VirtualDub documentation: Optimization

4 min read • original

How do I best speed up compression?

There is only one tip I can give: try enabling "fast repack." When this is enabled, VirtualDub attempts to negotiate a fast YUV or YCrCb path between the video decompression and compression codecs. Both input and output video streams must be compressed, and you can get weird artifacts or flipped video with buggy codecs -- do not use it if Indeo Video R3.2 or Radius Cinepak are involved. This option will also disable all video filters, however, so it's mainly only useful for transcoding. It's not unusual to see overall frame rates increase by 20% when using fast repack.

Fast repack cannot be used with Ben's original versions of Avisynth, because they only output RGB data. Others have since released updated versions of Avisynth that can output YUY2 as well, and these will work in fast repack.

How do I best speed up capture?

Preview mode tremendously slows down the system due to the constant CPU blits, so don't use it unless you're sure you can spare the CPU time, and even then it's best to switch to a 16-bit desktop. If you're extremely pressed for CPU bandwidth, turning off overlay can help as well. When the capture device is on the video card, however, overlay is practically free.

If at all possible, set the video capture format to YUY2, or failing that, UYVY. Most video codecs work in YCrCb space and work considerably faster if they don't have to do RGB-to-YCrCb conversion on input. Any YCrCb format accepted by the video codec you want to use should work, but YUY2 and UYVY are the most popular.

I enabled DirectDraw acceleration, and it's not any faster.

If this option is enabled, VirtualDub attempts to use a DirectDraw video overlay during input preview, and blits directly to the primary surface on an output preview. That's it. It has absolutely no effect when saving to disk.

Could VirtualDub be sped up with 3D video hardware?

Only if you want to rasterize a very complex 3D scene. Experience has shown that while video cards are fast at transferring data into video memory, or manipulating data already in video memory, they are very slow at transferring back into system memory, which would be required for VirtualDub to use 3D hardware. The performance ratio is approximately 10:1 for CPU blits (most cards cannot busmaster video-to-system blits), and more like 30:1 when 3D graphics are involved -- even with AGP.

Another problem is that popular 3D graphics accelerators do not have the accuracy needed for video work; at best, you can kick the accelerator into 32-bit. You'll still be limited to bilinear filtering, with decimation approximated by mipmapping, and since most convolution algorithms would have to be approximated with multiple textured primitives, you won't get the benefits of 48-bit intermediate accumulation that most of VirtualDub's software routines use.

A third problem is that many 3D drivers are bug-ridden and unreliable. Even now, many 3D OpenGL drivers -- including those from NVIDIA, ATI, and Matrox -- do not correctly handle all combinations of internal format and input format for `glTexImage2D()`, producing bizarre textures that have red and blue swapped, half intensity, or incorrectly thresholded alpha. Some are so bad that the machine will

bluescreen if you do as little as calling `glEnd()` right after `glBegin()`. The situation with Direct3D is better, but still not very palatable.

Incidentally, even 2D drivers aren't often very reliable. I managed to bluescreen the NVIDIA 6.50 WHQL drivers for Win2K once simply by right-clicking on VirtualDub's input DirectDraw overlay during a preview operation.

What do the Performance settings do?

Short answer: leave them alone.

AVI output buffering controls the size of VirtualDub's output buffer. A larger buffer can increase parallelism in the system by allowing VirtualDub to continue processing video while it is waiting on the hard disk. If you increase this value too much, however, it can cause VirtualDub to stall from trying to write too much to the disk at once. Generally, this should be left at 1MB, and it doesn't really affect speed at all unless VirtualDub is writing several megabytes per second to the hard disk -- something that doesn't happen during an encode to DivX. If you're doing a high bandwidth operation such as uncompressing an MPEG-1 file, though, raising this to 2MB or 4MB can help. Excessive seeking on your hard disk, or the processing fps dropping low or to zero during hard disk writes can be an indication that this is too high.

Wave input buffering doesn't really do much of anything, even when you're actually multiplexing in a WAV file, and should be left at 64K.

Stream data pipelining controls the number of audio and video buffers in flight in VirtualDub's pipeline; this is the companion to the AVI output buffering setting. One buffer is used for one video frame, or one video frame's worth of audio data. Again, this value rarely speeds anything up if you're doing a slow operation like recompression, and raising it can have extremely bad consequences -- setting 256 buffers on a 640x480, 24-bit uncompressed RGB file could conceivably cause VirtualDub to allocate 235MB of memory. Increasing it on a direct stream copy operation with a highly compressed input could help, but that tends to already be speedy with the defaults.

What kind of Inverse Discrete Cosine Transform (IDCT) does VirtualDub use? Can I speed it up or use a reference mode for higher quality, like in FlaskMPEG?

VirtualDub has two IDCT routines: a scalar one, and an MMX one. The scalar one is based on the 2D IDCT from the Java MPEG Player with corrections, and the MMX one is a modified and completed version of Intel's AP-922 IDCT. Both comply with the IEEE-1180 specification for IDCT accuracy. VirtualDub's internal MJPEG decoder only uses the MMX version, and the MPEG-1 decoder chooses the scalar one if MMX is not available. **Neither routine is used anywhere else in the program.**

Including MPEG-1 motion compensation arithmetic, the scalar IDCT runs in ~1000 clocks and the MMX one in ~300, on average. This is less than 10% of the CPU during VideoCD decoding on a Pentium MMX 200, and very little could be gained overall by further optimization. The quality gain by switching between scalar and MMX implementations is insignificant, and the only change that would occur if a slow floating-point implementation were introduced would likely be a halving of decoding speed.

So, to answer the last two questions, no and no.

Where's the xxx optimized version?

VirtualDub includes scalar, MMX, and integer SSE (MMX2) code paths in one executable, and automatically chooses whichever is appropriate for your CPU. There are no CPU-specific builds and you do not need to tweak settings to optimize for a particular processor.

Are there Athlon and Pentium 4 optimized versions?

No -- the main distribution is suitable for all CPUs from a non-MMX Pentium up. VirtualDub could probably stand for some Pentium 4 optimizations, but I don't have a Pentium 4 myself, don't really care to

optimize for it (4 cycle shift my #$*(@#$) and don't plan to get one any time soon. As for the Athlon, nearly all of VirtualDub's image processing routines are integer, so there are very few opportunities to take advantage of 3DNow! and the Athlon's superior FP unit.

In short: MMX is king, and SSE/3DNow! are kind of <bleh> for non-3D work.

Original URL:
http://virtualdub.com/docs_faster.html