Traffic Density Estimation

Rakshita Choudhary, Ramneet Singh

March 2021

1 Metrics

Utility is defined as the inverse of overall error. Error is calculated by summing up the squares of differences in queue densities of corresponding frames from the baseline output and current output. More formally, if we have n frames, with x_i and y_i ($1 \leq i \leq n$) being the queue densities of baseline and current output respectively, we define the $Error$ as:

$$Error := \sum_{i=1}^{n} (y_i - x_i)^2$$

Runtime is defined as the time taken by the program in processing the 382 second input video and writing outputs to a text file. All the runtime values used in this report and for plotting the graphs are average runtime values, calculated by running a program which averages their runtime values over 10 runs.

2 Methods

All the four mentioned methods have been implemented, and the comparison of sparse and dense optical flow for dynamic density has also been implemented for extra credit.

1. Method 1 performs **sub sampling of frames**. It takes as parameter the *number of frames to drop (x)*, and uses this parameter to decide the frames to be processed/skipped. Starting from the 1st frame, after processing every N$^{\text{th}}$ frame, it processes the (N+x)$^{\text{th}}$ frame. For the intermediate frames the queue density value is assumed to be the same as that for N$^{\text{th}}$ frame.

2. Method 2 **reduces the resolution** of each frame before processing it to obtain the queue density. It takes as parameters *X and Y, the width and height* (i.e. the resolution) for the new frames.

3. Method 3 **splits the work spatially across threads**, by dividing each frame into multiple parts and giving each part to a separate thread to

process. It takes as parameter the number of splits of each frame, which is equal to the *number of threads* to be used in processing.

4. Method 4 **splits the work temporally across threads**. It splits the total number of frames in the video into several parts and gives each segment of frames to a different thread to process (in parallel with all the other threads). The parameter for this method is the number of parts in which to split the video, which is also equal to the *number of threads to be used for processing.*

5. Method 5 implements **Sparse Optical Flow** to calculate dynamic density for the input video. It does not require any parameter as such.

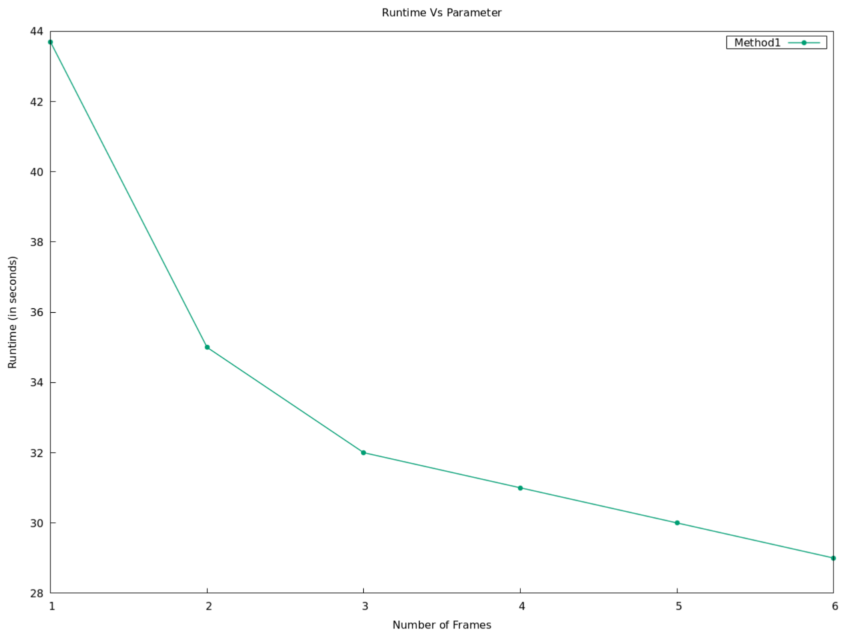
3 Trade-off Analysis

3.1 Sub Sampling Analysis

The table below contains runtime and error values for different parameters.

Number of Frames	Runtime (in seconds)	Error
1	43.7	0
2	35	0.08101
3	32	0.173855
4	31	0.454355
5	30	0.71478
6	29	1.00766

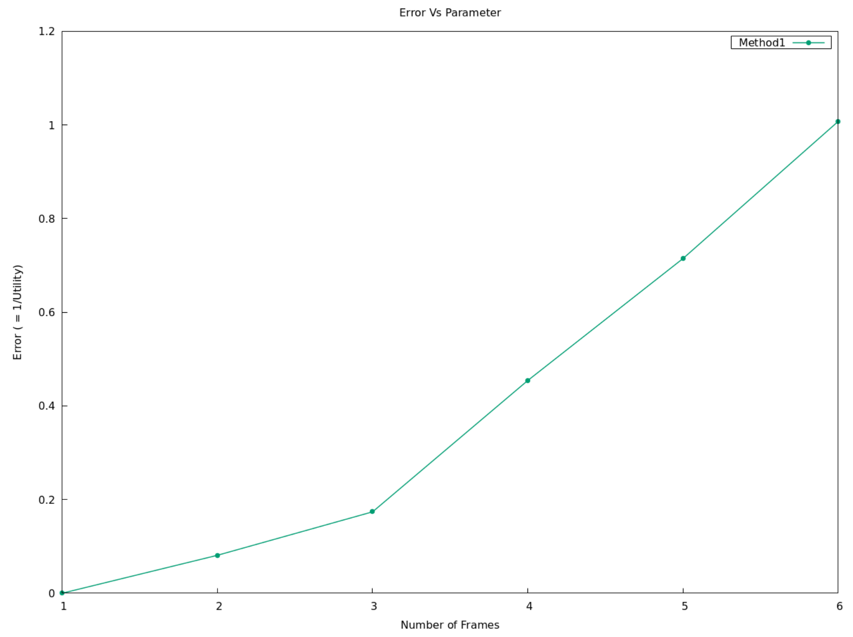
We first plot how the runtime metric varies as we change the parameter for this method. The following graph shows the dependence of runtime on the number of frames dropped while processing.



Clearly, runtime decreases as we increase the number of skipped frames. One can easily draw this result from the fact that the more frames are skipped, the less work the program is required to perform. Less work in the sense that the program does not need to calculate queue density for skipped frames, it can simply copy the previously calculated values, thereby reducing the runtime.

Another observation one can make is that the runtime decreases steeply for the initial increase in the number of skipped frames, but later this decline shrinks. The reason for this behaviour can be: Initially the major contribution to the runtime is by the queue density calculation for frames, but once we start to skip the frames and cut down the runtime with a certain significant amount, the major contribution starts to come from the pre-processing of frames (i.e. operations like homography computation and warpPerspective), operations like frame reading, writing to output file, copying values, etc. And most of these operations are to be performed irrespective of the parameter values. This leads to the negligible or very low decrease in runtime once it reaches a certain minimum value.

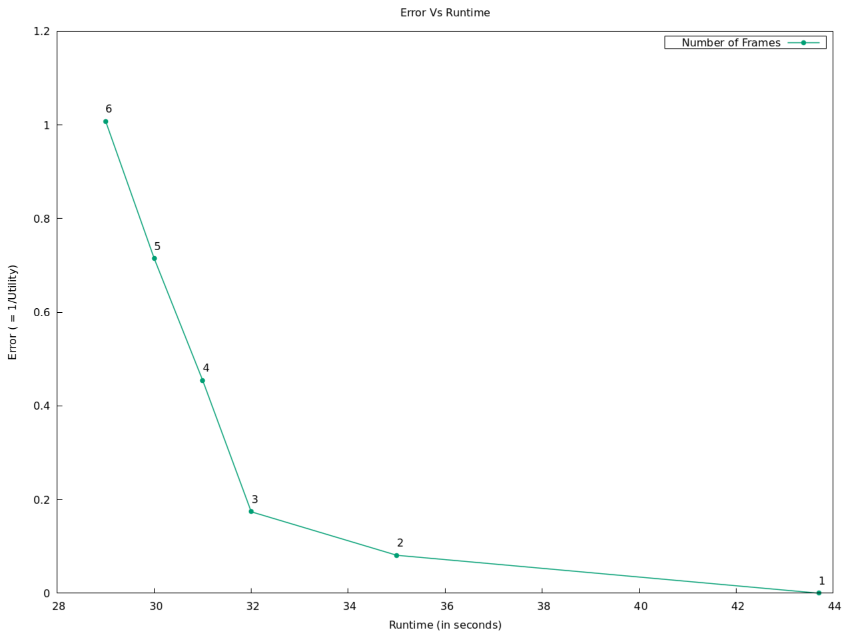
Following graph shows the dependence of error on the parameter values for the program.



Clearly, error increases with increase in the number of skipped frames. Every skipped frame adds a bit to the overall error value and the more frames are skipped, the more error gets added up.

In contrast to the case of runtime, this error does not saturate after a certain number of skipped frames, it keeps adding up. After a certain point, the program runtime would saturate (in some cases, increase) and if one still keeps increasing the parameter value, it would lead to large error values with no further significant decrease in runtime. So, one has to be very careful while choosing the optimal value of parameters.

Following graph shows the runtime and error values for the program, the points are labelled with their parameter values.



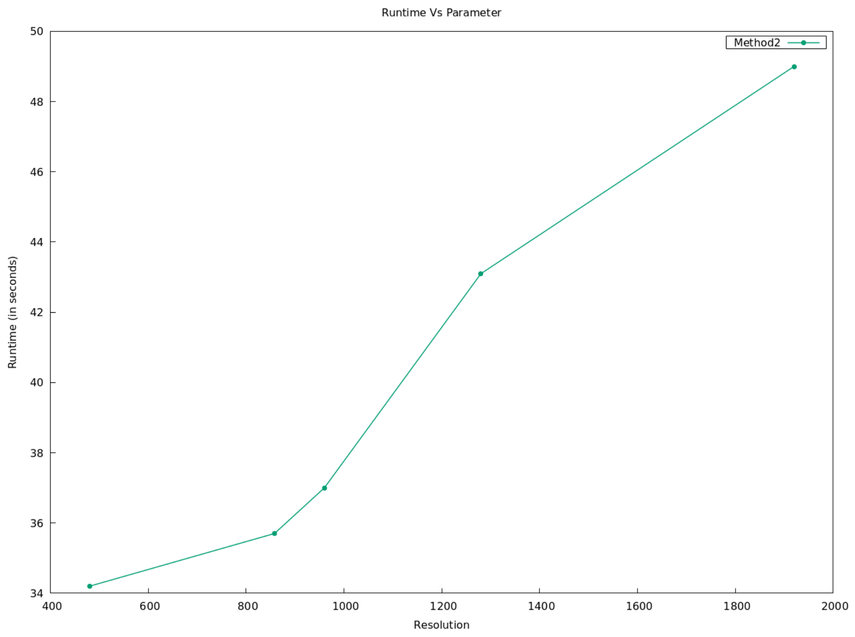
It can be clearly observed from the above graph that error increases very sharply with decrease in runtime (below a certain point), and one can get very large error values with negligible decrease in the runtime. So one has to make a trade-off between a low runtime and a low error value while deciding parameters.

3.2 Reduced Resolution Analysis

The table below contains runtime and error values for different parameters.

Resolution	Runtime (in seconds)	Error
1920x1080	49	0
1280x720	43.1	0.597671
960x540	37	1.04567
858x480	35.7	1.24961
480x360	34.2	1.88142

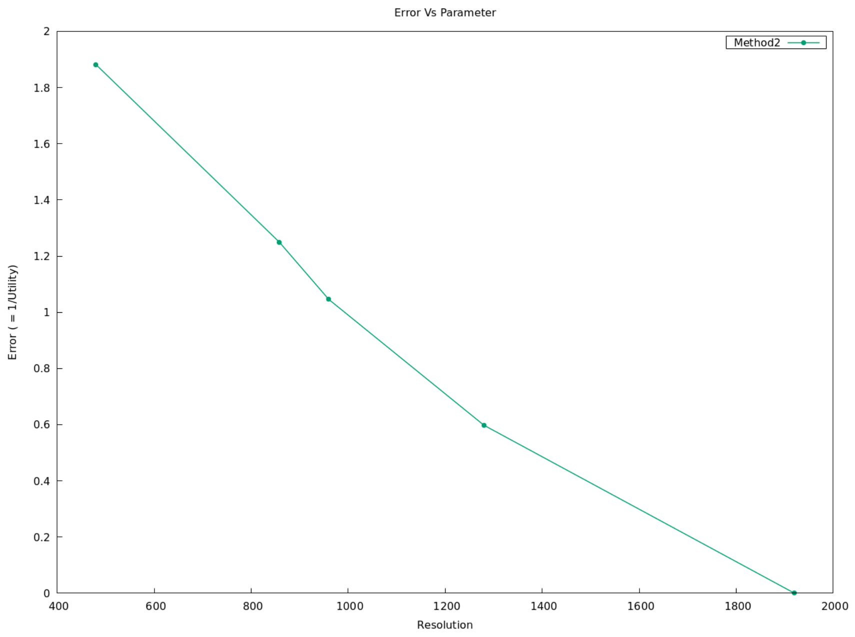
Following graph shows the dependence of runtime on the parameter values for the program, the ”resolution” on the x-axis is the ”width of resolution” parameter. Note that for all the resolution values considered, the resolution also decreases in a scaled manner with decreasing ”width” parameter.



Clearly, runtime decreases with decrease in the resolution. With decrease in resolution, the number of pixels also decreases. Queue density calculation involves the iteration over all pixel values, so with decrease in number of pixels, this iteration time decreases which leads to the overall decrease of runtime.

Similar to the case of previous method, the decrease in runtime is steep for the initial decrease in resolution but later it gradually starts to saturate (in some cases, increases). The reason can be: Initially, the major portion of runtime is constituted by the iteration time mentioned above. So, with reduced resolution, and hence reduced pixels, the runtime decreases sharply. But once we have cut down the runtime to a certain minimum value and have reduced the resolution to a certain amount, the major contribution to runtime is taken over by the resize function (change resolution function). Resize operation is a slow operation in itself. Resolution of the input video is 1920x1080, and the lower the value we want to change it to, the more the time it takes. This leads to negligible decrease (sometimes increase) in runtime below a certain resolution value.

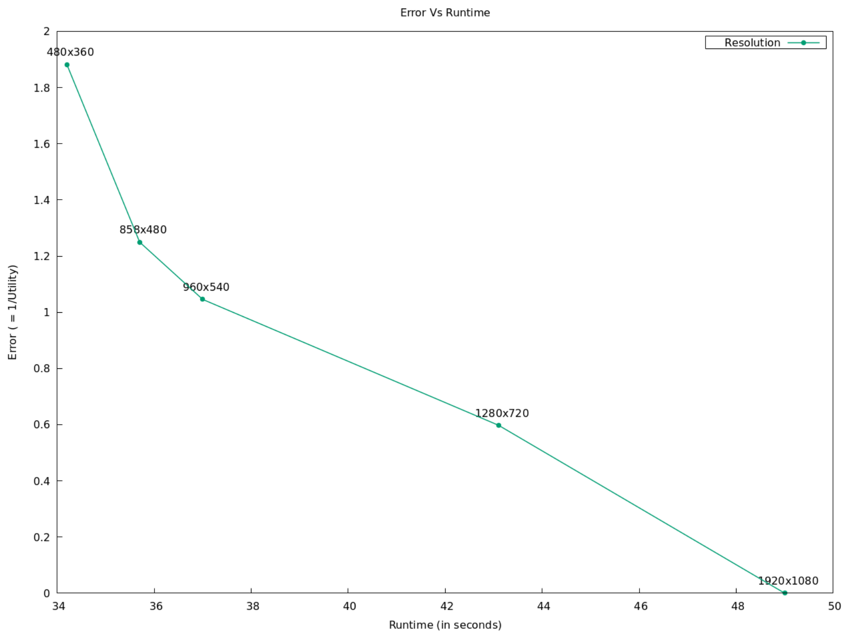
Following graph shows the dependence of error on the parameter values for the program.



Clearly, error increases with decrease in resolution, in fact, it increases sharply even for a small decrease in resolution values. Even if the height/width parameters change by a small amount, the resolution (and hence, number of pixels) changes by a comparatively large amount. The queue density is calculated by counting the number of pixels (after background subtraction) which pass a certain threshold and then dividing by the total number of pixels. So with lesser number of pixels this method is more prone to errors because a wrong decision on a certain pixel (whether to count it or not) would have a higher impact on final queue density value compared to the case when the number if pixels in large.

Again, in contrast with the case of runtime, this error does not saturate after a certain reduction in resolution, it keeps increasing. After a certain point, the program runtime would saturate (in some cases, increase) and if one still keeps decreasing the resolution, it would lead to large error values with no further decrease in runtime. So, one has to be very careful while choosing the optimal value of parameters.

Following graph shows the relation between runtime and error values for the program, the points are labelled with their parameter values.



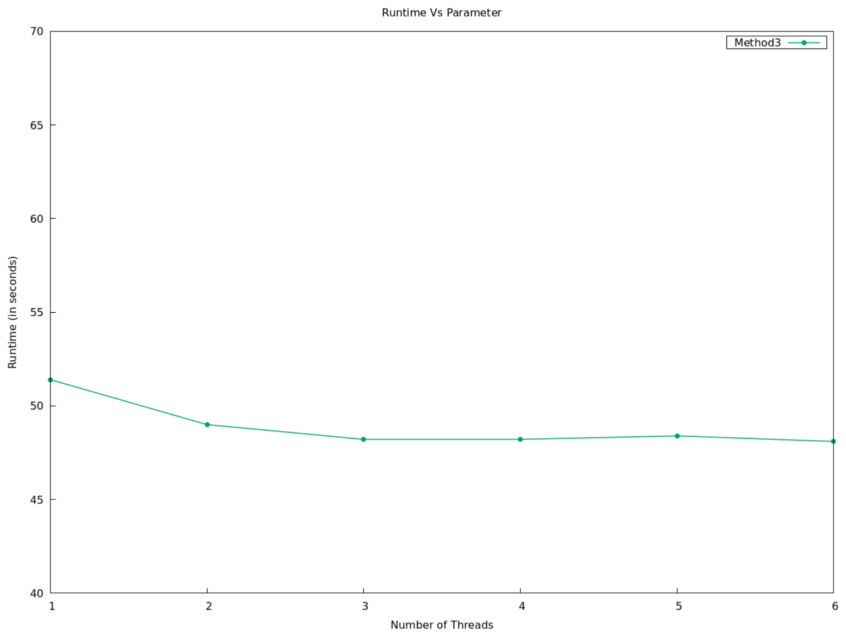
It can be clearly observed from the above graph that error increases very sharply with decrease in runtime, and one can get very large error values with negligible decrease in the runtime. So one has to make a trade-off between a low runtime and a low error value while deciding parameters.

3.3 Spatial Parallelisation Analysis

Our first investigation is how the runtime of the program varies as we increase the number of parallel threads to be used for processing. Increasing the number of threads of course, corresponds to increasing the number of strips we divide each frame into for the computation of the queue density. The table below contains averaged runtime values for different values of the number of threads.

Number of Threads	Runtime (in seconds)
1	51.4
2	49
3	48.2
4	48.2
5	48.4
6	48.1

We plot these values in the following graph to illustrate the variation of the runtime metric with the number of threads we use.

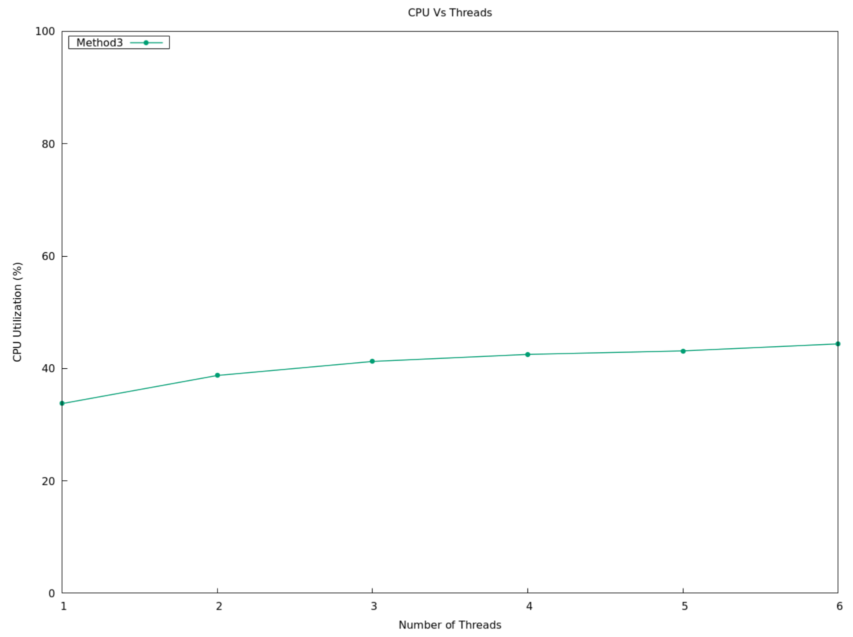


The first trend we observe is that there is a **slight decrease in runtime as we introduce parallelisation**, i.e, move from 1 to 2 threads. Subsequently, the runtime decreases further as we introduce one more thread in parallel, albeit by a very small amount. The trend thereafter is that the **runtime approximately saturates at the same value**, even though we increase the number of threads.

We now propose an explanation for this trend through an analysis of the CPU usage by our program for different numbers of threads. The table below contains CPU utilisation by our program (in %) corresponding to the number of threads we used.

Number of Threads	CPU Usage (in %)
1	33.75
2	38.75
3	41.25
4	42.5
5	43.125
6	44.375

We also plot these values in the graph below to better illustrate how CPU utilisation varies as we increase the number of threads.



The reason parallelisation decreases runtime initially, is because the CPU usage increases as we move from 1 to 2 threads. This means that we now utilize the hardware more efficiently and thus process faster. As we can see from the graph however, it doesn't increase much, which explains why the runtime decreases by a small amount.

In terms of the specific program, when we have 2 threads instead of 1, the program doesn't have to wait to process the first half of each frame before it moves on to the other part. Now each thread can take 2 parts and process them simultaneously. So the runtime decreases slightly.

Now, we also see that the CPU usage saturates after 2 threads, and correspondingly the runtime almost remains the same after 2 threads as well. We believe that that is because this particular method of multithreading does not fit well into our computational process. In particular, the time taken to process each frame in the baseline grew linearly in the total number of pixels. With more threads, the time gets approximately divided by some natural number, however it still remains linear. So the speedup we receive by increasing the number of threads is not very large. In comparison, as we increase the number of threads, the system overhead involved in creating and managing multiple threads increases. The net effect therefore is that runtime per frame remains the same.

This of course does not completely prove why runtime remains the same, because if we can't improve runtime per frame much, we can still try and process multiple frames in parallel to reduce overall runtime. However we are also constrained to process frames sequentially here as we still need to wait for each thread to finish processing one frame before we can move on to the next. The only way to avoid this without some form of synchronisation between threads is to make each thread have its own separate video capture. However, video input is a very costly process, and we don't gain nearly as much from our algorithm by parallelisation (for reasons explained earlier) as we lose due to multiple video inputs and outputs and thread overheads. Therefore after the slight decrease in runtime initially, it saturates if we try to increase the number of threads further.

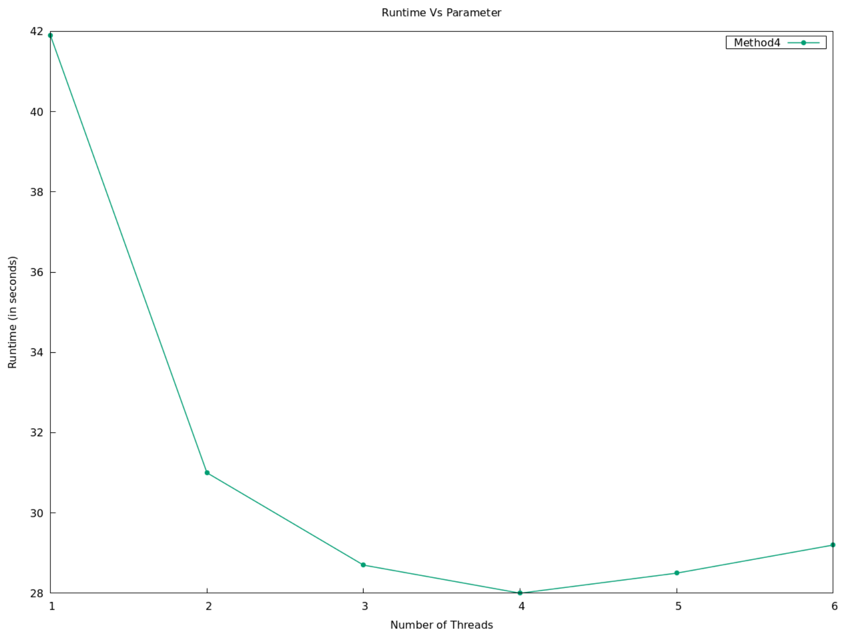
In terms of utility, this method preserves the outputs we use for our baseline. Therefore the error introduced is 0.

3.4 Temporal Parallelisation Analysis

In this method, we divide the set of all frames of the video into segments and give each segment to a separate thread to process. As we will analyse below, this opens up possibilities for speeding up our program greatly as we no longer have to process frames sequentially. We now backup our claim with numbers and justifications. The table below contains averaged runtime values for different numbers of threads used. The number of threads used is also equal to the number of segments into which we divide all the frames of the video.

Number of Threads	Runtime (in seconds)
1	41.9
2	31
3	28.7
4	28
5	28.5
6	29.2

We now plot these values on the graph below to illustrate the variation of runtime with the number of threads we use for processing.

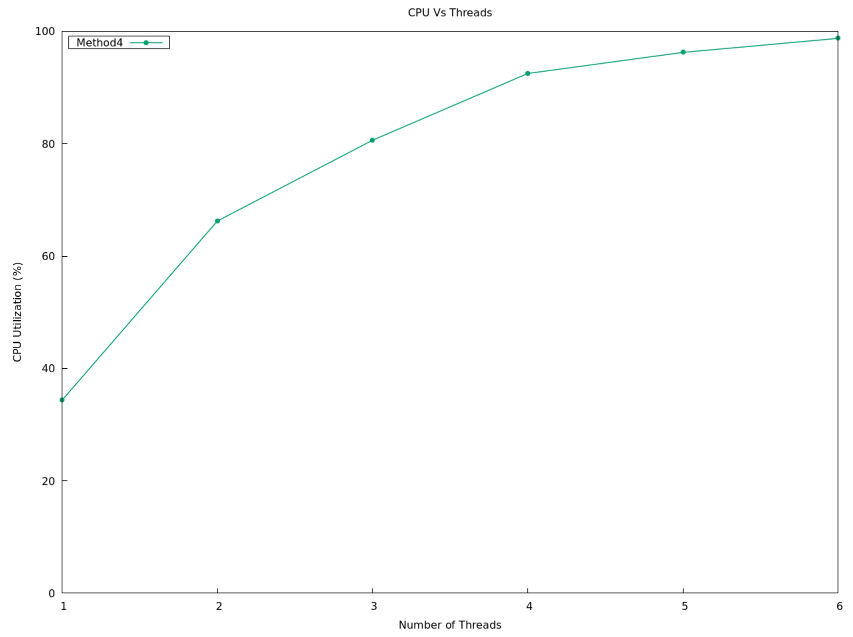


We observe that the **runtime decreases sharply as we introduce parallelisation**, i.e., as we move from 1 to 2 threads. It further decreases when we increase the number of threads to 3, albeit not as sharply. Thereafter we see a very small decrease which also turns into a small increase afterwards. So the benefit that we get out of multithreading **saturates after 3 threads**.

This trend in the runtime depends critically on how efficiently we are able to utilise our CPU. So we explain our graph with an analysis of the CPU utilization for different numbers of threads. The table below shows these values for our program.

Number of Threads	CPU Usage (in %)
1	34.375
2	66.25
3	80.625
4	92.5
5	96.25
6	98.75

We now plot these values on the graph below to better illustrate how CPU utilization varies as we vary the number of threads used for processing.



The whole reason that parallelising our program improves the runtime is because we are able to utilise our hardware more efficiently. This is confirmed by the fact that the sharp decrease in runtime we saw earlier is accompanied by a sharp increase in CPU Utilization. This happens because we have essentially removed a constraint that was there on our program earlier, and that frees us up to increase the hardware efficiency. What was that constraint? Well it was the one which prevented us from experiencing significant gains from spatial splitting of work in Method 3; the requirement of processing frames sequentially. What this method essentially does is identify that sequential processing of frames really makes no difference to the result of our algorithm, and so removes that constraint and uses the freedom to make processing faster. We see that utilization also increases as we move from 2 to 3 threads, however not as sharply. This is again in agreement with the fact that runtime also decreases correspondingly, and also not as sharply.

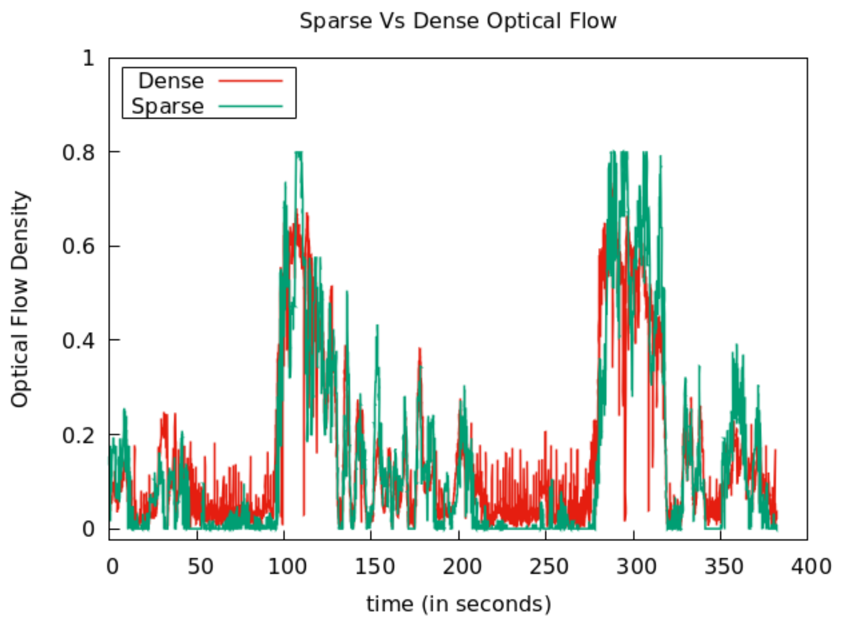
Now, after 4 threads, the increases in CPU utilisation become small and correspondingly the runtimes also saturate or increase slightly like we saw earlier. This is because after we have reached a certain level of CPU utilization, we are at a point where it is difficult to utilise much more of the CPU, and so successive gains get smaller and smaller. The gains that we obtain by adding another thread start to fall below the cost of adding another thread. These costs include various system overheads for creating a thread, as well as the allocation of resources for managing multiple threads. So if we don't get a significant gain in terms of CPU Usage these costs can actually increase the overall runtime, which is indeed what happens after 4 threads in this case.

It is also worthwhile here to contrast this with the previous strategy for multithreading. We explained earlier how that decreased the runtime per frame but only marginally. The previous strategy also imposed the constraint of sequential processing of frames for reasons explained earlier. This prevented us from optimising by processing multiple frames in parallel. In return for keeping the runtime per frame same (which is not a very significant loss), this method allows us to remove the constraint of sequential processing. We are now able to process multiple frames in parallel, and this leads to a sharp gain as compared to doing this without parallelization. It therefore turns out to be a much better strategy to use multithreading for this particular computational process.

In terms of utility, this method preserves the outputs we use for our baseline. Therefore the error introduced is 0.

3.5 Sparse Optical Flow Analysis

The graph below shows the optical flow density calculated by sparse and dense optical flow methods.



Sparse Optical Flow uses *Lucas Kanade Method* to calculate dynamic density. This method is significantly faster than the dense optical flow method because it performs optical flow computation only on a sparse feature set (limited set of points detected by *Shi-Tomasi Algorithm*) whereas dense optical flow method computes optical flow for all the points in the frame. For the input traffic video, where dense optical flow takes around 360 seconds to complete processing, this method consumes only 45 seconds. Although we can significantly cut down the runtime using this method, it introduces errors in dynamic density calculation compared to dense optical flow (as we can observe from the above graph). Dense Optical Flow might be slow, but it gives us reliable results. So we have to make a trade-off between a low runtime and accurate results while deciding the method to be used for dynamic density calculation.