Understanding and Analyzing Trade-Offs in Software Design

Prof. Rijurekha Sen

Hardeep Kaur 2019CS10354 | Devanshi Khatsuriya 2019CS10344

Assignment 1
COP290, Design Practices
Indian Institute of Technology, Delhi

April 1, 2021

Abstract

Traffic monitoring in many countries is the biggest concern, due to overcrowding, increasing daily commuters, globalization, shortened or narrow roads. Improper architecture in the signaling system and avoidance of traffic rules are affecting the traffic inflow and leading to congestion. Malfunctioning at traffic management sources ending up suing the wrong individual for rule violation. Traffic congestion at various intersectional crossings is leading to pollution and global warming that affect the internal city environment. Long waiting hours in traffic is an expensive deal for commuters due to high fuel consumption. Operating better road networks for free traffic flow has a direct impact on the productivity of the economy.

Contents

Abstract i

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 1
- 1.3 Objectives . . . 2
- 1.4 Contribution . . . 2
- 1.5 Thesis outline . . . 2

2 Metrics 3

3 Methods 4
- 3.1 Method 1: Sub-Sampling Frames . . . 4
- 3.2 Method 2: Reducing Resolution . . . 5
- 3.3 Method 3: Splitting each frame across different threads . . . 6
- 3.4 Method 4: Giving consecutive frames to different threads . . . 7

4 Results and Analyses 8

List of Figures

1.1 A typical view from a traffic camera in Lajpatnagar junction in New Delhi 1

3.1 Runtime Utility Plot for Method 1 4
3.2 Runtime Utility Plot for Method 2 5
3.3 Runtime Utility Plot for Method 3 6
3.4 Runtime Utility Plot for Method 4 7

Chapter 1

Introduction

1.1 Motivation

Traffic management and control, due to infrastructure constraints and rising number of vehicles, is a complex task and requires application of dedicated algorithms together with precise traffic data. The information about number of vehicles and their types is helpful in reducing travel times and emissions. It allows us not only to increase effectiveness of traffic control, but also to adapt management policy to changing conditions and predict infrastructure bottlenecks.

1.2 Problem Statement

Estimating traffic density for vehicles which are waiting for red signal in the straight stretch of the road going towards north in the given view.

Figure 1.1: A typical view from a traffic camera in Lajpatnagar junction in New Delhi

1.3 Objectives

1. Calculating queue density and dynamic density of vehicles in the given stretch.
2. Optimizing the software for accurate and fast results.

1.4 Contribution

1. An intelligent traffic management system provides an advantage by offering safe public transportation, strict punishments on violating traffic rules, ticketing system automation.
2. An intelligent traffic monitoring system may contribute in the development of efficient and reliable navigation systems.

1.5 Thesis outline

This paper presents the various approaches made for multi-metric optimization of traffic density estimation software, followed by a careful analysis of the various trade-offs involved due to conflicting requirements.

Chapter 2

Metrics

In the process of building software, multiple metrics need to be optimized, to reduce risks and eliminate failures. In our case of traffic density estimation, following metrics could be considered :

1. Accuracy : The code should output the correct density values.
2. Latency : For a given input frame, the output should be produced within a small time.
3. Throughput : The code should generate high number of outputs per unit time.
4. Temperature : Keeping the processor temperature under control becomes important for deployment in regions of high temperature, where cooling be external means is not possible.
5. Energy : The software should be energy efficient, in the sense that it could work even in limited supply of energy.
6. Security : Protecting the software from hackers is crucial.

Baseline : The baseline code processes each frame of the given video and generates queue density.

In case of conflicting requirements, we need to do some trade-offs. In this report, we consider utility, i.e., accuracy and runtime as the metrics to be optimized. The report consists of a detailed discussion of different methods used to optimize these metrics and a careful analysis of the trade-offs involved in optimization.

The formula used for calculation of utility for a given frame - queue density output file (generated by any of the 4 methods with any parameter value) with respect to the baseline output is as follows:

$$Error(percentage) = \sum_{n=0}^{num\_frames} \frac{|QD_n - BD_n|}{BD_n} \times 100$$

$$Utility(percentage) = 100 - Error(Percentage)$$

where QD_n and BD_n are the queue density and baseline queue density for the n^{th} frame.

Chapter 3

Methods

3.1 Method 1: Sub-Sampling Frames

The method involves skipping a fixed number of frames, say x, for each frame processed. The number of frames skipped every time, i.e. x, forms the parameter. So, after processing frame number N, frame number N+x would be processed. For all the intermediate frames, we use the queue density obtained on processing frame N.
Since the number of frames processed per unit time is less than that in baseline, a decrease in the total processing time (runtime) is expected. Moreover, since this method involves a loss of information, a decrease in utility of output values is also expected.

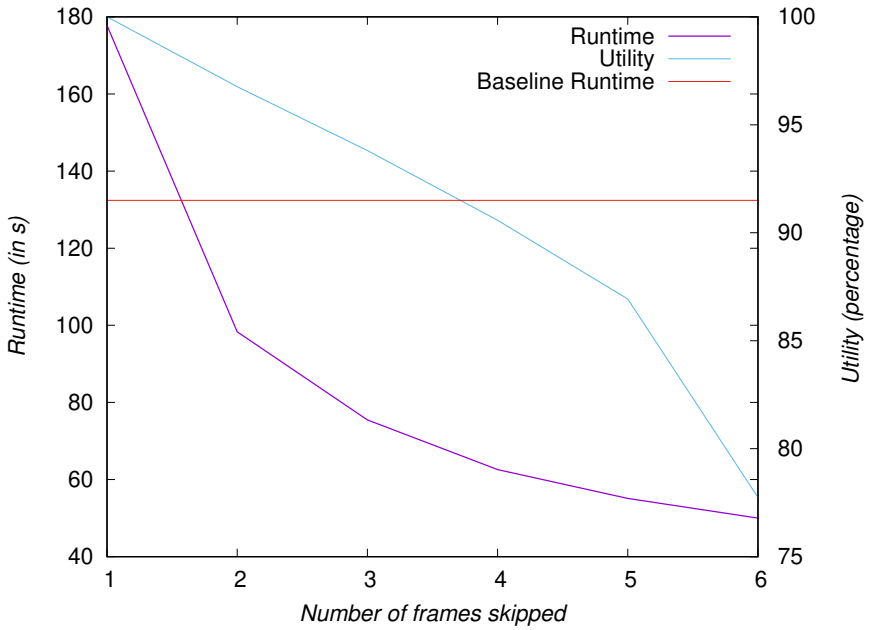

Figure 3.1: Runtime Utility Plot for Method 1

We observe that increasing the number of frames skipped leads to a decrease in both the runtime and utility. Hence, the trend obtained in both the plots is in agreement with the expectation.

3.2 Method 2: Reducing Resolution

In this method, each frame is processed with a lower resolution. The aspect ratio is maintained during the process. Hence, the only parameter for this method is the resolution factor. If the resolution factor is 0.6, the image would have 0.6 times the original resolution. So, it the original resolution in pixels was $x \times y$, the image would be $0.6x \times 0.6y$.
Since lower resolution frames might be processed faster, a decrease in resolution factor is expected to produce a decrease in runtime and vice versa. Since downsampling leads to loss of information, the accuracy of output values is expected to be lesser than that of baseline. Moreover, since the aspect ratio is maintained while lowering the resolution, the density values and hence the utility is expected to be almost constant.

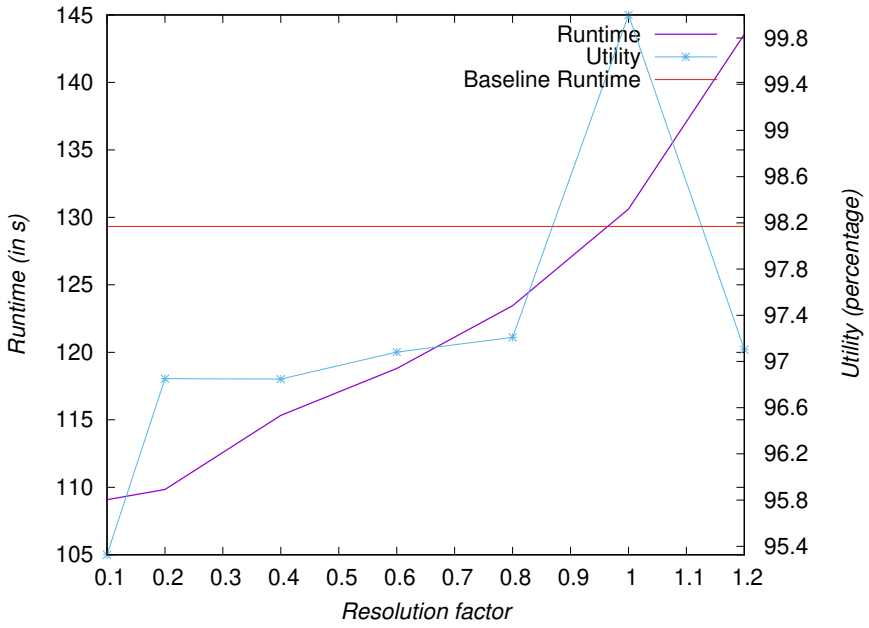

Figure 3.2: Runtime Utility Plot for Method 2

We observe that the runtime decreases with decreasing resolution factor. The utility comes out to be 100% at resolution factor = 1 and less than 100% for rest of the values. Hence, the curves show the expected behaviour.

3.3 Method 3: Splitting each frame across different threads

This method involves multi-threading for queue density estimation. Each frame is split into a given number of divisions, say x, and each division is given to a separate thread to be processed. The number of threads, x, forms the parameter here. The splitting can be performed either vertically or horizontally. The threads work parallelly to process their division of the frame and calculate queue density. For each frame, the density values obtained from all the threads are added up to get the queue density for the frame. The threads used are application level pthreads in C++.
As the number of threads increase, the processing becomes more parallel and a decrease in runtime could be expected. But since the work performed after multi-threading is minimal, not much efficiency in time is expected. Also, splitting each frame into the required number of divisions may increase runtime. Moreover, after processing one division of a frame, a thread has to wait for all other threads to complete execution and join. This, along with the overhead in creating and joining threads may lead to an overall increase in runtime even as the numbers of threads increase.
Finally, as each frame is being processed in a way similar to the baseline, the utility is expected to be similar to that of the baseline.

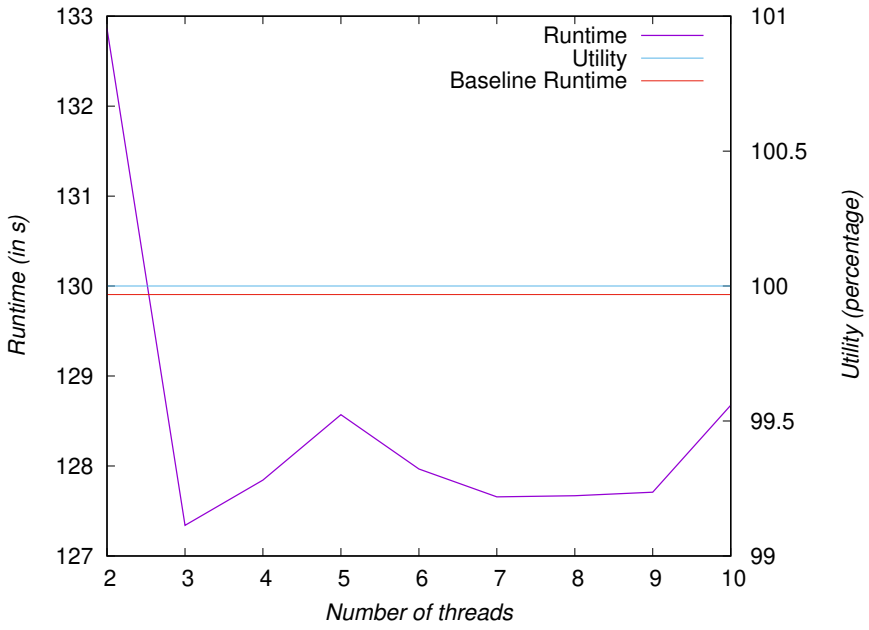

Figure 3.3: Runtime Utility Plot for Method 3

From the above plot, we observe that multi threading has lead to a decrease in the total processing time, increasing the time efficiency. However, it doesn't follow a regular trend as the number of threads are increased. This may be due to the reasons discussed above. The utility is 100%, as expected.

3.4 Method 4: Giving consecutive frames to different threads

This method also involves multi-threading. Consecutive frames are processed by different threads. The number of threads used, say N, forms the parameter here. We take N consecutive frames to be processed and give them to N different threads.
As multiple number of frames are being processed parallelly instead of sequentially, the total processing time should decrease.
Moreover, since each frame is processed in a similar way as baseline, the output values are expected to be same as that of baseline. Hence the utility/accuracy with respect to baseline is expected to be 100 percent.

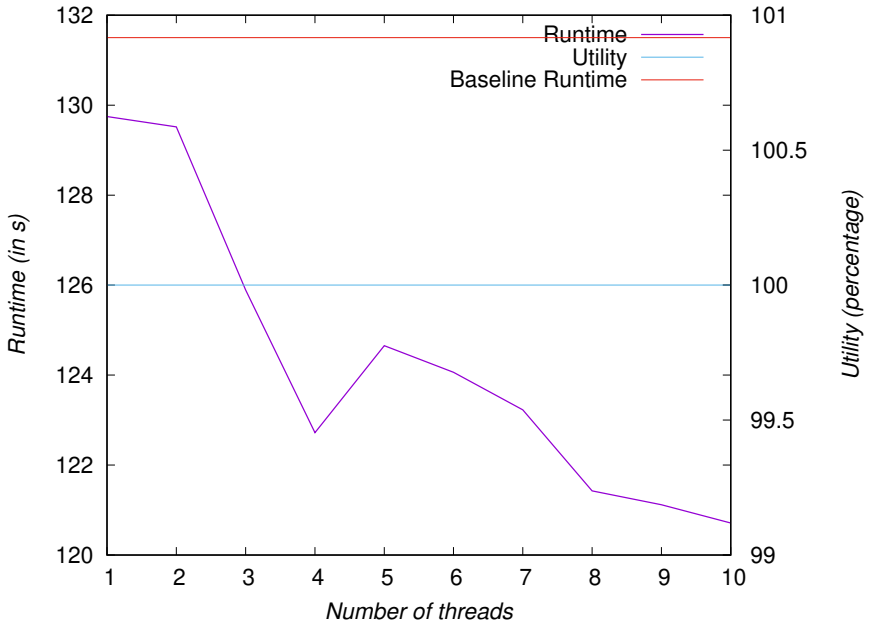

Figure 3.4: Runtime Utility Plot for Method 4

We observe that the runtime is roughly decreasing with increase in number of threads. The utility also comes to be 100 percent. Hence the plot for method 4 shows the expected behaviour.

Chapter 4

Results and Analyses

The plots of section 3.1 using the method of sub-sampling frames show an improvement in runtime with increase in number of frames skipped, but it involves a trade-off with the accuracy of output values. We can consider using this method during queue density estimation with a small value of the parameter, so as to obtain a better runtime without compromising much with the utility of output.

The plots of section 3.2 using the method of reducing resolution also show an improvement in runtime with a decrease in resolution (factor), but there is a corresponding decrease in accuracy of output due to loss of information while processing. For values of resolution factor other than 1, the utility attains almost a constant value, significantly lesser than the baseline. Hence this method though able to make the software time efficient, leads to poor utility.

The plot of section 3.3 shows that runtime is lesser than the baseline runtime for all number of threads greater than 2. The runtime does not decrease but fluctuates a little as the number of threads increase, which may be due to reasons as discussed in section 3.3. However, it is still better than the baseline runtime. Also, since all processing is similar to that for the baseline, we have 100 % utility which is the plus point for this method.

The plot of section 3.4 shows that there is a considerable decrease in runtime as compared to the baseline runtime as the number of threads increase. The runtime has decreased from 131,502 ms for the baseline to 120,711 ms for 10 threads, which is better as compared to values obtained in section 3.3. However, the decrease in runtime is not as large as that obtained via section 3.1. But since there is no loss of utility in this case, this can be considered as a good method for when lesser runtime is required with no compromise in utility.

Conclusion

We observe that method 1 and method 2, though decrease the runtime, also compromise with the utility of the output. On the other hand, method 3 and method 4, only help in making the code time-efficient, without any loss of utility. Hence, multi-threading proves to be quite useful in improving the performance without any utility compromise.