Faculty of Engineering and Technology

Electrical and Computer Engineering Department

MULTIMEDIA NETWORKING

ENCS5398

(Task #3)

"Use FFmpeg to compare between different codec standards and tune some codec parameters**"**

Instructor: Mohammad Jubran

Prepared By:

Name: Jad Samara **ID:** 1170685

Date: 24-11-2021

Table of Contents

Performance for different video codec standards

HEVC/H.265 Tuning

Performance for different video codec standards - AVC/H.264, HEVC/H.265, and AV1

In this task, we are going to use FFMPEG to compare the performance of the three video codec standards - AVC/H.264, HEVC/H.265, and AV1.

Generating the commands

I used python to make the process a bit faster.

AVC/H.264

H. 264 or MPEG-4 AVC (Advanced Video Coding) is a video coding format for recording and distributing full HD video and audio. The purpose of creating H.264/AVC was to pioneer a new digital video standard capable of delivering good video quality at a substantially lower bitrate than previous standards.

Constant rate factor (CRF) is an encoding mode that adjusts the file data rate up or down to achieve a selected quality level rather than a specific data rate.

The range of the CRF scale is 0–51, where 0 is lossless, 23 is the default, and 51 is worst quality possible.

The Python code to generate commands is shown below:

```
for i in [3, 9, 15, 21, 27, 33, 39, 45, 51]:
      print('ffmpeg -i input.mp4 -c:v libx264 -crf '+str(i)+' 264_crf_'+str(i)+'.mp4 -
      psnr &>> 264.txt')
```

Outputs: (AVC/H.264 codec commands)

```
ffmpeg -i input.mp4 -c:v libx264 -crf 3 264_crf_3.mp4 -psnr &>> 264.txt
ffmpeg -i input.mp4 -c:v libx264 -crf 9 264_crf_9.mp4 -psnr &>> 264.txt
ffmpeg -i input.mp4 -c:v libx264 -crf 15 264_crf_15.mp4 -psnr &>> 264.txt
ffmpeg -i input.mp4 -c:v libx264 -crf 21 264_crf_21.mp4 -psnr &>> 264.txt
ffmpeg -i input.mp4 -c:v libx264 -crf 27 264_crf_27.mp4 -psnr &>> 264.txt
ffmpeg -i input.mp4 -c:v libx264 -crf 33 264_crf_33.mp4 -psnr &>> 264.txt
ffmpeg -i input.mp4 -c:v libx264 -crf 39 264_crf_39.mp4 -psnr &>> 264.txt
ffmpeg -i input.mp4 -c:v libx264 -crf 45 264_crf_45.mp4 -psnr &>> 264.txt
ffmpeg -i input.mp4 -c:v libx264 -crf 51 264_crf_51.mp4 -psnr &>> 264.txt
```

the part: `-c:v libx264` denotes the codec type, and the `-crf 39` part denotes the CRF value of the compressed video. `&>> 264.txt` this appends the output to a txt file.

HEVC/H.265

High Efficiency Video Coding, also known as H.265 and MPEG-H Part 2, is a video compression standard designed as part of the MPEG-H. The HEVC compression standard was developed to double the compression efficiency of the previous standard, H.264/AVC, and at typical video distribution bitrates.

Python code to generate the HEVC/H.265 commands is shown below:

```
for i in [3, 9, 15, 21, 27, 33, 39, 45, 51]:
      print('ffmpeg -i input.mp4 -c:v libx265 -crf ' + str(i) + ' 265_crf_' + str(i) +
      '.mp4 -psnr &>> 265.txt')
```

Outputs: (HEVC/H.265 codec commands)

```
ffmpeg -i input.mp4 -c:v libx265 -crf 3 265_crf_3.mp4 -psnr &>> 265.txt
ffmpeg -i input.mp4 -c:v libx265 -crf 9 265_crf_9.mp4 -psnr &>> 265.txt
ffmpeg -i input.mp4 -c:v libx265 -crf 15 265_crf_15.mp4 -psnr &>> 265.txt
ffmpeg -i input.mp4 -c:v libx265 -crf 21 265_crf_21.mp4 -psnr &>> 265.txt
ffmpeg -i input.mp4 -c:v libx265 -crf 27 265_crf_27.mp4 -psnr &>> 265.txt
ffmpeg -i input.mp4 -c:v libx265 -crf 33 265_crf_33.mp4 -psnr &>> 265.txt
ffmpeg -i input.mp4 -c:v libx265 -crf 39 265_crf_39.mp4 -psnr &>> 265.txt
ffmpeg -i input.mp4 -c:v libx265 -crf 45 265_crf_45.mp4 -psnr &>> 265.txt
ffmpeg -i input.mp4 -c:v libx265 -crf 51 265_crf_51.mp4 -psnr &>> 265.txt
```

AV1

AOMedia Video 1 is an open, royalty-free video coding format initially designed for video transmissions over the Internet. AV1 codec is 30% better than H. 265. Besides being royalty-free and open-source.

Python code to generate the AV1 commands:

```
for i in [3, 9, 15, 21, 27, 33, 39, 45, 51]:
      print('ffmpeg -i input.mp4 -c:v libx264 -crf '+str(i)+' 264_crf_'+str(i)+'.mp4 -
      psnr &>> 264.txt')
```

Outputs: (AV1 codec commands)

```
./ffmpeg -i input.mp4 -c:v libaom-av1 -crf 3 -cpu-used 8 av1_crf_3.mkv -psnr
./ffmpeg -i input.mp4 -c:v libaom-av1 -crf 9 -cpu-used 8 av1_crf_9.mkv -psnr
./ffmpeg -i input.mp4 -c:v libaom-av1 -crf 15 -cpu-used 8 av1_crf_15.mkv -psnr
./ffmpeg -i input.mp4 -c:v libaom-av1 -crf 21 -cpu-used 8 av1_crf_21.mkv -psnr
./ffmpeg -i input.mp4 -c:v libaom-av1 -crf 27 -cpu-used 8 av1_crf_27.mkv -psnr
./ffmpeg -i input.mp4 -c:v libaom-av1 -crf 33 -cpu-used 8 av1_crf_33.mkv -psnr
./ffmpeg -i input.mp4 -c:v libaom-av1 -crf 39 -cpu-used 8 av1_crf_39.mkv -psnr
./ffmpeg -i input.mp4 -c:v libaom-av1 -crf 45 -cpu-used 8 av1_crf_45.mkv -psnr
./ffmpeg -i input.mp4 -c:v libaom-av1 -crf 51 -cpu-used 8 av1_crf_51.mkv -psnr
```

The command starts with: `./` since I switched to windows power shell.

Collecting the Data – PSNR, and Rate

After applying the commands, and generating the different codec videos and saving the results in their respectable files, the time is now for collecting the important data from the output files.

The video name: (it contains the CRF value used)

```
…
Output #0, mp4, to '264_crf_3.mp4':
…
```

The video rate:

```
…
frame=  726 fps= 99 q=-1.0 LPSNR=Y:inf U:inf V:inf *:inf size=   67962kB time=00:00:29.04
bitrate=19166.2kbits/s speed=3.97x
…
```

And finally, the video PSNR value:

```
…
[libx264 @ 0x7fffe92e5b40] PSNR Mean Y:55.739 U:59.367 V:60.363 Avg:56.721 Global:56.554
kb/s:19035.70
…
```

This is basically the process for collecting the data for the 3 video codecs, and since for each codec I made 9 videos using 9 different CRF values, collecting the data will become a bit tedious.

To solve that problem, I made another python program to collect the data for me.

```
# collect the 264 codec videos data
with open ('text', 'rt') as myfile:
    for myline in myfile:
        if myline.__contains__("to '264"):
            print("crf: "+myline[28:30])
        if myline.__contains__(" kb/s:"):
            print("Rate: "+myline[94:100])
            print("PSNR: " + myline[39:45])
            print()
```

It outputs the data in a good visual way as seen bellow.

```
crf: 3
Rate: 241601
PSNR: 55.660
…
crf: 51
Rate:  613.6
PSNR: 31.052
```

and the same is done for each codec.

Quality Comparison

Quality comparison based on CRF:

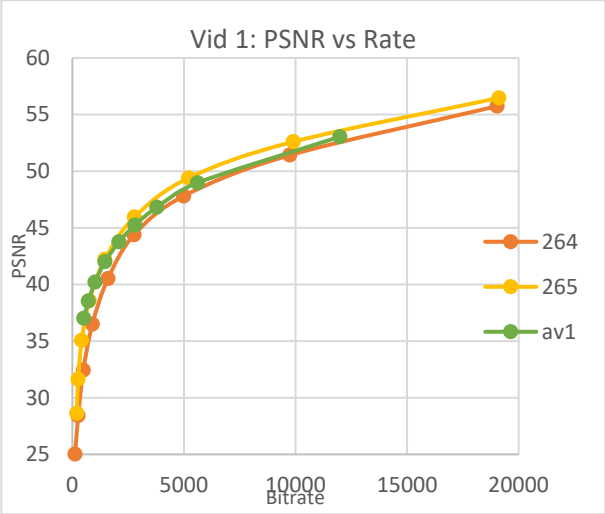
AVC/H.264 – CRF 51	HEVC/H.265 – CRF 51	AV1 – CRF 51
1.32 MB	0.507 MB	4.70 MB

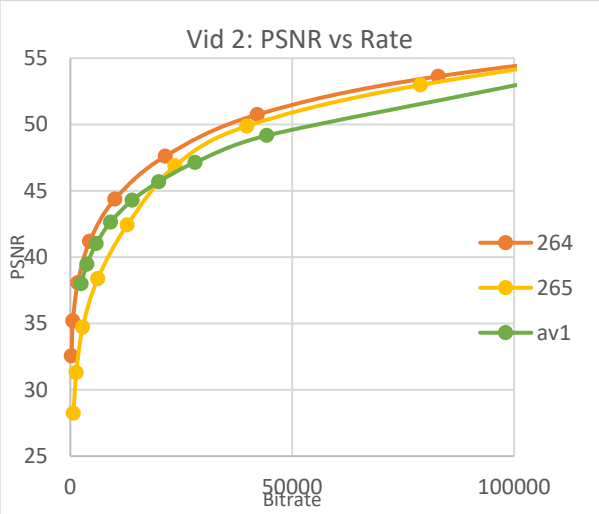
Quality comparison based on Size: (Closest sizes)

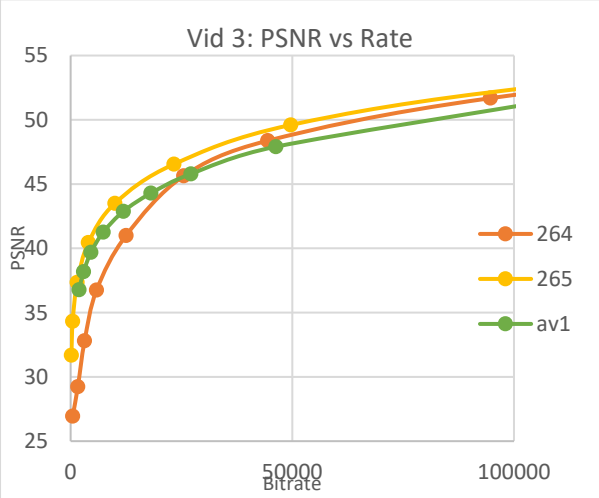
AVC/H.264 – CRF 39	HEVC/H.265 – CRF 39	AV1 – CRF 51
5.41 MB	3.24 MB	4.70 MB

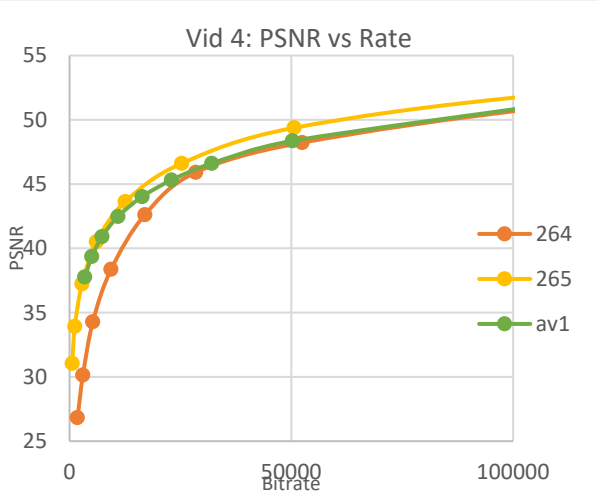
Rate-PSNR Curves for the Different Codecs

Below are the Rate-PSNR curves for 4 different videos.

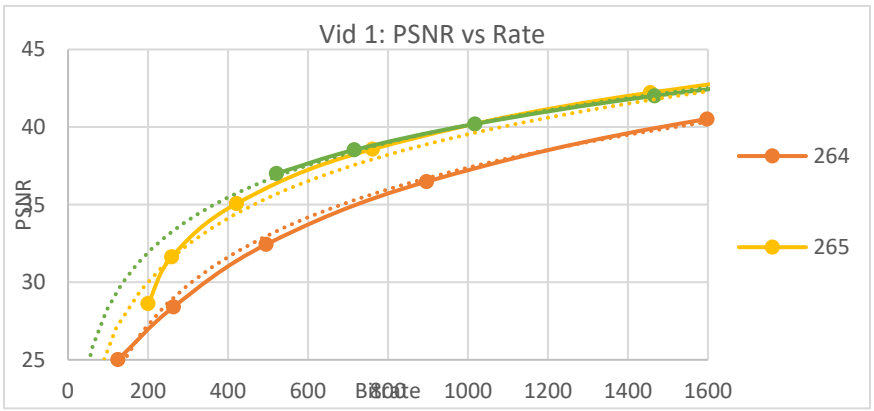




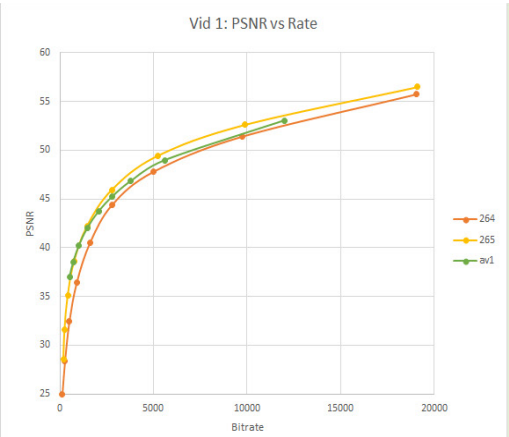


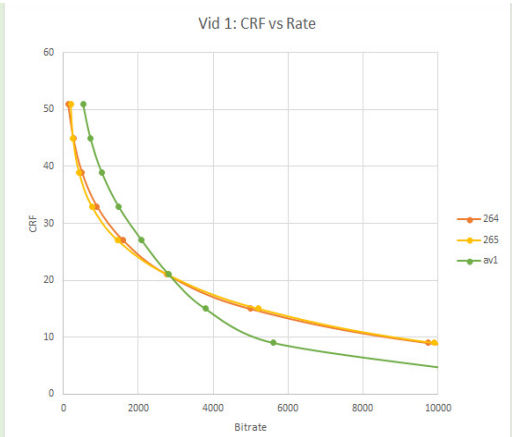


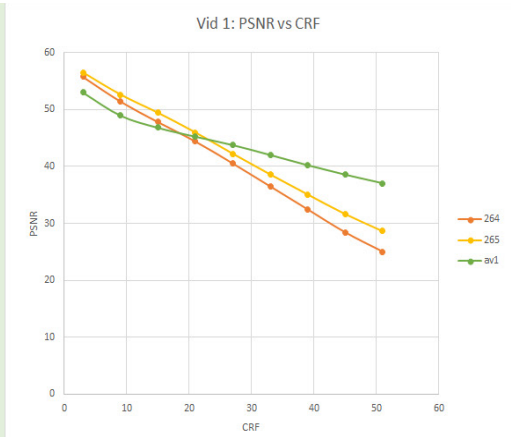
When I checked the videos, I expected the AV1 to have the best Rate-PSNR Curve due to its better visual quality, followed by HEVC/H.265 and finally, AVC/H.264. but the curves weren't as I expected. The HEVC/H.265 was better 3 out of 4 times. The AVC/H.264 was better 1 out of 4 times. The AV1 was better when the bitrate was low, the curve below shows the result.

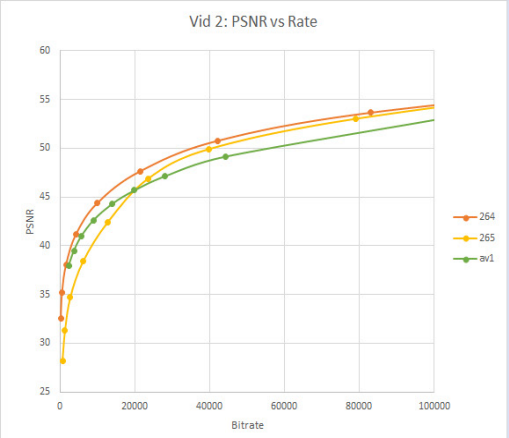


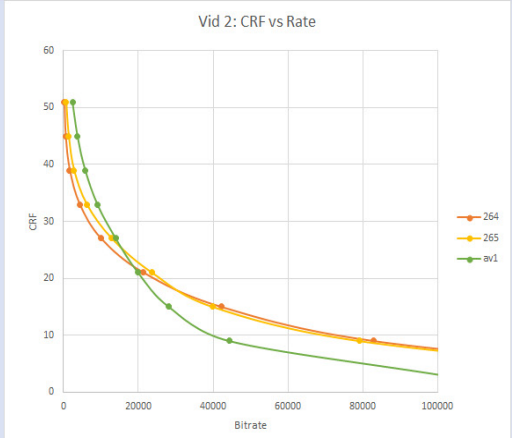
Bellow are different curves.

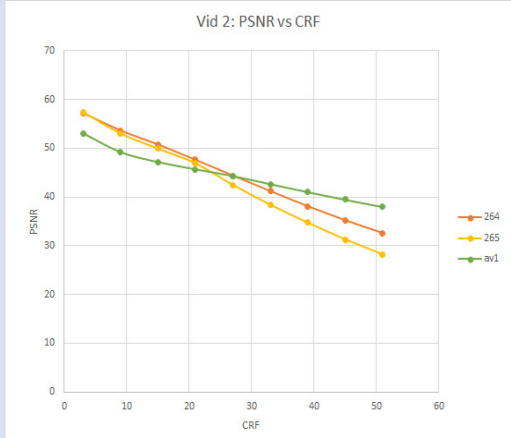


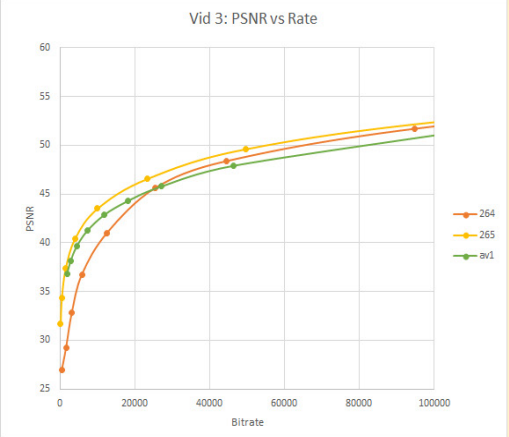


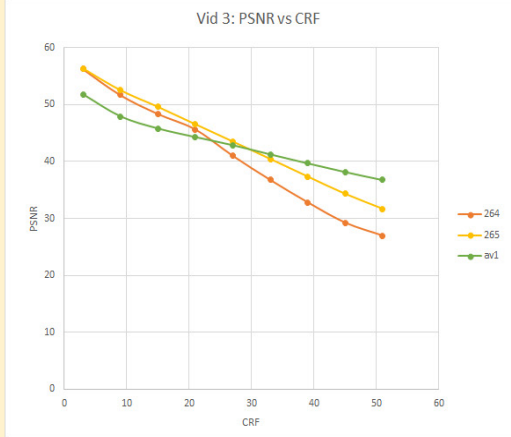


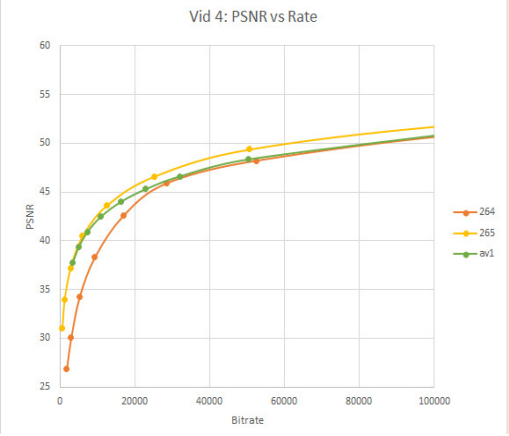


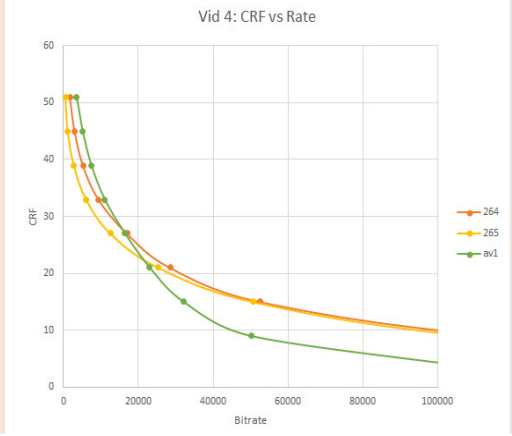












HEVC/H.265 transcoding

using different CU sizes, Partitioning depths, and Motion estimation search ranges

In this task, we are going to use FFMPEG to transcode HEVC/H.265 and tune some of its codec parameters like CU size, partitioning depth, and Motion estimation range.

The tuning parameters for the HEVC/H.265 codec

Below are the different parameters -Maximum CU size, Maximum partitioning depth, and Motion estimation search range- that I am going to tune in this task.

Maximum CU size

Maximum CU size (width and height). The larger the maximum CU size, the more efficiently x265 can encode flat areas of the picture, giving large reductions in bitrate. However, this comes at a loss of parallelism with fewer rows of CUs that can be encoded in parallel, and less frame parallelism as well. Because of this the faster presets use a CU size of 32. Default: 64

```
--ctu, -s <64|32|16>
```

Maximum partitioning depth

The transform unit (residual) quad-tree begins with the same depth as the coding unit quad-tree, but the encoder may decide to further split the transform unit tree if it improves compression efficiency. This setting limits the number of extra recursion depth which can be attempted for intra coded units. Default: 1, which means the residual quad-tree is always at the same depth as the coded unit quad-tree

```
--tu-intra-depth <1..4>
--tu-inter-depth <1..4>
```

Motion estimation search range

Motion search range. Default 57

The default is derived from the default CTU size (64) minus the luma interpolation half-length (4) minus maximum supple distance (2) minus one extra pixel just in case the hex search method is used. If the search range were any larger than this, another CTU row of latency would be required for reference frames. The Range of values is an integer from 0 to 32768

```
--merange <integer>
```

Generating, running the commands and collecting the results

I used python to streamline the process of trying out different tuning parameters.

Since I am going to change the parameters of the HEVC/H.265, and for every change measure the data for a set of CRF values, running the commands manually will prove to be tedious.

Program to vary Maximum CU size

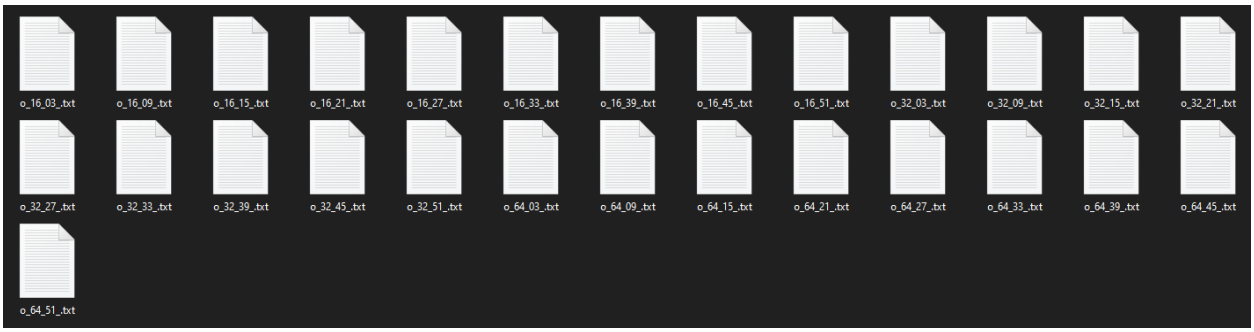
The program below generates the commands for changing the CU size, and then runs them and stores the command's output.

```
import subprocess
# Vary Maximum CU size
for j in [16, 32, 64]:
    for i in [3, 9, 15, 21, 27, 33, 39, 45, 51]:

        print('ffmpeg.exe -benchmark -i input.mp4 -c:v libx265 -crf ' + str(i) + ' -x265-
                  params ctu='+str(j)+' ./videos/265_cu_' + str(j) + '_crf_' + str(i) +
                  '.mp4 -psnr > ./data/o_'+ str(j) +'_'+ str(i) +'_.txt 2>&1')

        p = subprocess.Popen('ffmpeg.exe -benchmark -i input.mp4 -c:v libx265 -crf ' + str(i)
                  + ' -x265-params ctu='+str(j)+' ./videos/265_cu_' + str(j) + '_crf_' +
                  str(i) +'.mp4 -psnr > ./data/o_'+ str(j) +'_'+ str(i) +'_.txt 2>&1',
                  shell=True, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
        p.stdout.readlines()
```

the '`subprocess`' import is what is going to allow us to run the commands, the print prints the current command, and finally, '`p = subprocess.Popen`' runs the current command. Running the program generates the following output files:



The name of each file tells the CU value with the CRF values.

Program to convert the output into .CSV

Even though we have all the encoded videos results, we still need to sort them and save them in a way to allow us to understand them and use them to make meaningful graphs.

The program is as follows:

```
import os

print('element,rate,size,time,PSNR')

for filename in os.listdir(os.getcwd()+'\\sorted data\\z_'):
    with open(os.path.join(os.getcwd()+'\\sorted data\\z_', filename), 'r') as f:  #
open in readonly mode
        for myline in f:     # For each line, read to a string,
            if myline.__contains__(", mp4, to '"):
                k = myline.find('.mp4\':')
                print(myline[k - 2:k].replace('_', ''), end=',')

            if myline.__contains__("rtime="):
                i = myline.find("rtime=")
                print(myline[i + 6:i + 7 + 5].replace('s', ''), end=',')

            if myline.__contains__("video:"):
                print(myline[0 + 6:0 + 7 + 5].replace('k', '').replace(' ',
                                ''').replace('B', ''), end=',')

            if myline.__contains__("LPSNR="):
                i = myline.find("bitrate=")
                print(myline[i + 8:i + 9 + 6].replace(' ', '').replace('k', ''),
                                end=',')

            if myline.__contains__("Global PSNR: "):
                j = myline.find("PSNR: ")
                print(myline[j + 6:j + 7 + 5])
```

we start by importing the '`os`' import that will allow to open system files, and in our case the output files, this is followed by a print statement that prints the CSV header for the data.

The file opened is called '`z_`' and this file has the output files of each parameter.

Each if statement looks for a certain value, the first looks for the CRF value, the second looks for the actual run time of the command (actual encoding time), the third looks for the video size, the fourth looks for the video rate, and finally, the fifth looks for the PSNR value.

Using the same method, I created 2 more programs to Vary Maximum partitioning depth and Vary Motion estimation search range.

Running the program on the CU value of 16 gives the following result:

```
element,rate,size,time,PSNR
3,19103.8,67257,30.878,56.461
9,9902.9,34631,24.537,52.594
15,5216.8,18015,19.705,49.391
21,2791.6,9415,15.852,45.941
27,1461.7,4700,12.365,42.205
33,768.3,2241,9.600,38.542
39,428.5,1036,8.013,34.987
45,267.8,466,6.325,31.530
51,207.8,254,5.521,28.523
```

This then can be saved into a CSV file that looks like this:

	A	B	C	D	E	F	G	H	I
1	element	rate	size	time	PSNR				
2	3	19103.8	67257	30.878	56.461				
3	9	9902.9	34631	24.537	52.594				
4	15	5216.8	18015	19.705	49.391				
5	21	2791.6	9415	15.852	45.941				
6	27	1461.7	4700	12.365	42.205				
7	33	768.3	2241	9.6	38.542				
8	39	428.5	1036	8.013	34.987				
9	45	267.8	466	6.325	31.53				
10	51	207.8	254	5.521	28.523				

And now we can use Excel to make meaningful graphs using the collected data.

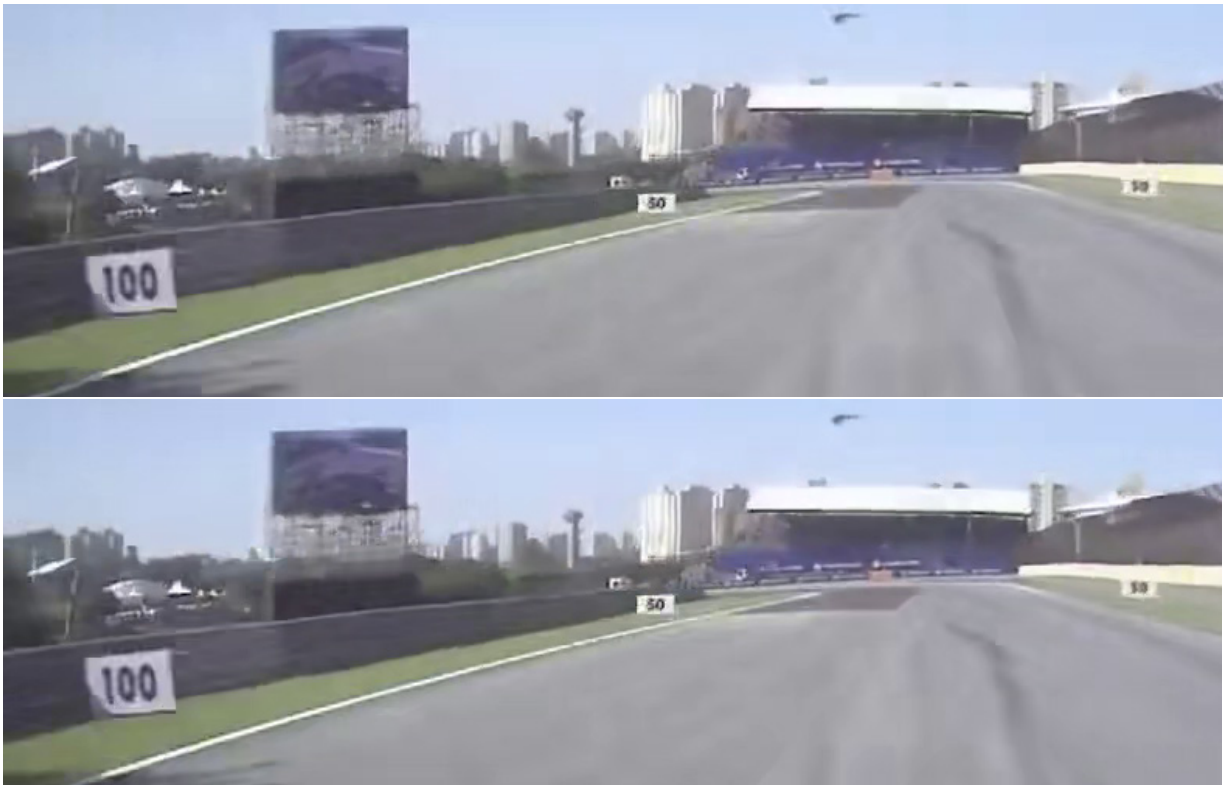
Quality Comparison

Following are visual video comparisons for tuning the different variables.

Varying Maximum CU Size

CU=16 CU=32 CU=64 CRF=51

Vary Maximum partitioning depth



TU-Inter-Intra-Depth=1 (above) TU-Inter-Intra-Depth=4 (below)

Vary Motion estimation search range

The writing is the most notable difference.

Motion Estimation Range=64 CRF=45

Motion Estimation Range=256 CRF=45

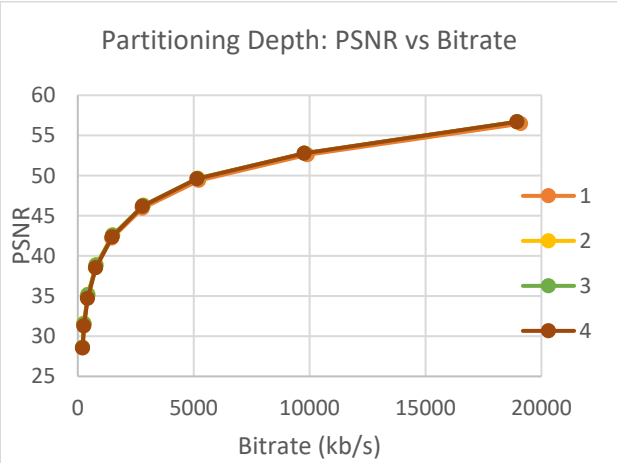
Motion Estimation Range=16384 CRF=45

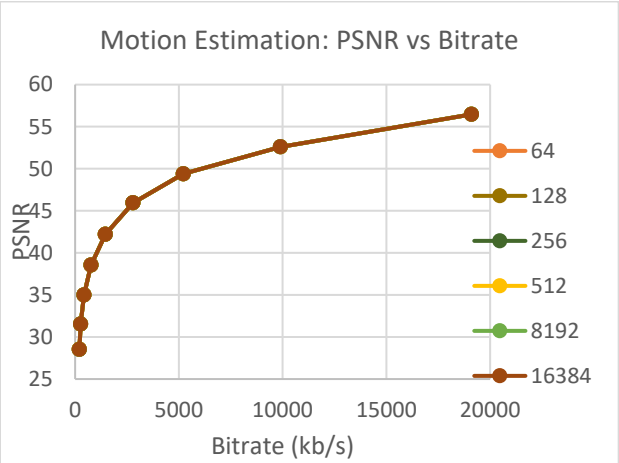
The results

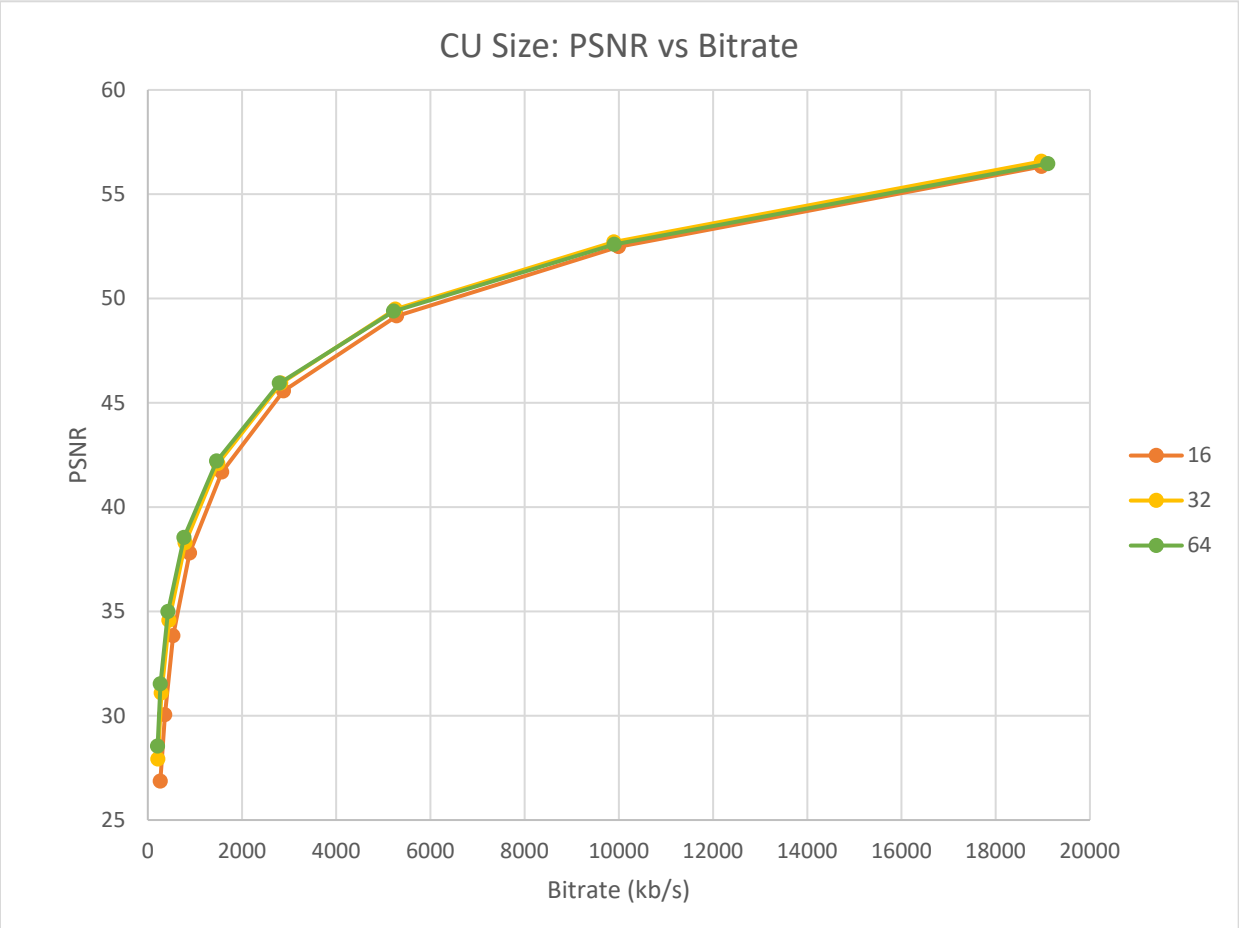
The results are shown as different graphs, focusing on PSNR, Bitrate, Encoding Time and Final Size.

PSNR vs Bitrate

The resulting curves have minimal change in towards an increase results a better PSNR. With CU Size having the biggest effect.

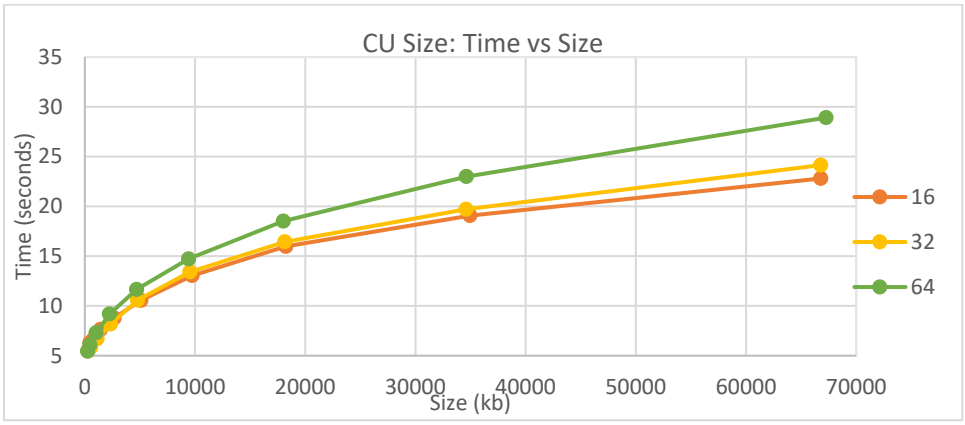




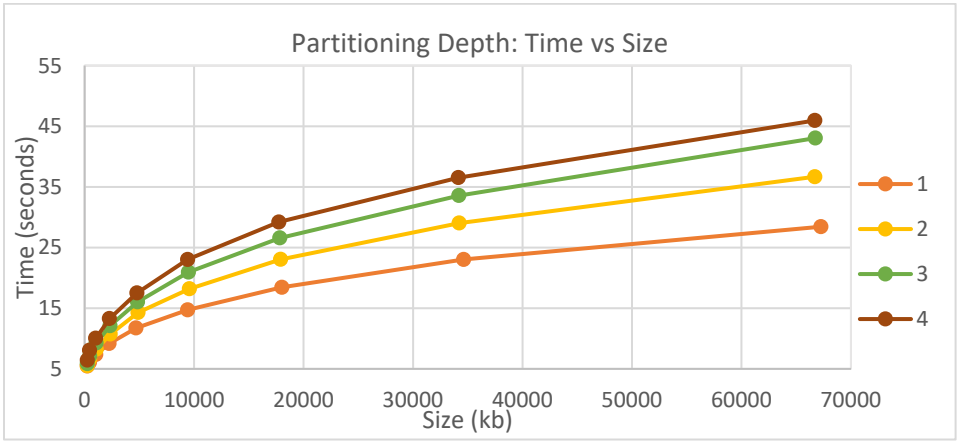


Time vs Size

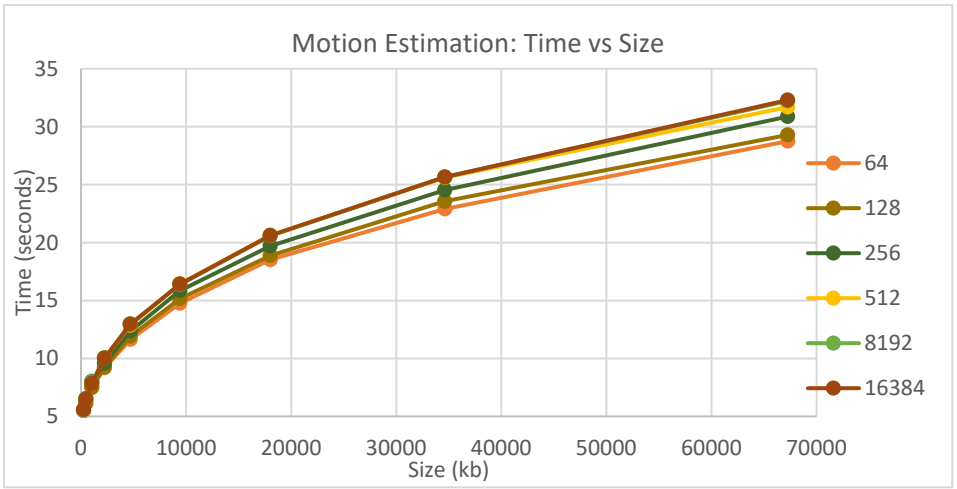
For each parameter, the Time vs Size curves are drawn below:



For the CU size, increasing the CU value results in encoding time increase.



Also, increasing the Partitioning Depth results in more encoding time.



For the Motion estimation, the same happens, but after increasing the value more than 1000, the curve tends to remain the same and have minimal change.

Time vs Bitrate

The time vs Bitrate curves followed a similar curve to the time vs Size curves.