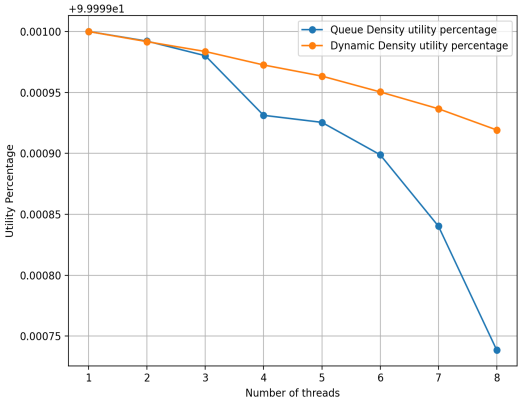
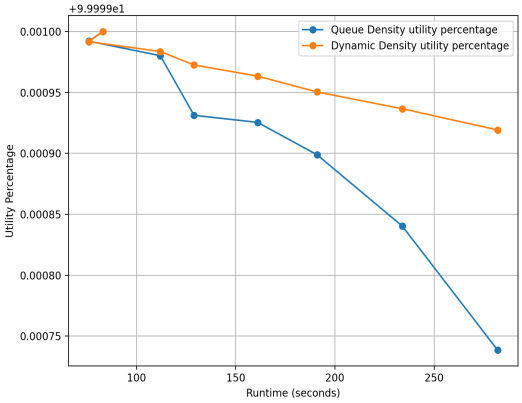


Figure 9: CPU Utilisation Percentage vs Number of threads and Runtime vs Number of threads



(a) Utility Percentage vs Number of threads



(b) Utility Percentage vs Runtime

3.1.1 Output Values

Number of threads	Runtime(s)	CPU Utilisation(%)	Queue Density Error	Dynamic Density Error
1	83	67.42935541310541	0.0	0.0
2	76	77.17454923594018	7.891720101621075e-08	8.31623816182226e-08
3	112	71.48594872629461	1.985101564092747e-07	1.640737282812542e-07
4	129	66.74038597677905	6.877588195607497e-07	2.744401062835002e-07
5	161	58.294503807166855	7.454863076921711e-07	3.657608413560277e-07
6	191	57.591970255691216	1.011375080920341e-06	4.957595759748897e-07
7	234	53.87883158618619	1.597583095063611e-06	6.339740273091669e-07
8	282	50.24152623384642	2.6140911725640653e-06	8.078352636383261e-07

Number of threads	Runtime(sec)	Queue Density Utility(%)	Dynamic Density Utility(%)
1	83	100.0	100.0
2	76	99.99999210828052	99.99999168376253
3	112	99.99998014898831	99.99998359262986
4	129	99.99993122416535	99.9999725559969
5	161	99.9999254514248	99.99996342392924
6	191	99.99989886259418	99.99995042406697
7	234	99.9998402419457	99.99993660263746
8	282	99.99973859156609	99.9999192165389

In this method every thread is splitting the same frame and then calculating the densities on each of the splits. Due to this the runtime increases as we increase the number of threads. The utility is still on par with what we obtained in $Method - 3$.
We first came up with this method as our method-3 but since it was taking too much time and had poor CPU utilisation we came up with our current method-3 where we decided to make the splitting of frames a pre-processing step.

`Note`: Runtimes are for Intel Core i9-9980H processor (2.30 GHz x 8, Turbo Boost up to 4.90GHz)

3.2 Method-6

In this method instead of processing consecutive frames by different threads, we divide the whole video into chunks and then assign each thread to one chunk for density estimation.
We have plotted the *CPU utilisation vs Runtime* graph,

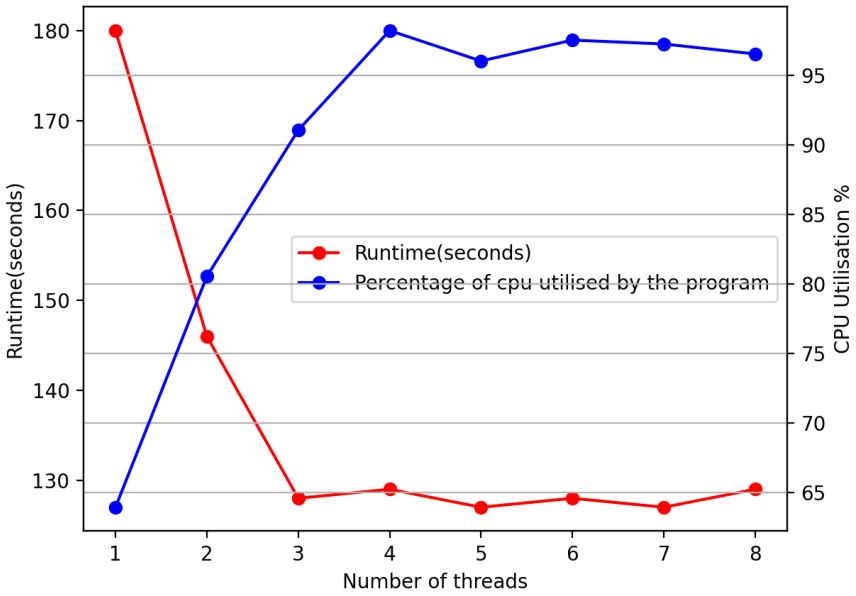
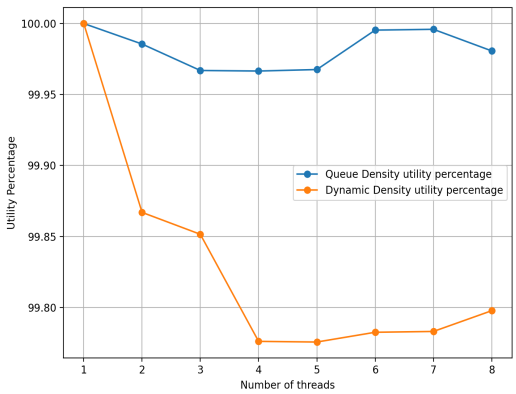
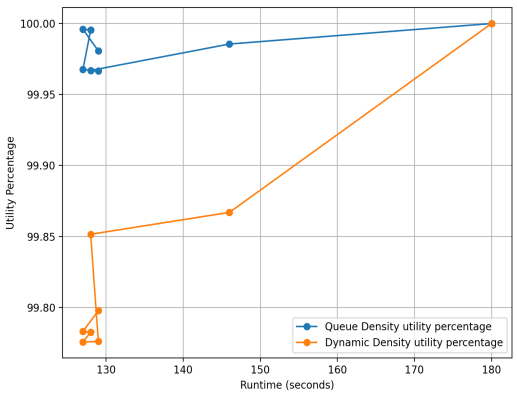


Figure 11: CPU Utilisation Percentage vs Number of threads and Runtime vs Number of threads



(a) Utility Percentage vs Number of threads



(b) Utility Percentage vs Runtime

3.2.1 Output Values

Number of threads	Runtime(s)	CPU Utilisation(%)	Queue Density Error	Dynamic Density Error
1	180	63.9224	0.0	0.0
2	146	80.5473	0.0001444928031390518	0.0013323414650222088
3	128	91.0714	0.00033090657945098134	0.0014870296765722199
4	129	98.2463	0.0003348274034232664	0.0022442835060664623
5	127	96.055	0.0003243256577953945	0.002248930979182977
6	128	97.5589	4.6426128546892077e-05	0.0021799171578281056
7	127	97.2803	4.073270053237911e-05	0.002174127747226057
8	129	96.5612	0.00019285136818097967	0.002027331932943954

Number of threads	Runtime(sec)	Queue Density Utility(%)	Dynamic Density Utility(%)
1	180	100.0	100.0
2	146	99.98555280720147	99.86694313068199
3	128	99.96692028834913	99.8515178297364
4	129	99.96652846684422	99.77607420236755
5	127	99.9675779495233	99.77561153624922
6	128	99.99535760267385	99.78248245444688
7	127	99.9959268958553	99.78305888297943
8	129	99.98071858162982	99.7976769826195

In this method we see that splitting the video into chunks reduces the runtime significantly by just using 3 threads. It also managed to keep the utility very high. The utility here is not 100% like method-4 because we used `cv::Video.set()` which did not split the video in precise chunks in terms of frame numbers. `Note`: Runtimes are for Intel Core i5-6200U processor (2.30 GHz x 4)