COP290 Report

Parth Gupta (2019CS10380)
Ananjan Nandi (2019CS10325)

March 2021

1 Metrics

We have used root mean squared error as utility metric. As error increases, utility decreases. RMS of n numbers $x_1, x_2, \ldots, x_n$ is defined as $\sqrt{\frac{x_1^2+x_2^2 \ldots +x_n^2}{n}}$. To find RMS error, deduct corresponding baseline value from the values before applying the formula. We have used the C++ inbuilt library std::chrono to calculate the time taken by the program to a high accuracy.

2 Input Coordinates

We wrote a C++ code to display an empty frame from the video(the first frame) and ask the user to click on 4 points counter-clockwise starting from top-right. These 4 points should enclose the region to be analyzed and will be used for angle correction in all methods and baseline. The points are written to cor.csv and this file is used in the rest of the project.

3 Baseline

We wrote a C++ code to process every frame of the video and generate the baseline value (queue density in each frame). We write the result to stationary.csv and use it throughout the project.

4 Queue Density and Dynamic Density

- Queue Density: We have used absdiff() between the current frame and a selected empty frame (empty2.png) to calculate queue density. After absdiff(), we perform thresholding and dilation on the resulting object to reduce noise. We finally calculate the ratio of number of non-zero pixels and total number of pixels in the frame. This gives us the queue density. This part is common across methods.

- Dynamic Density: We have implemented it using sparse and dense optical flow as mentioned in the extra credit section. More details are at the end of the report.

5 Methods

5.1 Method 1

We do sub-sampling in this method. We take a parameter x and we skip $x-1$ frames and process every x^{th} frame starting from the first one. We vary x from 1 to 8 and also 16 and 20 and plot the graphs from the values generated, which are written to m1.csv. Runtime is defined as the time taken by the program to execute for a given value of x. Error refers to the RMS value of the error taken across all the frames of the video. The error in a particular frame is the absolute difference in the queue density value of baseline and the queue density value of the frame. Utility is assumed to increase as this error decreases. This is the same for all methods. If the frame is being skipped, we use the queue density value of the last frame that was processed.

5.2 Method 2

We change the resolution(down-scaling) for the frames. We measure the error in queue density for different resolutions. The amount by which the width(x) and height(y) are scaled down is a parameter for this method. Rather than using 2 parameters, we use a single parameter defined as $x \times y$. It signifies the factor by which the total number of pixels decrease. We use the inbuilt function resize() to perform the change to each frame of the video while keeping aspect ratio same. The values of error and runtime are found for $(x, y) = (1, 1), (0.8, 0.8), (0, 7, 0, 7)$ and $(0.6, 0.6)$ and they are written to m2.csv.

5.3 Method 3

We use spatial allocation of work across threads. The pthread library is used. A struct pass is written to pass information to and get information from threads. The parameter x is the number of threads. The current frame is passed through warpPerspective() and split horizontally and equally into x pieces using crop(). Using the struct, the piece and piece number is passed to the corresponding thread to be processed. The thread then takes the absdiff of the piece with the corresponding piece of the reference frame (empty) and returns the number of non-zero pixels as an argument of the struct. When all the threads exit, the total number of non-zero pixels is found from the structs returned by the threads and used to calculate the error. The next frame is then processed. We found values for 1, 2, 3, 4, 5, 6, 7, 8, 16 and 20 threads and stored them in m3.csv.

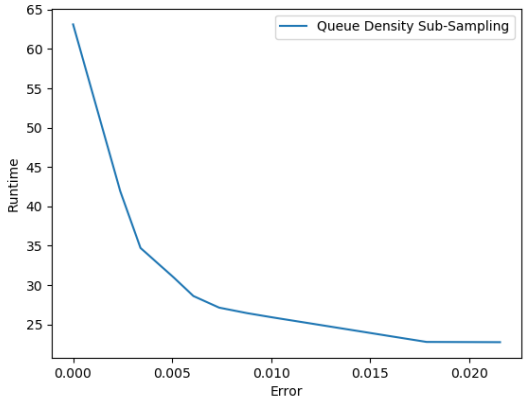
5.4 Method 4

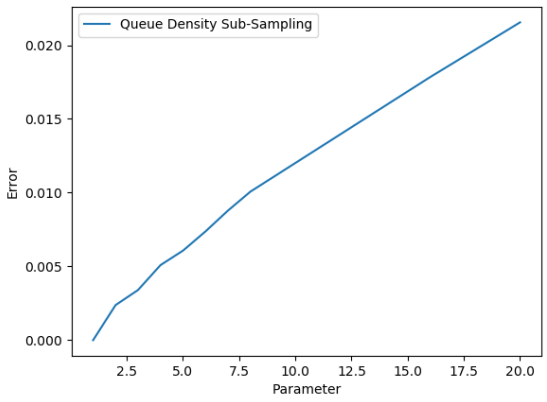
We use temporal allocation of work across threads. The pthread library is used. Another struct pass is written to pass information to and get information from threads. The parameter x is the number of threads. We read chunks of x frames at a time and pass each frame to a different thread, along with the frame number, using the struct. The thread then processes the frame using warpPerspective() and absdiff() and returns the value of queue density for the frame in the struct. When all active threads exit, the next x frames are processed. We found values for 1, 2, 3, 5, 6, 10, 12, 16 and 20 threads and stored them in m4.csv.

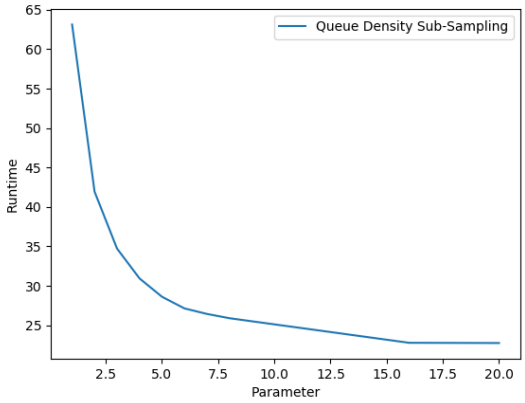
6 Trade-Off Analysis

6.1 Method 1

Error	Runtime	Parameter
2.59259e-07	63.1311	1
0.0023789	41.943	2
0.0034022	34.724	3
0.0050906	30.945	4
0.0060706	28.617	5
0.0073711	27.14	6
0.0087831	26.441	7
0.01007	25.9	8
0.017838	22.782	16
0.021547	22.753	20





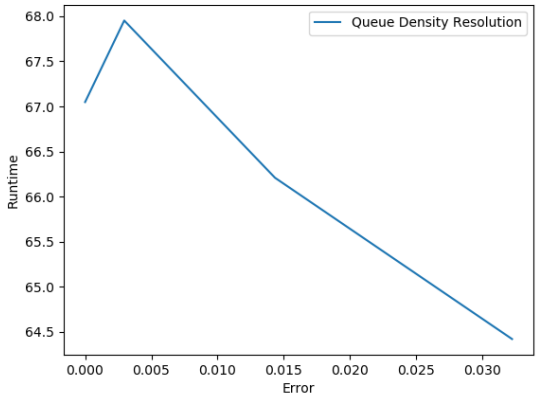


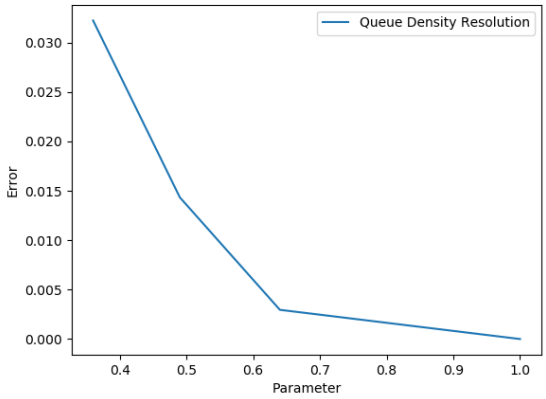
As we increase the number of frames to be processed, we will know the accurate value of queue density for a higher number of frames, thus reducing the number of frames in which we extrapolate data. This in turn, reduces the overall utility/error. Increasing the number of frames to be processed means decreasing the value of x. Also, we do not need any extra processing to decrease the number of frames being processed so the runtime decreases if we increase the value of x. So the overall effect is the following:

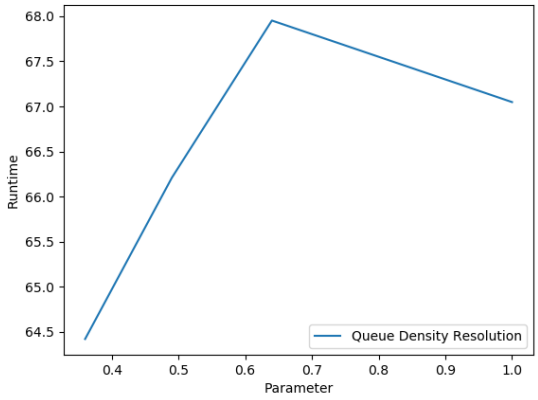
- If x(parameter) increases, error increases.
- If x(parameter) increases, runtime decreases.
- If error increases, runtime decreases

6.2 Method 2

Error	Runtime	Parameter
2.58886e-07	67.0481	1
0.0029545	67.952	0.64
0.014335	66.209	0.49
0.032226	64.422	0.36





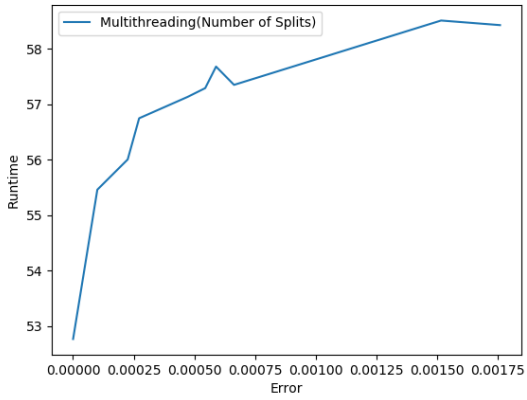


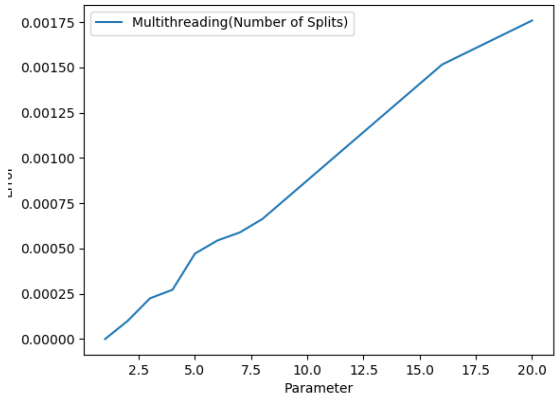
In general, a lower resolution frame should be easier to perform operations on, and hence, lower resolution should lead to decrease in runtime. However, as resolution decreases, we lose data in the frame matrix, and hence the error should increase. The results obtained do not completely agree the above logic. This is because when the parameter is $1(x = 1$ and $y = 1)$, the system doesn't need to perform any operation in resize as it is already in the required resolution. Hence, the runtime is low when the parameter is 1. However, the error only depends on resolution and increases if resolution is decreased. Note that here increasing the parameter means increasing the resolution. The results we obtain are as follows:

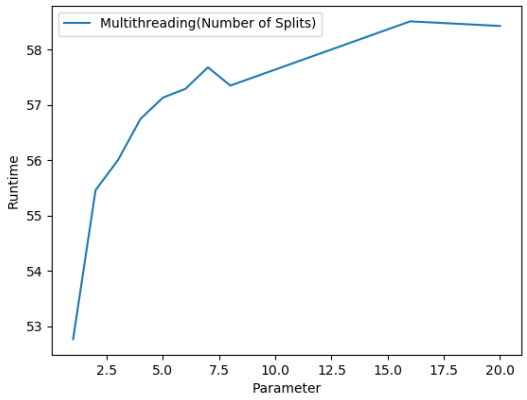
- If $x \times y$(parameter) increases, error decreases.
- If $x \times y$(parameter) increases, runtime increases most of the time.
- If error increases, runtime decreases most of the time.

6.3 Method 3

Error	Runtime	Parameter
2.59349e-07	52.763	1
0.00010007	55.46	2
0.00022514	56.006	3
0.00027226	56.75	4
0.0004719	57.136	5
0.00054426	57.294	6
0.00058892	57.683	7
0.00066296	57.353	8
0.0015156	58.515	16
0.0017584	58.432	20







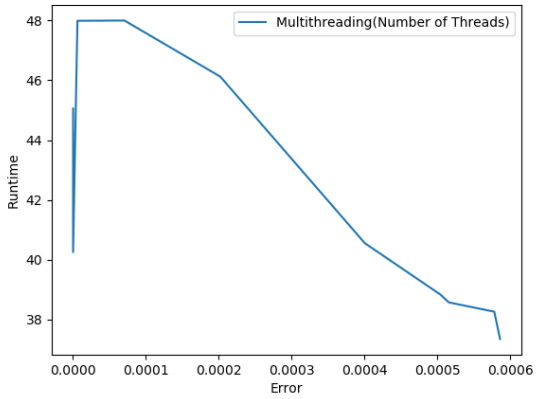
As the number of threads increases, runtime should have decreased. This is because more threads allow more efficient usage of the system cores. However,

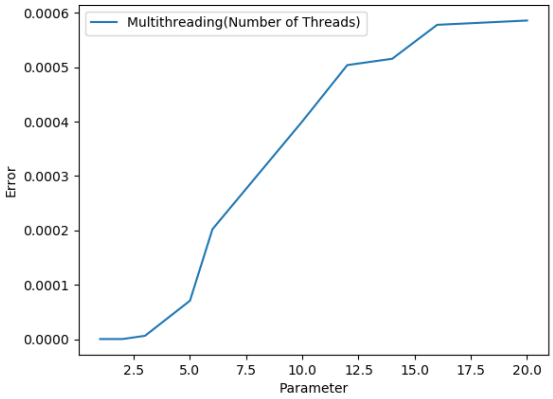
we see that error increases and runtime also increases as number of threads is increased. This is because generating the splits of the frame and creating and destroying the threads take more time than the speedup in the absdiff() operations due to threading. Also, the warpPerspective() operation has not been included in the process for the thread. Therefore, the potential speedup is reduced. Running htop while the program was executing, we have confirmed that CPU cores are being better utilized when multithreading is being used. Also, the runtime for all parameters in this method is better than the runtime without threading. The increasing runtime is therefore probably due to the cost of generating splits to be processed. Accuracy decreases as threads are increased due to the loss of accuracy in the process of generating pieces and processing them individually using dilate and threshold instead of together. So the overall effect is the following:

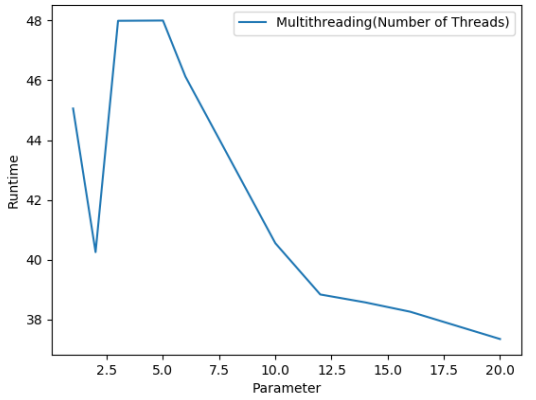
- If x(parameter) increases, error increases.
- If x(parameter) increases, runtime increases.
- If error increases, runtime increases.

6.4 Method 4

Error	Runtime	Parameter
2.59349e-07	45.0573	1
2.5935e-07	40.256	2
6.2138e-06	47.989	3
7.0714e-05	47.998	5
0.00020216	46.122	6
0.00040042	40.554	10
0.00050387	38.84	12
0.0005155	38.576	14
0.00057791	38.264	16
0.00058581	37.351	20







As the number of threads increases, runtime should have decreased. This is because more threads allow more efficient usage of the system cores. In this case, we observe this trend starting from 5 threads. Before that we observe some peculiar behavior. The runtime sharply decreases from 1 thread to 2 threads, and then increases again for 3 threads. From there, it is nearly constant till 5 threads. To explain this, we used htop. While generating baseline, one core showed 100% use while other cores varied from 20% to 30%. In comparison, for 2 threads, every core showed 60% to 70% use, which is significantly higher. For 3 threads, every core showed 40% to 50% use. This percentage then gradually increases as number of threads are increased. The OS seems to assign work much more efficiently when 2 threads are used. This explains the behavior of the runtime as CPU core utilization is directly connected to runtime. As number of threads increases, error slightly increases. This may be due to the threading process causing some inaccuracies in the returned values. So the overall effect is the following:

- If x(parameter) increases, error increases.
- If x(parameter) increases, runtime decreases in general.
- If error increases, runtime decreases in general.

7 Dense and Sparse Optical Flow

- Dense Optical Flow: OpenCV implements this using Gunner Farneback's algorithm. It calculates the optical flow for all the points in the frame. Thus, it is time consuming but the result obtained is quite accurate compared to Sparse Optical Flow algorithm.
  Time taken: 327.143 seconds
- Sparse Optical Flow: OpenCV uses Lucas Kanade algorithm. It takes input as a set of points to track, the previous frame and the current frame. It returns the future positions of the same set of points along with their status value(1 if found, 0 otherwise). We find whether a point has moved by taking the norm with the original point set and thresholding it. The set of points to track is decided by the function goodFeaturesToTrack(). This uses Shi-Tomasi algorithm to detect the corner points to analyze. We pass the first frame as argument to it and it generates a list of points to be tracked by Lucas Kanade algorithm. It is clear that the optical flow is computed for smaller number of points and hence it takes less time. However, since it doesn't compute optical flow for all the points, it gives error when compared with dense optical flow algorithm. We perform both dense and sparse optical flow and note the error in dynamic density of sparse flow by taking rms value over all the frames.
  Time taken: 81.8202 seconds
  Error: 0.0871846