Traffic Density Estimation using OpenCV
Analysis of Trade-offs in Software Design

Ishita Chaudhary 2019CS10360
Swaraj Gawande 2019CS10406

31 March 2021

1 Metrics

The aim of this report is to analyze trade-offs between different methods and its parameters while optimizing the program to estimate queue density using background subtraction. This includes analyzing the outputs and plotting their graphs against several factors, such as, number of threads, utility (defined as inverse of error), run-time and size factor (for method 2).

For error calculation, the output of subtask 2 is taken as the baseline. Error is calculated as the absolute difference of fraction of non-zero pixels after background subtraction in the frame from output of subtask2 and the corresponding frame from the method from subtask3 added up for all frames and divided by total number of frames(average of absolute difference, between corresponding frames).

2 Analysis of different methods

2.1 Method 1: Sub-sampling Frames

This method processes every x frame, where x is the parameter to be given from command line. The method will process (N+x)th frame after (N)th frame, the value of queue density between these subsequent frames is obtained by considering the queue density value of frame N.

Consider the figure below, describing the Utility vs Run-time Pattern for x in range [1,20].

The right-most point has the value of x as 1, it decreases as we go left, with the left-most point having the value of 20.

From this we can conclude that, as we skip more number of frames (as we go left in the graph), the run-time of program decreases drastically. This also reduces the utility of the program as more errors are generated due to ignoring calculation for large number of frames.

It can also be noticed that x=3 had an utility of about two thousand owing to the fact that baseline (output of subtask 2), was being processed at the same rate. So the figure given below is the not having the data corresponding to parameter x=3 as this would require a different scale and mask the variations in all other data points.

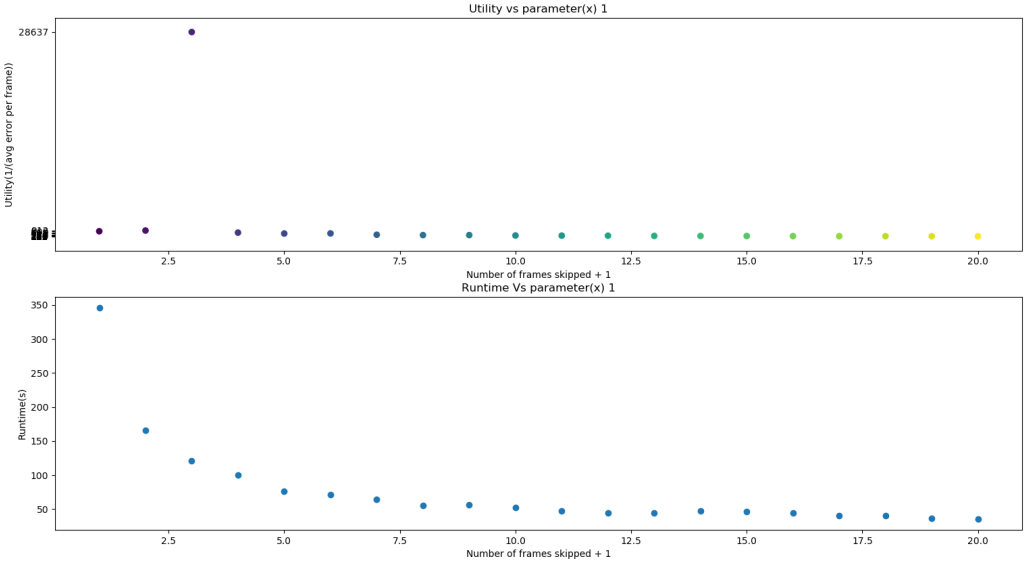

Figure 1: Utility vs Run-time for Sub-sampling Frames

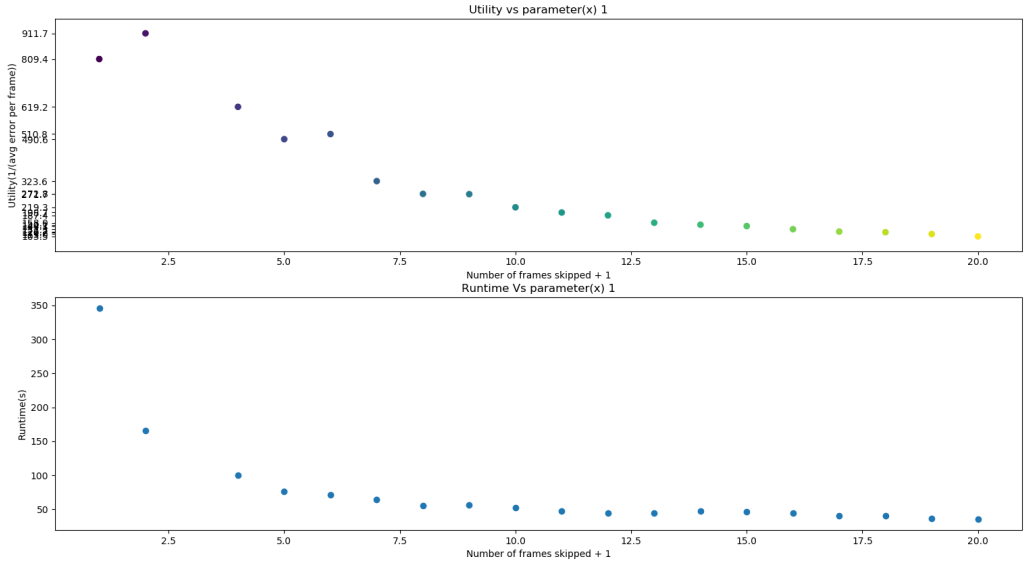

Figure 2: Utility vs Parameter and Run-time vs parameter for Sub-sampling Frames

2.2 Method 2: Reducing Resolution

In this method, the resolution of each frame to be processed (video file processed at 5 fps) is reduced. The parameters given are X and Y, such that the resulting resolution is XxY. The height and width of the frame are decreased by the same ratio, that is, the size factor. Since, we are decreasing the resolution, size factor is always less than one. Size factor equal to one indicates the original resolution of frame. Also we can observe that there is no significant difference in run-times in changing the resolution of frame on which we are applying background subtraction, this behavior is because the resize function is itself quite costly and the background subtraction no more determines the run-time however utility increases with size with some exceptions and is maximum for original resolution as expected. The graph below indicates

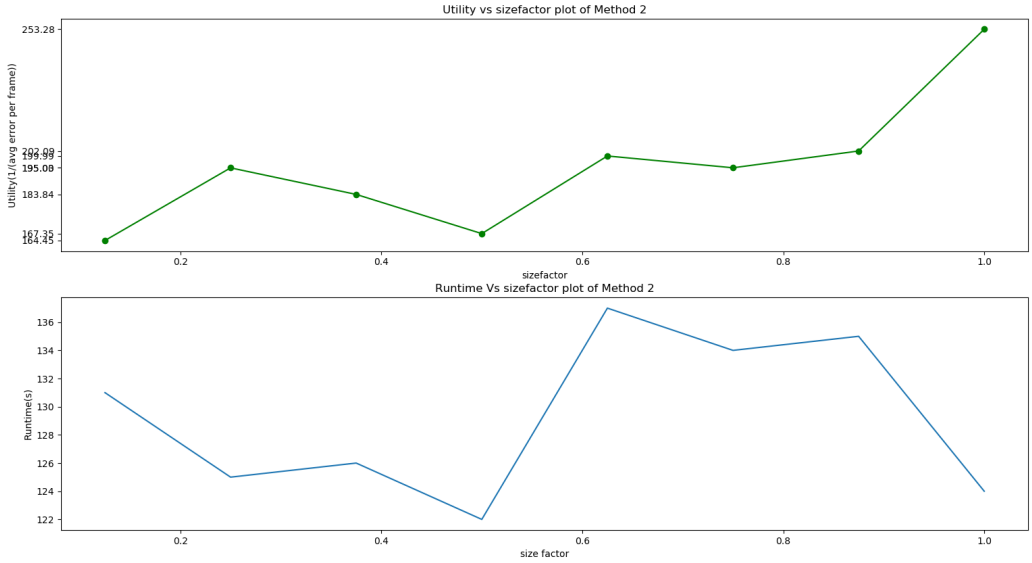

Figure 3: Utility vs Run-time for reduced resolution of frames

2.3 Method 3: Multi-threading: splitting work spatially across threads

The video is processed at 5 fps, and each frame that is supposed to be processed is divided into x parts, where x is the parameter given in the command line. These x parts of a frame are processed by x different threads concurrently. The work is split spatially across all the threads.

Consider the figure below, first graph describes the Utility vs Run-time Pattern for x in range [1,16]. The right-most point represents a single thread, while the left-most represents 16 threads.

We can deduce that the utility (or error) does not change much with the number of threads, which is in fact reasonable because we are just dividing a frame into multiple parts and assigning these parts to different threads to run parallel with each other. But as we can notice, The run-time however in this case is quite different from what we would expect it is because the additional computations required to make strips and the time to create and join the threads is more than the actual time required in background subtraction in a thread.

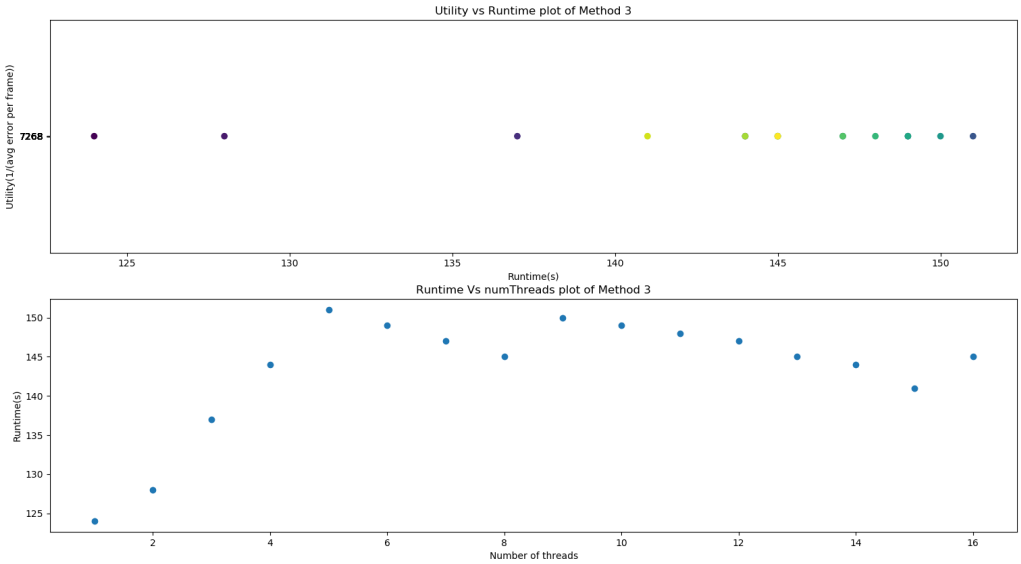

Figure 4: Utility vs Run-time for splitting work spatially across threads

We then used mpstat to get CPU utilization for each case and plot it against number of threads used and against runtime. It is however clear that using more threads would mean more efficient use of hardware and the CPU utilization indeed increases with the number of threads used but as discussed earlier is not the compution in the threads which determines the runtime here so although we are processing more instructions per unit time we are also performing more number of instructions in total in preparation of the frames for spatially allocating the work to threads combined with the time required for allocation work and waiting to join them is domination the actual time required for background subtraction And so we have the random behaviour in CPU utilization vs Runtime graph.

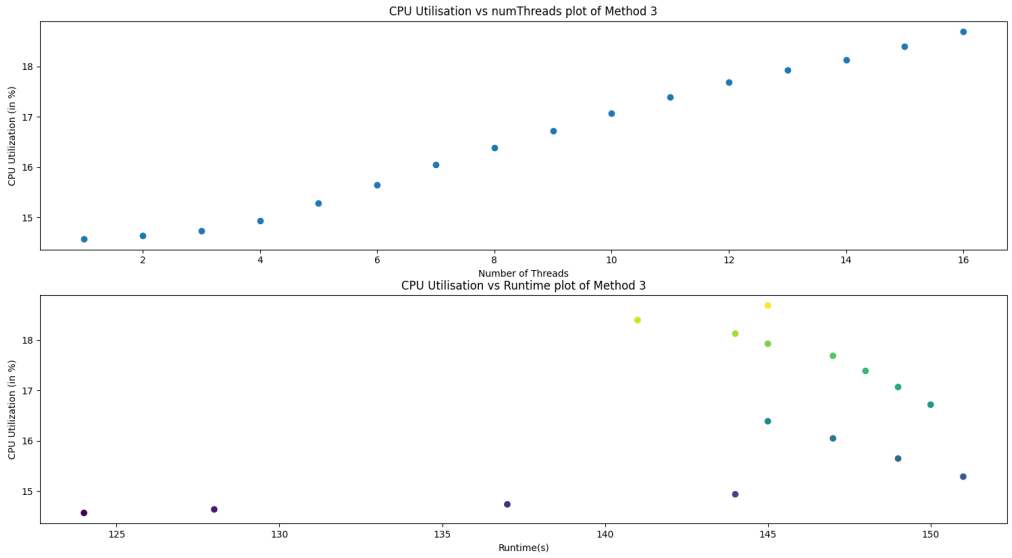

Figure 5: CPU utilization vs number of Threads and CPU utilization vs Run-time for splitting work spatially across threads

Following is a screenshot of terminal which shows CPU usage for some of the initial datapoints for above graph.

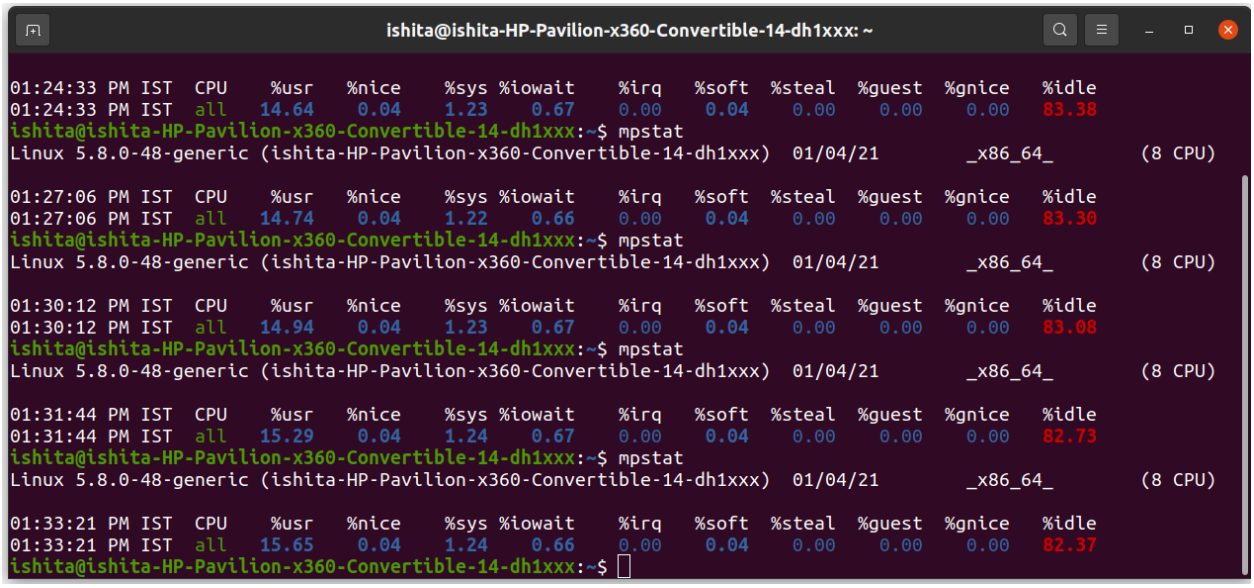

Figure 6: CPU utilization

2.4 Method 4: Multi-threading: splitting work temporally across threads

This method processes every x frames in x different threads which run parallel to each other, where x is the parameter to be given from command line. The video file is processed at 5 fps as in task 2, but every consecutive frame that is to be processed is given to different thread, and the work is split temporally across them. After the given number of threads are assigned work we then join all the active threads and then once again start assigning them frames.

Consider the figure below, first graph describes the Utility vs Run-time Pattern for x in range [1,16]. The right-most point represents a single thread, while the left-most represents 16 threads.

As we can observe, the utility does not change with the number of threads, as we are just assigning our frames to different threads, hence almost same error is generated for different number of threads. Also the error is very small giving very high utility or say the method is applicable if utility is considered for any number of threads.

Whereas the run-time decreases drastically as the number of threads is increased from one to six. After which there was a reduction but not very significant. These threads run parallel to each other, and hence, decrease the workload on a unit thread to result in significantly lesser run-time. Also the threads after eight were having similar run-times and having more threads would not make much of a difference. Also the reduction in run-times is also decreasing with increasing number of threads as the threads only help in reducing the time for background subtraction and some computations but a part of tasks which need to be performed outside the threads require same amount of time. Then we used mpstat to get the avg value of CPU utilization in each case and plot the values in graphs against the number of threads and against runtime. From these graph it is clear that by having more threads helps us in use the hardware more efficiently and thus compute more instructions per unit time which reduces the total runtime as evident in the second graph the right most point is with 1 thread has high runtime and low CPU utilization as the CPU utilization increase runtime decreases.

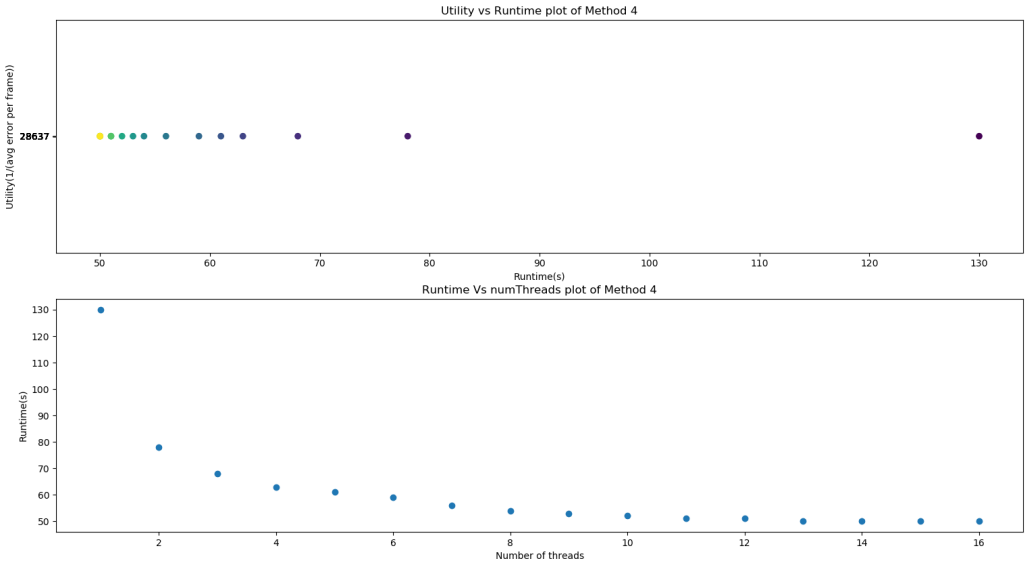

Figure 7: Utility vs Run-time for splitting work temporally across threads

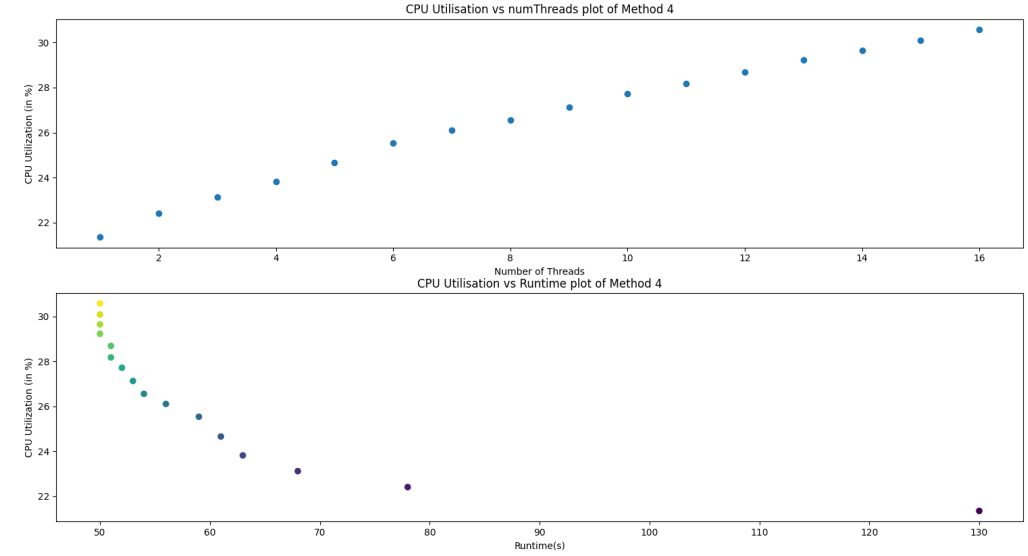

Figure 8: CPU utilization vs number of threads and CPU utilization vs Run-time for splitting work temporally across threads

Following is a screenshot of terminal which shows CPU usage for some of the initial

datapoints for above graph.

Figure 9: CPU utilization