Traffic Density Estimation using OpenCV Functions

COP290

Tushar Singla
2019CS10410
Computer Science & Engineering

Pranay Gupta
2019CS10383
Computer Science & Engineering

Course Co-ordinator
Prof. Rijurekha Sen
Assistant Professor
Department of CSE

1 Metrics

Utility- Our utility is the root mean of the weighted sum of the absolute differences. It gives us thy percentage error in calculating the queue density by a certain method when compared to baseline(i.e. subtask-2)

$$U = \sqrt{\frac{1}{n}\sum \frac{(x-y)^2}{x^2}} \cdot 100$$

Runtime- Runtime is defined as the time taken between the start of the program and the end of the program. Variable declarations and user inputs are excluded from the runtime. It is measured in seconds.

2 Methods

All 4 methods have been implemented succesfully.

2.1 Method 1

This method takes an input x from the user. The density is calculated for every alternate x frame and the rest of the frames store the last computed queue density value. Here, time is saved because we do not have to transform and crop every frame of the input video and we do not have to calculate queue density for those frames, thereby saving time, but there is a corresponding error in the output queue density because we are assuming intermediate values to be the same as the last computed value. The x is out parameter which we vary to do out trade-off analysis.

2.2 Method 2

This method takes an input resolution from the user. The reference image is scaled down to this resolution using the resize function. Every frame of the input video is also converted to this resolution and the density is calculated between the final blank-road image and frames at the input resolution. Time is saved because we need to traverse through a smaller sized matrix during density calculation. There is a corresponding error because there might be errors during resizing and the scaled down image might not contain the entirety of the information of the original image. In this method the resolution(XxY) is our parameter for trade-off analysis.

2.3 Method 3

This part takes the number of threads as input. The program is split into multiple threads. The cropped and transformed images of the frames of the video are computed and stored. Then they are passed as a stream to the threads and every thread does the computation over a small part of each frame and not the whole frame. The overall density is calculated by summing all these values for a given frame from different threads and dividing it by the total number of pixels. Time is saved because multiple threads run synchronously to compute the density, thereby saving time. There is no error in this method because exact same computations are being done synchronously.But the down point of this method is the extra memory usage. Here the number of threads is out parameter for the trade-off analysis.

2.4 Method 4

This part also takes the number of threads (x) as input. The program is split into multiple threads. Every thread goes through the video frame by frame, but the calculation is done for certain frames and not all. This saves time because densities for multiple frames are being calculated synchronously. There is no error in this method because the exact same computations are being done just by different threads. As we keep on increasing the number of threads the memory usage keeps on increasing as all threads try to excess the video simultaneously. Here also, the number of threads is out parameter for the trade-off analysis.

3 Trade-off Analysis

3.1 Method 1

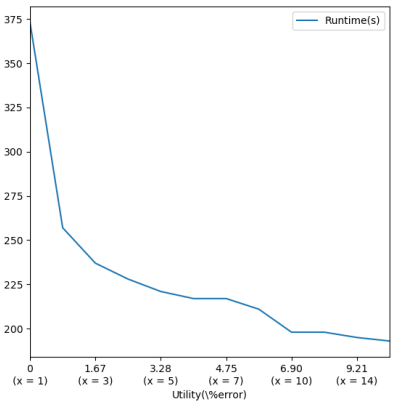

Figure 1: Runtime vs Parameter

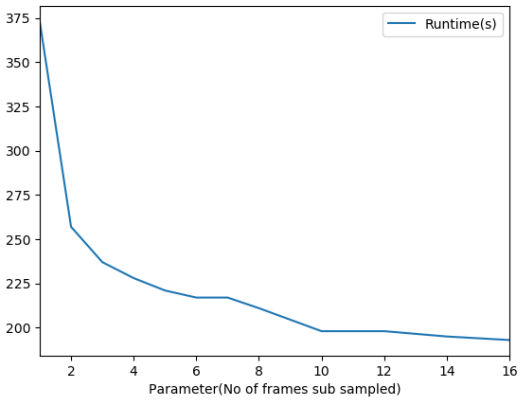

Figure 2: Utility vs Parameter

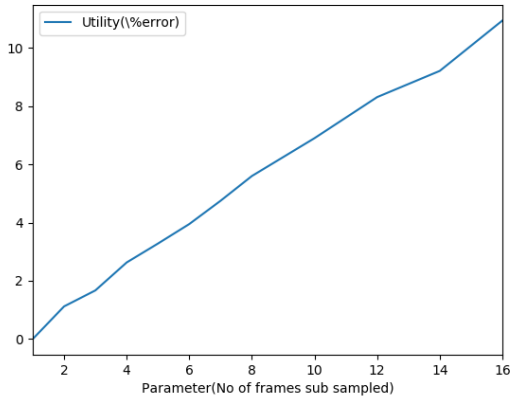

Figure 3: Runtime vs Utility

In this part, we observe that as the value of the parameter x increases, the runtime decreases and the error increases. This is because we don't have to transform and crop the frames and we don't have to find out the density for the intermediate frames. Higher the parameter value, lesser is the number of frames that we have to do calculations for, thereby gradually reducing the runtime. The runtime is not decreasing multiplicatively because there are other overhead times like time to stream the video and other variable initialization times (which have to be stored/accessed from the memory) by the program. As the value of x increases, the error also increases because the density of the intermediate values has changed by a lot in comparison to the baseline.

3.2 Method 2

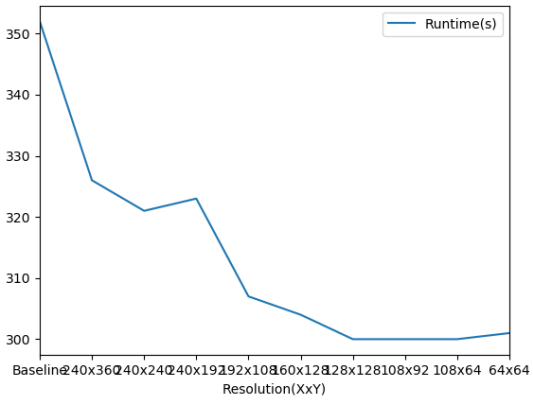

Figure 4: Runtime vs Resolution

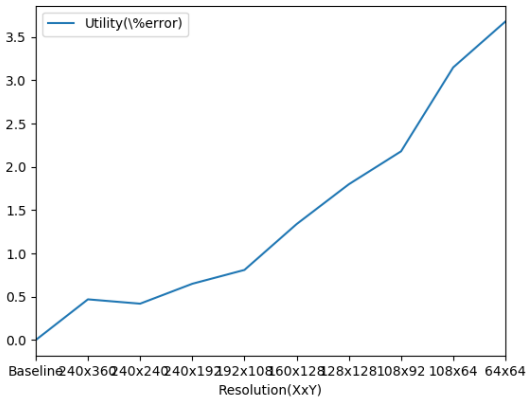

Figure 5: Utility vs Resolution

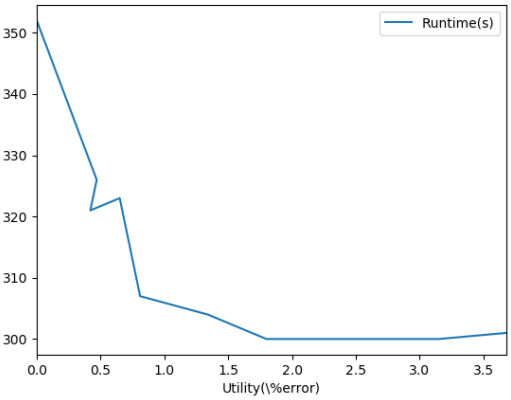

Figure 6: Runtime vs Utility

In this part, we observe that runtime first decreases significantly and then becomes almost constant. This is because there is an increase in runtime due to the resize function of opencv and time is being saved in computing density for a smaller matrix. Initially, a lot of time is saved in computing density on a smaller matrix compared to the resize function that takes some time to compute the new matrix each time. As we decrease the resolution, the resize function takes larger time which is balanced by the time saved in computing the density making the overall runtime of the code almost constant. There is some error in this method because opencv uses the inter linear method for scaling down the image which makes use of the neighborhood of the image to modify the image. As a result, there might be some values which are incorrectly classified or some information might be lost on resizing creating an error in comparison to the baseline.

3.3 Method 3

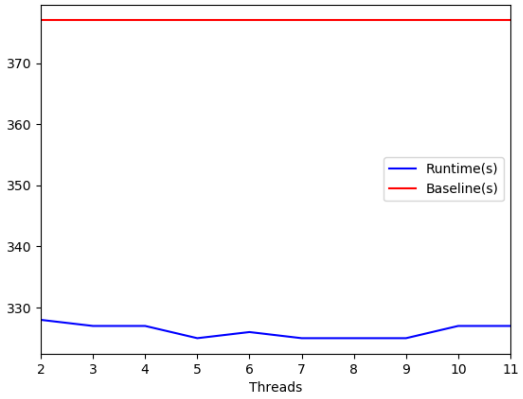

Figure 7: Runtime vs Threads

Runtime Analysis- In this part, we observe that there is a drop in the runtime from the baseline but the runtime remains almost constant on increasing the number of threads. This is because the majority of the runtime is used for transforming all the frames of the input video and storing them in a vector. The remaining computation of density accounts for a smaller part of the runtime. Moreover, as the number of processors are limited thus all the threads do not run simultaneously and wait for a processor to get free. As a result, the time saved in calculation and the time lost in waiting by the threads almost balances out making the runtime almost constant. However, this method saves significant time with respect to the baseline, because the density calculation saves more time than the time wasted in waiting.

Memory analysis- Since, we are storing all the frames coming from the input video in a vector, the memory cost is very high (represented by the peak memory usage). The memory cost is almost similar for any number of threads because the majority of the memory is being used in storing the frames which remains the same for all the threads. The peak memory usage is 1.4 GB for a 180 MB video because videos are stored in an encoded format (only the pixels that have changed in subsequent frames are stored). However, when we store the that video in a vector, the video is stored in a decoded format, making the memory usage much larger than the size of the video. **Peak Memory Usage- 1.4 GB**

3.4 Method 4

Runtime Analysis- In this part, there is a significant drop in the runtime initially and the runtime starts increasing as soon as the number of threads are increased. This is because each thread in the code opens a separate instance of the input video and calculates the density for predetermined set of frames from those. Initially, this saves time as the time to open 2/3/4 instances of a video is less than the time saved from synchronously computing the densities. After that, the runtime cost of opening more instances of the input video becomes larger than the time saved by the synchronous

computation. As a result, the graph dips down from the base line, reaches a minimum value and then goes on increasing.

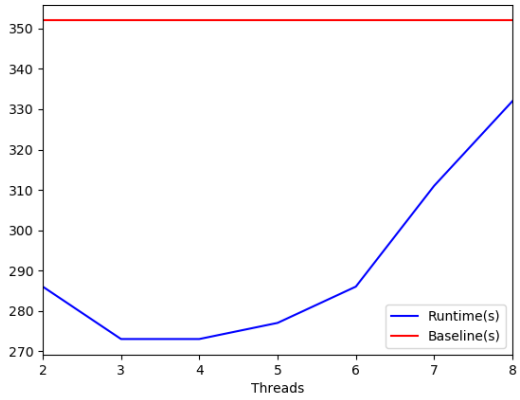

Figure 8: Runtime vs Threads

Memory Analysis- Since, we are opening separate instances of the video for each thread, we observe that the memory usage depends almost linearly on the number of threads used.

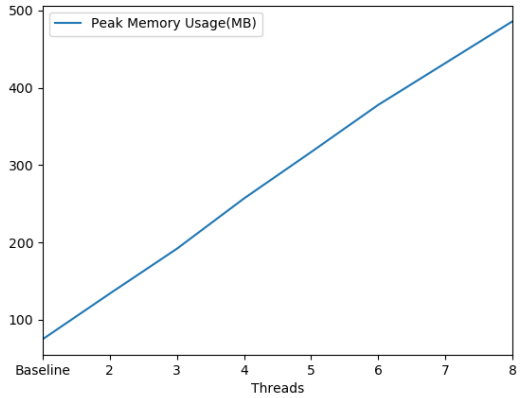

Figure 9: Memory vs Threads

No of Threads	**Peak Memory Usage(Mb)**
Baseline	70
2	134
3	192
4	257
5	317
6	378
7	432
8	486