GPU-Assisted Decoding of Video Samples Represented in the YCoCg-R Color Space

Wesley De Neve
Ghent University - IBBT
Multimedia Lab
Sint-Pietersnieuwstraat 41
B-9000 Ghent, Belgium
Wesley.DeNeve@-UGent.be

Dieter Van Rijsselbergen
Ghent University - IBBT
Multimedia Lab
Sint-Pietersnieuwstraat 41
B-9000 Ghent, Belgium
Dieter.VanRijssel-bergen@UGent.be

Charles Hollemeersch
Splash Damage Ltd.
25 Elmfield Road
Bromley Kent BR1 1LZ
United Kingdom
chollemeersch@-gmail.com

ABSTRACT

Although pixel shaders were designed for the creation of programmable rendering effects, they can also be used as generic processing units for vector data. In this paper, attention is paid to an implementation of the YCoCg-R to RGB color space transform, as defined in the H.264/AVC Fidelity Range Extensions, by making use of pixel shaders. Our results show that a significant speedup can be achieved by relying on the processing power of the GPU, relative to the CPU. To be more specific, high definition video (1080p), represented in the YCoCg-R color space, could be decoded to RGB at 30 Hz on a PC with an AMD Athlon XP 2800+ CPU, an AGP bus and an NVIDIA GeForce 6800 graphics card, an effort that could not be realized in real-time by the CPU.

Categories and Subject Descriptors

I.3.3 [**Computing Methodologies**]: Computer Graphics—*picture/-image generation*

General Terms

Performance

Keywords

FRExt, GPU, H.264/AVC, pixel shaders, YCoCg, YCoCg-R.

1. INTRODUCTION

H.264/Advanced Video Coding (H.264/AVC) is a standardized specification for digital video coding, characterized by a design that targets efficiency, robustness, and usability. The first version of this standard primarily focused on entertainment-quality video, based on an eight bits per sample representation and a 4:2:0 chroma sampling format. In July, 2004, a new amendment was added to H.264/-AVC, called Fidelity Range Extensions (FRExt, Amendment 1) [4]. The extensions in question make it possible to address the needs of more demanding applications, especially in the domain of studio editing and post-processing, content contribution, and content distribution. Among other coding tools, FRExt introduces a new color space called YCoCg (luma, chroma orange, and chroma green), as well as a variant that is known as YCoCg-R. The R refers to its lossless reversible RGB (red, green, and blue) to YCoCg mapping, an important property in the context of lossless video coding and its applications (e.g., medical imaging). The YCoCg-R color space is only available in FRExt's High 4:4:4 Profile.

As shown by Shen *et al.* [2], a color space conversion is one of the most computationally intensive parts in a typical video decoder architecture: a CPU load of up to 40% is possible for an ordinary color space conversion. Hence, this observation makes it very desirable to handle this step efficiently. Although recent graphics hardware offers more and more support for common color spaces, such as ITU-R Recommendations BT.601 and BT.709 (International Telecommunication Union - Radiocommunication Sector), this is often not the case for new or proprietary color spaces because the hardware has built-in coefficients that cannot be changed. In this study, we investigate how the DirectX 9 programmable graphics pipeline can be exploited to assist the CPU in decoding YCoCg-R video samples to RGB for visualization purposes. This allows to repurpose already existing consumer graphics hardware for accelerating new color space conversions.

The outline of the paper is as follows. In Section 2, we give a brief overview of the YCoCg color space and some of its variations. Section 3 discusses how shaders can be used in order to accelerate the decoding of YCoCg-R video data. Some experimental results are provided in Section 4 while Section 5 concludes.

2. THE YCoCg(-R) COLOR SPACE

As discussed by Malvar *et al.* [1], the YCoCg color space is not only characterized by a simple set of transformation equations relative to RGB, but also by an improved coding gain relative to both RGB and YCbCr (luma, chroma blue, chroma red). The primary motivation behind the development of this color space is to address some shortcomings with respect to the different YCbCr color spaces, such as difficult to use floating-point coefficients and rounding errors [3]. When relying on the YCoCg color space, rounding errors can be eliminated if two additional bits of accuracy are used for representing luma and chroma samples. However, it is even possible to devise a smarter approach that does not require adding precision to the luma samples and that only adds one bit of precision to the chroma samples. This scheme is known as the YCoCg-R color space and can be described by the following equations, relative to RGB and aimed at being executed by integer arithmetic:

$$\begin{array}{c} Co = R - B \\ t = B + (Co >> 1) \\ Cg = G - t \\ Y = t + (Cg >> 1) \end{array} \Leftrightarrow \begin{array}{c} t = Y - (Cg >> 1) \\ G = Cg + t \\ B = t - (Co >> 1) \\ R = Co + B \end{array} \tag{1}$$

In the context of H.264/AVC FRExt, the YCoCg-R color space is used as an intermediate format by a coding tool known as the residual color transform. However, video data that are represented in the just mentioned color space can also be used as input for an encoder and as output of a decoder, an option that can be signaled by H.264/AVC's Video Usability Information (VUI). It is the latter application that is aimed at in this paper. To be more specific, a decoder is targeted that delivers video samples in the YCoCg-R color space, hereby making use of eight bits per luma component and nine bits per chroma component, and using a 4:4:4 chroma sampling format. Note that the targeted decoders do not necessarily have to claim conformance with the H.264/AVC (FRExt) specification. The YCoCg-R video data could even be the result of a rendering process that has stored its data to an intermediate file on disk for further processing or visualization.

3. GPU-ASSISTED YCoCg-R DECODING

In this section, a discussion is provided on how a programmable Graphics Processing Unit (GPU), as often available in mainstream personal computers and game consoles, can be used for accelerating the decoding of YCoCg-R video samples, despite the simple and integer-valued nature of the color space transform.

3.1 Shaders

In order to gain access to the computational power of the GPU on a graphics card, it is necessary to make use of vertex or pixel shaders. Shaders are small programs that are designed to be executed by the GPU of a graphics card. They are typically used for rendering programmable effects in games (and motion pictures). Vertex shaders generally describe a procedure to be applied to vertices in a 3-D scene. In the context of digital video, they can, for instance, be used for the implementation of deformation effects. However, our interest lies in the usage of pixel or fragment shaders. This type of shaders is able to calculate one or more per-pixel colors using texture values or mathematical functions (e.g., perlin noise), and they are invoked once for every pixel that is drawn. As such, they are suited for implementing the color space transforms.

Pixel shaders come in different versions, each of them imposing certain restraints on the number of available registers in the shader units on the graphics hardware and their accuracy, the maximum length of the pixel shaders in terms of arithmetic and texture instructions, ... In this paper, second generation pixel shaders (commonly refered to as ps_2_0 pixel shaders) are used for the implementation of the inverse YCoCg-R transform, as well as Microsoft's High-Level Shader Language (HLSL) for writing them.

3.2 Video Textures

Several steps are needed in order to make the video samples, as represented in the YCoCg-R color space, available to the pixel shader units on the graphics hardware. Since no official file format is available for the persistent storage of YCoCg-R video samples, an own format was developed, as illustrated by Figure 1. For every eight chroma bytes, a ninth byte is provided that acts as a collector for the (most significant) ninth bit of each of the eight chroma samples. This allows to limit the overhead in the file format, but it requires an additional reconstruction step when transferring the YCoCg-R frames from a disk to the system memory.

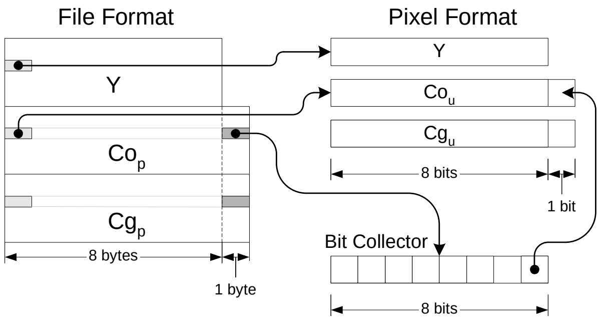

Figure 1: File format for YCoCg-R pictures. Co$_p$ and Cg$_p$ are packed chroma samples. Co$_u$ and Cg$_u$ are unpacked samples.

The unpacked YCoCg-R video samples are subsequently transfered from the system memory to the video memory of the graphics card by using three separate textures per picture (i.e., one texture for each color channel). As such, this makes it possible for a pixel shader to get access to the video samples in question for further processing. With respect to the luma samples, an integer-valued 8-bit texture format is used (described by the D3DFMT_L8 identifier in Direct3D). Because of the fact that chroma samples are represented with nine bits, an integer-valued texture format of two bytes per element is used (D3DFMT_L16) for both of the channels, thus, unfortunately introducing an overhead of 44% per chroma channel.

3.3 The YCoCg-R to RGB Pixel Shaders

With respect to ps_2_0 hardware, it is important to be aware of the fact that this hardware operates primarily on floating-point data. This is already an indication that it will not be straightforward to translate the YCoCg-R inverse transform equations, focusing on integer arithmetic, to the just mentioned floating-point hardware. In case of ps_2_0 hardware, it is also relevant to know that the length of a pixel shader is limited to 64 arithmetic instructions and 32 texture instructions, thus severely limiting a ps_2_0 shader's complexity.

A first attempt to implement the YCoCg-R inverse transform as a pixel shader, resulted in the following code fragment in HLSL:

```
float4 PS_1(VSOutput_1tex inp) : COLOR {
   float t;
   float3 r;
   float3 y = tex2D(ySampler, inp.tex0);
   float3 co = tex2D(uSampler, inp.tex0);
   float3 cg = tex2D(vSampler, inp.tex0);
   t = y - cg / 2;
   r.g = t + cg;
   r.b = t - co / 2;
   r.r = co + r.b;
   return float4(r,1);
}
```

Figure 2 (i) clearly shows that this shader does not produce the desired output. The luma information is processed in a correct way, but the chroma information is wrong. An explanation for this result can be found in the fact that video samples are internally represented in the graphics hardware as normalized floats with the same range (clamped at [0..1]). Because chroma samples are characterized by a sample depth of nine bits, their range is twice the range of the luma samples (having a sample depth of eight bits). Taking this observation into account, a new pixel shader was devised, again based on the set of equations in (1) but in which divisions were left out and in which multiplications were introduced in order to use the full range of possible chroma sample values. However, as illustrated by Figure 2 (ii), this approach does not lead to better results. The nature of the distortion indicates that a lot of color intensities were limited to their maximum value, instead of using modulo math for compensating the overflows. Correcting this behaviour resulted in the output as depicted by Figure 2 (iii), still showing some artifacts that could not be explained by the authors at the time of writing, thus requiring further research.

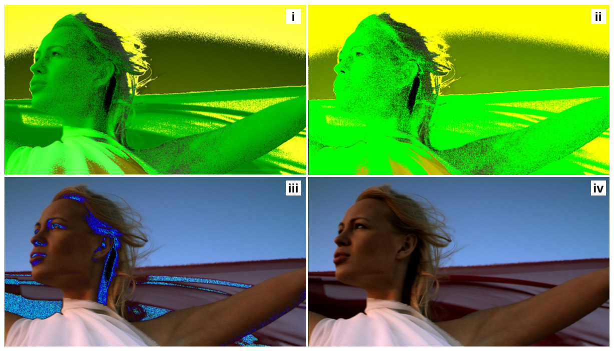

Figure 2: Shader output: (i) chroma and luma samples with the same range; (ii) chroma samples with a correct range, but resulting in an overflow during decoding; (iii) overflow correction; (iv) correct RGB output by emulating integer math.

Instead of starting from a simple set of instructions in order to obtain the desired result, another approach was used in which the integer oriented inverse YCoCg-R operations are emulated by the floating-point graphics hardware. The resulting pixel shader, highly optimized and not discussed in detail due to place constraints, is described below in HLSL. It compiles to a shader with a length of 22 instructions. Its correctness was verified by relying on an exhaustive test that takes into account all possible RGB colors (using eight bit per channel). The floating-point coefficients in the shader make it possible to compensate for the internal scaling of the texture values to normalized floats; the fractional part compensates for rounding errors. As such, they no longer fulfill one of the design goals of the YCoCg(-R) color space (i.e., elegant transform coefficients).

```
float4 PS_7(VSOutput_1tex inp) : COLOR {
    float t;
    float4 r = 0;
    float3 f = {255.5, 512.4921875, 512.4921875};
    float3 ycocgr = 0;
    ycocgr.x = tex2D(ySampler, inp.tex0);
    ycocgr.y = tex2D(uSampler, inp.tex0);
    ycocgr.z = tex2D(vSampler, inp.tex0);
    ycocgr = floor(ycocgr * f);
    t = ycocgr.x - floor(ycocgr.z / 2);
    r.g = t + ycocgr.z;
    r.b = t - floor(ycocgr.y / 2);
    r.r = r.b + ycocgr.y;
    r = frac(r / 256 + 0.001953125);
    return r;
}
```

4. SIMULATION RESULTS

This section covers some performance results that were obtained, as well as a discussion pertaining to the test architecture used.

4.1 The Persephone Engine

The Persephone Engine is an own developed software platform, built on top of the DirectX 9 Effects Framework[1]. As such, it allows to quickly implement and test vertex and pixel shaders on uncompressed video data. The work flow in the engine is as follows. First, uncompressed video data are fetched from the hard disk by objects called producers. The video data in question are typically YCbCr video data in a 4:2:0 planar format. They can optionally be cached in the system memory in order to make an application independent of the transfer speed of a hard disk. This behavior makes it also possible to rule out possible differences with respect to the processing speed of different decoders. Second, after an optional pre-processing step, the video data are then pushed to a renderer for further processing and visualization. For this research, the Persephone Engine was extended with three additional renderers: a CPU-based renderer that relies on an MMX-based (MultiMedia eXtensions) implementation of the YCoCg-R inverse color space transform, shown in Figure 3, and two GPU-based renderers that rely on pixel shaders for implementing the desired functionality.

```
mov eax, y
punpcklbw mm0, [eax]
psrlw mm0, 8
mov eax, cg
movq mm2, [eax]
mov eax, co
movq mm1, [eax]
movq mm3, mm2
psrlw mm3, 1
psubw mm0, mm3
paddw mm2, mm0
movq mm3, mm1
psrlw mm3, 1
psubw mm0, mm3
paddw mm1, mm0
pand mm1, mm7
pand mm2, mm7
packuswb mm1, mm3
packuswb mm2, mm5
punpcklbw mm2, mm1
psllw mm0, 8
movq mm6, mm0
punpckhwd mm0, mm2
psrad mm0, 8
mov eax, rgb
movntq [eax+8], mm0
punpcklwd mm6, mm2
psrad mm6, 8
movntq [eax], mm6
```

Figure 3: MMX version of the YCoCg-R to RGB transform (register mm7 is initialized with 00FF00FF00FF00FF).

The CPU renderer transforms the YCoCg-R video data to RGB samples by relying on MMX instructions, a common way for accelerating color space conversions. As such, this implementation is a good reference point for a comparison with the GPU-based implementations. Note that a picture, after having been decoded by the CPU, is immediately copied to an RGB surface in the video memory of the graphics card. This picture will then be placed in the frame buffer for visualization purposes.

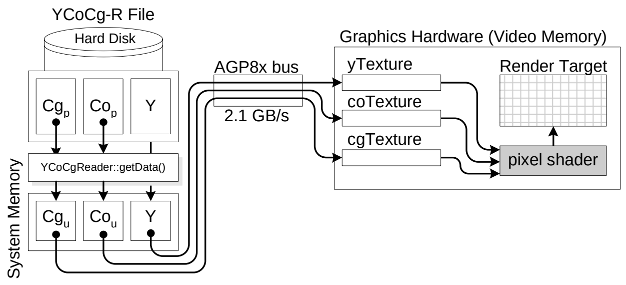

Figure 4: Data flow for the first GPU renderer.

The first GPU-based renderer is an intermediate step towards a more optimized version. Figure 4 illustrates the data flow pertaining to this implementation. Just like in the CPU oriented renderer, the video data are first retrieved from the frame cache. After unpacking the YCoCg-R samples (i.e., going from the file format to

[1]It is worth mentioning that the Persephone Engine is a lightweight alternative for a test platform built on top of the Video Mixing Renderer 9 object (VMR-9). This complex and difficult to maintain object, bridging the gap between DirectShow and Direct3D, also makes it possible to render (compressed) video data to textures.

the pixel format), each color channel is then uploaded via the Advanced Graphics Port (AGP) bus to a corresponding texture in the graphics card's video memory. The textures are subsequently decoded to an RGB buffer.

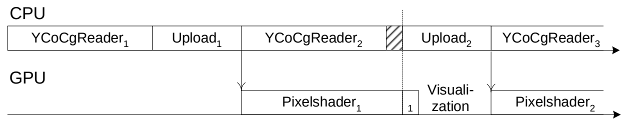

Figure 5: Time line of events for the first GPU renderer.

The time line for the GPU-based implementation is plotted in Figure 5. After retrieving a picture, it is uploaded from the system memory to the video memory. Subsequently, a draw instruction is sent to the GPU, resulting in the evaluation of our shader for all pixels in the picture. Note that this operation is performed in an asynchronous way with the application, thus allowing the CPU to retrieve and upload a new picture at the same time. However, the latter is only possible when no draw operations are being executed that are making use of the targeted piece of video memory. This is also illustrated by the dead period in the timeline: the upload procedure has to wait for the completion of the pixel shader routine.

The just mentioned read and write conflicts can be solved by making use of a technique known as texture double buffering, as implemented by the second GPU renderer. Hereby, additional textures are introduced, allowing to switch between them during successive read and write operations: while the GPU is still relying on a first texture for the current draw operation, the CPU can already upload video data to a second texture. The double buffering technique also makes it possible to get rid of an unnecessary copy of the YCoCg-R data in the system memory since it is now possible to immediately upload the data in question to the textures in the video memory. As such, the CPU only has to take care of unpacking the YCoCg-R data from the frame store, while the GPU is able to apply the conversion in parallel with the activities of the CPU.

4.2 Discussion of the Results

The performance of the GPU-based renderers was tested on a PC with an AMD Athlon XP 2800+ CPU. Two graphics cards were used: an NVIDIA GeForce FX 5900 and an NVIDIA GeForce 6800 graphics card. During each test, 300 YCoCg-R pictures were transformed to RGB pictures for visualization on a screen-aligned quad, hereby providing a 1:1 mapping between the texels in the textures and the pixels on the screen. No disc activity was allowed during each run in order to minimize the influence from the hard disc. The test sequence used was the high definition progressive VIPER test sequence as distributed by FastVDO.

Table 1: Performance results.

Graphics Card	Resolution	CPU Renderer (fps)	GPU Renderer 1 (fps)	GPU Renderer 2 (fps)
FX5900	768x576	75.2	81.9	148.2
	1024x576	55.0	60.6	109.4
	1280x720	35.4	39.0	70.3
	1920x1080	15.7	17.3	30.3
6800	768x576	75.3	81.6	149.7
	1024x576	54.9	59.6	110.1
	1280x720	35.6	38.6	70.8
	1920x1080	15.9	17.3	31.5

With respect to Table 1, the most important observation is the fact that the second GPU renderer clearly outperforms the CPU renderer, as well as the GPU renderer: the second GPU renderer is approximately twice as fast as the CPU renderer. A more detailed analysis showed that this behaviour can entirely be explained by the faster read routine of the second GPU renderer: the latter does not have to write the YCoCg-R pictures to the system memory but can immediately upload them to the video memory of the graphics card. Apparently, this makes the routine in question about 25% faster than the read routine of the CPU renderer (that takes up two third of the total time needed for processing a picture). NVPerfHUD, a tool for diagnosing performance bottlenecks in Direct3D 9 applications, even showed that both GPUs still had idle time in case of the second GPU renderer, illustrating the fact that the performance of a GPU is now determined by other factors, such as the speed of the CPU, the AGP bus and the driver that provides the GPU with the necessary instructions and vector data. Note that the second GPU renderer is the only one that can achieve real-time decoding of 1080p pictures in the YCoCg-R color format, captured at 30 Hz.

5. CONCLUSIONS

In this paper, we have discussed an implementation of the inverse YCoCg-R color space transform, aimed at integer arithmetic, by making use of pixel shaders, aimed at processing fixed-point and floating-point vector data. Special attention had to be paid, not only to the design of the pixel shaders, but also to the underlying application in order to exploit a maximum parallelism between the CPU and GPU. As such, we have demonstrated that our pixel shader based approach is significantly faster for YCoCg-R color space decoding than an optimized MMX-based implementation.

6. ACKNOWLEDGMENTS

The authors would like to thank Michiel Ronsse and Kristof Beyls for their advice with respect to the MMX implementation.

The research activities that have been described in this paper were funded by Ghent University, the Interdisciplinary Institute for Broadband Technology (IBBT), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research-Flanders (FWO-Flanders), the Belgian Federal Science Policy Office (BFSPO), and the European Union.

7. ADDITIONAL AUTHORS

Additional authors: Jan De Cock, Stijn Notebaert (Ghent University - IBBT, Multimedia Lab, email: {Jan.DeCock, Stijn.-Notebaert}@UGent.be), and Rik Van de Walle (Ghent University - IBBT - IMEC, Multimedia Lab, email: Rik.Vande-Walle@UGent.be).

8. REFERENCES

[1] H. Malvar and G. Sullivan: YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range. JVT-document JVT-I014, Trondheim, Norway, JVT, July 2003.

[2] G. Shen, G. Gao, S. Li, H.-Y. Shum, and Y.-Q. Zhang, Accelerate Video Decoding with Generic GPU. IEEE Trans. Circuits Syst. Video Technol. 15 (5) (2005) 685–693.

[3] G. Sullivan: Approximate Theoretical Analysis of RGB to YCbCr to RGB Conversion Error. JVT-document JVT-I017, Trondheim, Norway, JVT, July 2003.

[4] G. Sullivan, P. Topiwala, and A. Luthra, The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions. In Proceedings of SPIE Annual Meeting 2004: Signal and Image Processing and Sensors, volume 5558, pages 454-474, Denver, 8 2004.