x264

Algorithm Overview

h264 is a popular video encoding format for high quality and well compressed videos. *x264* is the most widely used algorithm for creating *h264* videos. Veetle uses VLC in its broadcasting software, which in turn uses x264 for transcoding.

All modern video files are not simply a series of images, as in an animated GIF. If they were, the file size would be huge. Instead, video files use motion compensation to keep file sizes low. Consider a video of a basketball game. In one scene, we see the ball move across the screen. First we store a complete picture of the ball. In h264, this would be an I frame. In the next milliseconds of the scene, instead of redrawing the whole ball in a new position, we can more succinctly say that the ball has moved 1 inch to the right. These instructions are called P and B frames in h264. Let's say the scene completely changes to a commercial break. P and B frames would struggle to describe the beginning of the commercial as differences from the scene with the basketball. So the the h264 video will most likely just cut the flow with another I frame.

I frames take up much more space than P and B frames. The key to a good encoding is to minimize the use of I frames and maximize the use of P and B frames. Since P and B frames contain far less information than I frames, x264 needs to use clever algorithms to make the P and B frames describe as much info as they can. These algorithms are directly affected by the x264 settings we describe below. In most cases, boosting these settings will require more CPU processing power. You'll need a fast multi-core processor to squeeze out better quality from x264. Exceeding the limit of your processor results in choppy streams.

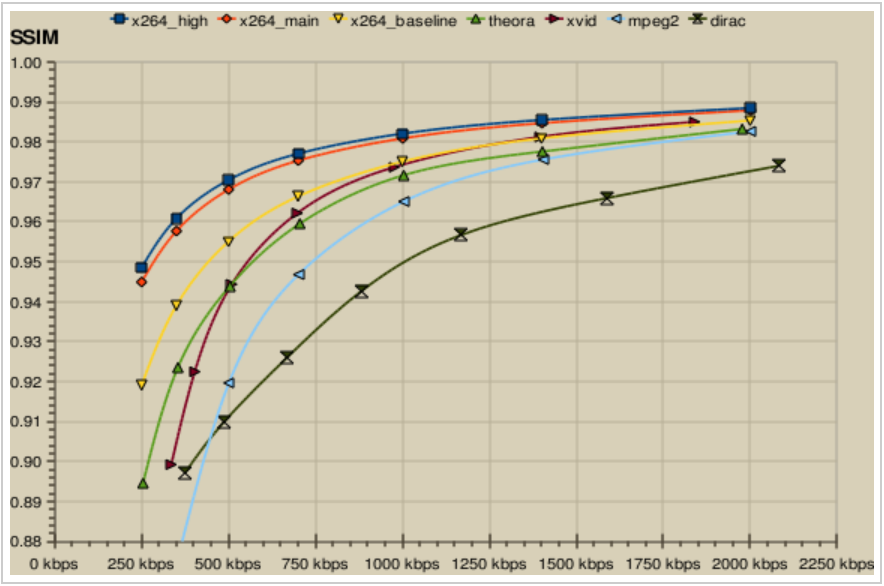
x264 tweaking benefits low bitrate streams

When broadcasting at high bitrates (over 1000 kbps), you may hardly notice changes in your stream quality as you change your x264 settings. Take a look at the graph to the right showing SSIM (a measurement of fidelity to the original source video) over three x264 profiles. Profiles are basically different sets of x264 settings, with the *high profile* being settings pushed quite high. When bitrate exceeds 1000 kbps, you'll notice that all three x264 settings yield more or less the same SSIM. Looking at the lower end of the chart, we see x264 settings greatly affecting SSIM.

When doing offline pre-transcoding, you should always use high x264 settings because you will usually have no time constraints and no concern for CPU usage. You should try to squeeze as much quality out of the video no matter if you're transcoding to a low or high bitrate.

In realtime transcoding, you won't be able to max out the x264 settings haphazardly. A smooth realtime transcode requires you to keep your CPU usage in check. When transcoding to low bitrate / resolution, your CPU can afford to use higher x264 settings. However, if you're transcoding to high bitrate/resolution, you will need to lower x264 settings to keep your CPU from burning up. At this end of the spectrum, x264 settings hardly affect quality, so it doesn't hurt to lower the settings. The only way to use both high bitrate/resolution and high x264 settings for maximum quality is to have an extremely powerful multi-core CPU.

The most important settings



subme

Subme stands for sub-pixel motion estimation. When P and B frames describe changes from a previous frame, they can say something like "the actor moved his hand *10* pixels up". They can get even more precise by saying "the actor moved his hand *10.4* pixels up". Subme deals with these fractional changes in movement. Higher numbers increase edge crispness.

- range: 1-9
- default: 5
- suggested: 7 (realtime), 9 (pre-transcode)

ref

Ref stands for reference frames. This is the number of previous frames P and B frames can refer to. More frames being considered means better quality to compression ratio. Be careful not to increase this

number too much. People watching your stream with weak computers may struggle to smoothly playback streams with high ref. 3 to 5 ref should be good. 6 and beyond don't offer much for their processing overhead. Some people report that PSNR quality even starts to decrease at 12.

- range: 1-16
- default: 3
- suggested: 3 (realtime), 4-5 (pre-transcode)

bframes

Bframes sets the maximum number of consecutive B frames. B frames are much smaller than I and P frames, yet provide a surprising amount of detail. When b-adapt = 1 (it is by default), bframes should be set to 16. Large bframe counts do not burdern the CPU in this mode. x264 will almost never put 16 bframes in a row, but at least it will have the opportunity to do so if it feels like it will lead to the better compression. When b-adapt = 2, bframes should be between 3 to 5. B-adapt = 1 should be used for live content, while b-adapt = 2 should be used for pre-transcoding.

- range: 1-16
- default: 3
- suggested: 16 (realtime b-adapt = 1), 5 (pre-transcode b-adapt = 2)

me

Motion estimation is the algorithm used to detect motion around a single point in the image. The default me = hex (hexagon) is good enough for most purposes. The next step up is umh (uneven multi-hexagon). Only the best quad+ core processors can handle this baby. Even greater is esa (exhaustive) and tesa (transformed exhaustive). These two naive algorithms offer the best quality but are only practical for overnight pre-transcodeing. Never use dia, the weakest motion estimation. Hex provides much better quality than dia without much overhead.

- range: dia, hex, umh, esa, tesa
- default: hex
- suggested: hex (realtime), umh (pre-transcode)

merange

When umh or greater is enabled for motion estimation, merange can be set to determine the search radius. An merange of 24 is about as much any modern processor can handle for realtime transcoding. Larger meranges can help with high motion films and cartoons. If you see horrible pixelation during high motion scenes, merange should be increased.

- range: 4-64
- default: 16
- suggested: 16 (realtime hex), 24-32 (pre-transcode umh)

deblock

Deblock decides how to blur surrounding pixels of similar color into a bigger block. This can be used to either sharpen or soften an image. Deblock has two values separated by a colon: X:Y. Both X and Y are integers varying from -3 to 3, 0 being the default for both. X decides the strength of deblocking. Higher numbers assimilate the colors more. Y is the upper bound of what colors to consider for deblocking. Higher numbers, means more parts of the picture will be smoothed. The default 0:0 setting usually ensures optimal PSNR. However, there are rare cases where tilting it from 0:0 can increase quality. For example, cartoons contain huge areas of the same color. Deblocking those patches of color would save space, allowing us to allocate more bits for the edges. Some suggest 2:2 for cartoons.

- range: -6:-6 to 6:6
- default: 0:0
- suggested: stay within -3 and 3 for both parameters

scale

Resolution is actually a VLC transcode setting, not an x264 setting, but it's very important in this discussion. Transcoding high resolution videos demands a very strong CPU. The more pixels you have the more times subme, ref, me, and merange calculations have to be applied. If you find that your transcode is eating up all your CPU and causing your stream to stutter, the easiest thing to do is lower your resolution. Use the scale property to alter your resolution. Note that scaling it down 50% doesn't just cut down the number of pixels by half. It actually cuts down the work 4-fold. (Example: 640x480 == 307,200. 50% scale of 640x480 is 320x240 = 76,800. 307,200 / 76,800 = 4). That's a ton of less pixels x264 has to deal with.

- range: 0.0-1.0
- default: 1.0
- suggested: 0.65 - 0.75 for 720p videos

threads

Imagine if you had to cook a meal for a huge party. 1 person couldn't do it by himself. 2 people working simultaneously would finish the job twice as fast. 4 people would get it done *almost* 4 times as fast. Why almost? Because the 4 people would need to communicate constantly to keep on track and maybe someone would have to wait while another is hogging up the sink. What if we had 100 people in the kitchen? I bet the meal would never be cooked because of all the people suffling around and butting heads. This analogy can be converted to threads of a computer program and cores of a CPU. The cooks are the threads and the cores represent the space in the kitchen. You must have the right balance to get the job done quickly. For

most cases, you should set threads to be the same number of logical cores you have. For example, an Intel Core i7 920 has 4 physical cores / 8 logical cores (each physical core has hyperthreading), so you would set threads to 8. There are cases where you would use less threads than logical cores. If you plan on doing other activities, like watching a movie while you're broadcasting, lower the thread count by 1 to reduce thrashing.

- range: 1 - 128
- default: unset
- suggested: 2 - 12 (same as the number of logical cores you have)

Setting it up on Veetle

You can access the x264 settings by expanding "advanced options" in the quality step of the broadcast page. The sliders are there for your convenience. Advanced users can edit the stream output field directly below the sliders. Any setting that you leave out in the x264 block will take on its default value. For example, the following two transcode blocks lead to identical results:

```
#transcode{vb=800, venc=x264{bpyramid=none, weightp=0}, vcodec=h264, acodec=mp3, ab=64, threads=4}:std{access=http,
mux=asf, dst=127.0.0.1:1234}
```

```
#transcode{vb=800, venc=x264{me=hex, subme=5, ref=3, bframes=3, deblock=0:0, bpyramid=none, weightp=0}, scale=1.0,
vcodec=h264, acodec=mp3, ab=64, threads=4}:std{access=http, mux=asf, dst=127.0.0.1:1234}
```

Realtime transcoding suggestions

High Definition 720p

CPU	width	vb	threads	ref	subme	me
Intel Core 2 Duo / Intel Core i3 / AMD Athlon II	1280	1300	2	1	1	hex
Intel Core 2 Quad / Intel Core i5 / AMD Phenom	1280	1300	4	2	3	hex
Intel Core i7 / AMD Thuban	1280	1300	8	3	5	umh

```
#transcode{venc=x264{subme=1, ref=1, bframes=16, b-adapt=1, bpyramid=none, weightp=0}, vcodec=h264, vb=1300, width=1280, acodec=mp3, ab=96,
threads=2}:std{access=http, mux=asf, dst=127.0.0.1:1234}
```

```
#transcode{venc=x264{subme=3, ref=2, bframes=16, b-adapt=1, bpyramid=none, weightp=0}, vcodec=h264, vb=1300, width=1280, acodec=mp3, ab=96,
threads=4}:std{access=http, mux=asf, dst=127.0.0.1:1234}
```

```
#transcode{venc=x264{subme=5, ref=3, bframes=16, b-adapt=1, bpyramid=none, weightp=0}, vcodec=h264, vb=1300, width=1280, acodec=mp3, ab=96,
threads=8}:std{access=http, mux=asf, dst=127.0.0.1:1234}
```

Standard Definition 480p

CPU	width	vb	threads	ref	subme	me
Intel Core 2 Duo / Intel Core i3 / AMD Athlon II	640	600	2	3	5	hex
Intel Core 2 Quad / Intel Core i5 / AMD Phenom	640	600	4	4	7	hex
Intel Core i7 / AMD Thuban	640	600	8	5	9	umh

```
#transcode{venc=x264{subme=5, ref=3, bframes=16, b-adapt=1, bpyramid=none, weightp=0}, vcodec=h264, vb=600, width=640, acodec=mp3, ab=96,
threads=2}:std{access=http, mux=asf, dst=127.0.0.1:1234}
```

```
#transcode{venc=x264{subme=7, ref=4, bframes=16, b-adapt=1, bpyramid=none, weightp=0}, vcodec=h264, vb=600, width=640, acodec=mp3, ab=96,
threads=4}:std{access=http, mux=asf, dst=127.0.0.1:1234}
```

```
#transcode{venc=x264{subme=9, ref=5, bframes=16, b-adapt=1, bpyramid=none, weightp=0}, vcodec=h264, vb=600, width=640, acodec=mp3, ab=96,
threads=8}:std{access=http, mux=asf, dst=127.0.0.1:1234}
```