An application of mine uses DirectDraw to draw video frames on multiple screens, and so far the visualization pipeline used a group of off-screen YUV surfaces that, at the end of the process, are drawn into the primary surface. So this is the code that creates the primary surface:

{CODE brush: cpp; ruler: true;}DDSURFACEDESC ddsd;
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(DDSURFACEDESC);
ddsd.dwFlags = DDSD_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
if ((ddrval = lpDD->CreateSurface(&ddsd, &lpDDSPrimary, NULL)) != DD_OK)
{
return VISLIB_ERROR;
}{/CODE}

and this is the code that creates an off-screen YUV surface:

{CODE brush: cpp; ruler: true;}ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT |
DDSD_WIDTH | DDSD_PIXELFORMAT;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwWidth = SurfaceWidth;
ddsd.dwHeight = SurfaceHeight;
ddsd.ddpfPixelFormat.dwFlags = DDPF_FOURCC;
ddsd.ddpfPixelFormat.dwFourCC = MAKEFOURCC('U','Y','V','Y');
ddsd.ddpfPixelFormat.dwYUVBitCount = 16;
if ((ddrval = lpDD->CreateSurface(&ddsd, &lpDDSOff, NULL)) != DD_OK)
{
return VISLIB_ERROR;
}{/CODE}

At the end of the pipeline, the processed image is in the lpDDSOff surface, and gets copied to the primary surface lpDDSPrimary with a simple blit:

{CODE brush: cpp; ruler: true;}ddrval = lpDDSPrimary->Blt(&DestRect, lpDDSOff,

&SourceRect, DDBLT_WAIT, NULL);{/CODE}

The last blit does not even perform any rescaling, as the image in the off-screen surface already has the correct dimensions of the DestRect in the primary surface, so it's just a copying of data from the off-screen to the primary surface, and color-space conversion from YUV to the color space of the primary surface (these days, RGB32 is definitely a safe bet). This visualization pipeline has been running fine for years (it works fine up to nVidia 28x driver series), but then after nVidia 29x drivers series, including the latest 30x versions, performance dropped in a dramatic way, blits were so slow that they were dragging down the whole system. So I started benchmarking the various steps of the visualization pipeline, and it turns out that the latest step, that humble blit you see above, was about 100 times slower with newer drivers than with older ones! The performance was even poorer if the code was blitting on a secondary monitor, requiring literally many milliseconds to draw a single image to the screen. Even worse, the GPU usage is really high even when drawing only a few video streams at the same time, so the GPU is clearly a performance bottleneck, and it should not be, as there are no complex operations going on.

The solution is switching the off-screen surfaces from YUV to RGB32, so the declaration of the surfaces becomes:

{CODE brush: cpp; ruler: true;}ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH | DDSD_PIXELFORMAT;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwWidth = SurfaceWidth;
ddsd.dwHeight = SurfaceHeight;
ddsd.ddpfPixelFormat.dwFlags = 0;
// Set up the pixel format for 32-bit RGB (8-8-8).
ddsd.ddpfPixelFormat.dwSize = sizeof(DDPIXELFORMAT);
ddsd.ddpfPixelFormat.dwFlags= DDPF_RGB;
ddsd.ddpfPixelFormat.dwRGBBitCount = 32;
ddsd.ddpfPixelFormat.dwRBitMask = 0x00FF0000;
ddsd.ddpfPixelFormat.dwGBitMask = 0x0000FF00;
ddsd.ddpfPixelFormat.dwBBitMask = 0x000000FF;{/CODE}

The blit to the primary surface become lighting-quick, with minimal GPU usage. The downside of this solution is that

- copying a RGB32 image to video memory instead of a YUV image takes exactly twice the bandwidth
- the CPU must perform a YUV -> RGB32 conversion while copying data to video memory

After benchmarking, both downsides seem to be quite minor, due to the speed of sysmem to vidmem memcopy, and that highly optimized versions of YUV->RGB32 color-space conversions are available in the Intel IPP library. Summing up, the performance of RGB32 pipeline on 30x driver series is on a par with that of the YUV pipeline on 28x driver series, and definitely within the required performance boundaries.