GPU-DRIVEN RECOMBINATION AND TRANSFORMATION OF YCoCg-R VIDEO SAMPLES

Dieter Van Rijsselbergen, Wesley De Neve, and Rik Van de Walle
Department of Electronics and Information Systems - Multimedia Lab
Ghent University - IBBT
Gaston Crommenlaan 8/201, B-9050 Ledeberg-Ghent, Belgium
e-mail: {Dieter.VanRijsselbergen, Wesley.DeNeve, Rik.VandeWalle}@UGent.be

ABSTRACT
Common programmable Graphics Processing Units (GPU) are capable of more than just rendering real-time effects for games. They can also be used for image processing and the acceleration of video decoding. This paper describes an extended implementation of the H.264/AVC YCoCg-R to RGB color space transformation on the GPU. Both the color space transformation and recombination of the color samples from a nontrivial data layout are performed by the GPU. Using mid- to high-range GPUs, this extended implementation offers a significant gain in processing speed compared to an existing basic GPU version and an optimized CPU implementation. An ATI X1900 GPU was capable of processing more than 73 high-resolution 1080p YCoCg-R frames per second, which is over twice the speed of the CPU-only transformation using a Pentium D 820.

KEY WORDS
GPU, H.264/AVC, YCoCg-R, pixel shaders, performance

1 Introduction

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 consumer-quality video, characterized by an eight bit per sample representation and a 4:2:0 chromaticity subsampling format. In July 2004, the Fidelity Range Extensions (FRExt) were amended to the H.264/AVC standard [1]. These extensions serve the more demanding markets like studio post production and medical imaging wherein preservation of quality is of great importance. Among the coding tools introduced by the Fidelity Range Extensions are two new color space transformations: YCoCg (luma, chroma orange, and chroma green) and its reversible variant YCoCg-R. The latter color space transformation allows for a lossless conversion from and to the RGB color space. This lossless quality is important in the context of (near) lossless video coding in the aforementioned application areas. The YCoCg-R color space was defined in the context of the High 4:4:4 profile of the FRExt amendment.

A color space conversion is one of the most computationally expensive operations in a typical video decoding process. As shown by Shen et al. [2] no less than 40% of the decoding time can be spent on this color space transformation, in case of a YCbCr to RGB tranformation. The goal of this research is to utilize the programmable 3-D graphics pipeline, implemented by consumer-oriented GPUs, to perform the color space transformation from YCoCg-R to RGB. More specifically, this paper discusses an extended implementation of a preliminary basic GPU-assisted YCoCg-R decoding process [3].

This paper is outlined as follows: Section 2 briefly outlines the YCoCg-R color space, followed by Section 3, where the basics of GPU shaders, and the architecture and shortcomings of the basic GPU-assisted implementation are laid out. Section 4 continues with describing the extended implementation. Section 5 compares performance results of the basic GPU, extended GPU and CPU-only version, and Section 6 concludes this paper.

2 The YCoCg-R Color Space

The YCoCg color space is characterized by a simple set of transformation operations, as well as an improved coding gain compared to both RGB and YCbCr [4]. The simple transformation definition addresses ambiguities such as difficult to use floating point constants and rounding errors to which YCbCr and similar color spaces are often subjected. By providing an additional bit for the representation of the chroma components, compared to the luma component, this transformation offers a reversible and lossless conversion between the RGB and YCoCg color space and is hence called YCoCg-R.

The YCoCg-R scheme requires a mere six integer operations per transformation and is defined by the following equations (1). If intermediate results were stored using more bits than reserved for each color channel, an additional bit masking operation must be applied to the final result.

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

2.1 The YCoCg-R file format

As there exists no standard file format for storing YCoCg-R images, an own format was developed, as illustrated in Figure 1. Considering the nature of current video coding specifications where luma and chromaticity are processed individually (and are combined only in the final decoding step) a planar format for laying out color component frames was opted for.

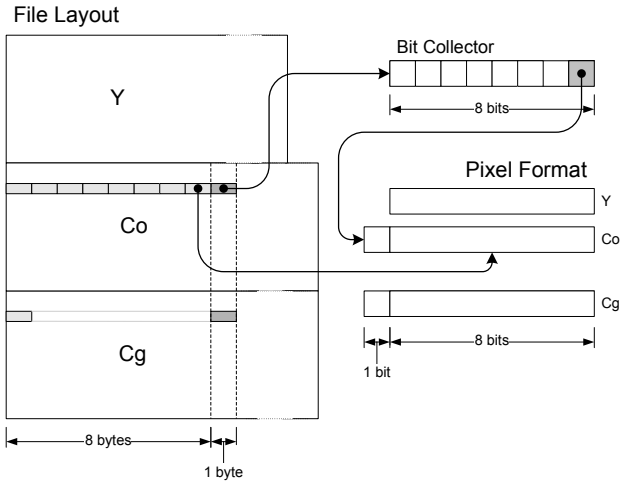

Figure 1. The YCoCg-R file format and its decomposition

A common representation of eight bits was chosen for the luma components. As is required for lossless YCoCg-R processing, one additional bit is added for representing the chroma samples. In order to maintain as much byte alignment as possible and not to suffer from any unused bits, the most significant bit of each of eight consecutive chroma samples is collected in a byte-sized bit collector. This collector is stored after the eight lower-bit bytes of the chroma samples. A unit of eight bytes and a bit collector is referred to as an interval. This file layout induces a memory fetch pattern that is identical for all samples in an interval.

3 GPU-assisted YCoCg-R decoding

3.1 Shaders

While the first generations of so-called 3-D acceleration chips implemented a 3-D pipeline of fixed functionality, newer programmable hardware generations allow full control over the way in which vertices and pixels are processed by exposing an extensive instruction set of vector-oriented floating point operations. These GPUs offer great floating point processing power for graphics-oriented applications and provide a number of parallel processing paths. GPUs operate simultaneously on both image pixels and pixel components (pixels can be considered as a vector of e.g., red, green, and blue for the RGB color space). In both cases, the GPU realizes the Single Instruction Multiple Data (SIMD) paradigm [5].

In order to gain access to the programmable core of the GPU, it is necessary to make use of vertex or pixel shaders. Vertex shaders are used to transform vertices in space into points on a two-dimensional plane which represents the screen. Pixel shaders are then employed to fill in (or rasterize) rectangles made up of transformed vertices. The pixel shader is invoked for every pixel drawn and its output is determined as a function of texture samples and numerical operations. As such, pixels shaders are suited for the implementation of a color space transformation. All shaders in this research were compiled for the PS_2_0 pixel shader profile, using the High Level Shading Language (HLSL) compiler. Rewriting the shaders in assembly code produced no substantial speedups. The compiler has proven capable of performing most of the optimizations itself.

3.2 The Persephone engine

The Persephone Engine, shown in Figure 2, is an own developed software platform, built on top of the Direct3D API and its Effects framework, that allows for extensive testing of the application of pixel and vertex shaders on uncompressed video images. The data flow in the Persephone Engine is as follows. Producer objects read the uncompressed video data from hard disk, thereby caching this data in system memory buffers to avoid hard-disk delays. Secondly, the data can be preprocessed and is then uploaded over the system interconnect bus (e.g., AGP or PCI Express) to the video memory on the graphics card. Finally, the 3-D graphics pipeline state, scene geometry and actual visualization (on a display or read back from the frame buffer into a file) is handled by a Renderer object. This Renderer object also initiates the upload process carried out by the Producer.

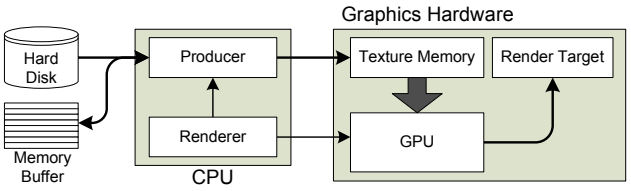

Figure 2. Schematic view of the Persephone engine

3.3 Limitations of the basic implementation

Figure 3 depicts the workflow of a basic implementation. The YCoCgReader producer reads YCoCg-R frames from disk using the routines provided by the operating system. It copies the luma plane directly to video memory in a 8-bit per sample texture format. The compacted chroma samples are recombined into a texture of 16-bit words in the video memory. The pixel shader then applies correct scaling to

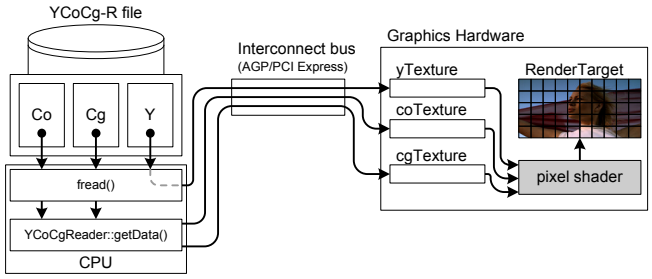

Figure 3. Data flow in the basic GPU implementation

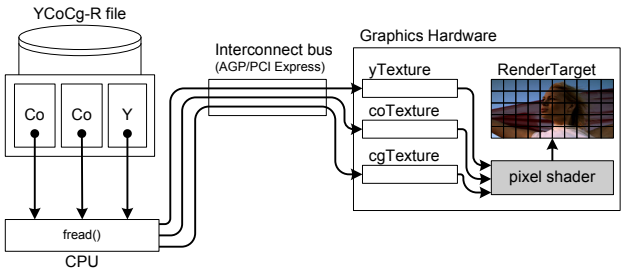

Figure 4. Data flow proposition for the extended implementation

all samples fetched from the texture memory and performs the color space transformation from YCoCg-R to RGB.

This approach, although already proven to be significantly faster than a pure CPU implementation [3], still has a number of limitations. Storing and uploading the chroma samples in 16-bit units, and thereby wasting 7 bits, introduces a 44% overhead in bus bandwidth and video memory usage. Also, the recombination of the chroma samples is performed by the CPU. This requires a couple of instructions per sample, and is of considerable cost compared to the mere six integer operations required for the actual color space transformation.

4 Recombination of YCoCg-R samples on the GPU

In order to deal with the limitations mentioned above, the basic implementation was extended to rely entirely on the GPU to perform the remaining chroma sample recombination and actual color space transformation.

For a given output pixel location (x, y), the following steps need to be executed in order to reconstruct the chroma samples.

1. Determination of the pixel's interval and position therein. Both can be determined solely based on the x-coordinate.

$$interval_{x,y} = \left\lfloor \frac{x}{8} \right\rfloor \quad (2)$$

$$bit_{x,y} = x \ (mod\ 8) \quad (3)$$

2. Texture fetch of the least significant byte (LSByte) of the Co and Cg samples. Both chroma channels employ a single coordinate, defined in (4).

$$coord_{LSByte,x,y} = (x + interval_{x,y}, y) \quad (4)$$

3. Texture fetch of the Co and Cg bit collectors, using (5).

$$coord_{collectors,x,y} = 8 + 9\ interval_{x,y} \quad (5)$$

4. Bit extraction based on the collector and pixel position. Essentially, this step implements (6) (with C_i being either Co or Cg). The GPU approach, however, will be somewhat different and is discussed later.

$$MSB_{C_i,x,y} = (collector_{C_i,x,y} \gg bit_{x,y}) \,\&\, 1 \quad (6)$$

5. Recombination of the least significant byte with the most significant bit, resulting in reconstructed chroma samples. This final step performs the calculation in (7).

$$C_{i,x,y} = MSB_{C_i,x,y} + (LSByte_{C_i,x,y} \ll 8) \quad (7)$$

A single pixel shader is executed for each pixel in the transformed image and an identical code path is followed on each invocation.

The data flow in this extended implementation, depicted in Figure 4, is much simpler. A new Producer object reads data from disk directly into a texture surface in video memory. Profiling shows that this operation is essentially a repeated *movsd* instruction, albeit used with some nuance; since the actual width (or pitch) of a texture surface can be larger than the requested texture size, frames are copied scanline per scanline, instead of in a single operation. All textures, both luma and chroma, contain single byte elements (i.e., the D3DFMT_L8 texture format) and no memory and bus bandwidth is wasted.

4.1 Extended vertex processing

The first step of the recombination, the calculation of a pixel's interval and bit index, could be performed entirely in the pixel shader itself. However, a more elegant solution can be devised by extending the geometry and vertex shading, thereby profiting from a number of inherent graphics pipeline principles.

The preliminary implementation employs a single quad[1] (rasterized as two rectangles) of four vertices as underlying geometry to which the entire transformed image is mapped. Each vertex is assigned a single texture coordinate, as illustrated in Figure 5. The upper left pixel of

[1] Since the GPU rasterizes in terms of triangles, each quad has to consist of two triangles. Both triangles will share two of the four vertices, as is also illustrated in Figure 5.

the image is mapped to texture coordinate (0,0), the lower right pixel to coordinate (1,1) and all other pixels are interpolated between these extremes.

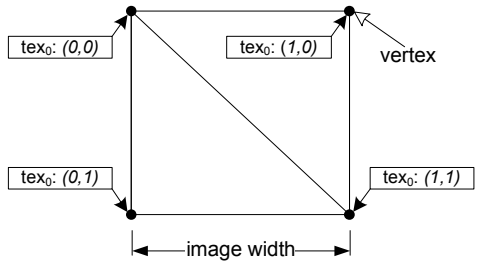

Figure 5. The original geometry, consisting of a single quad and texture coordinate

The novel approach uses a quad per pixel interval (i.e., one quad for every 8 pixels wide vertical strip) instead of one spanning the entire image. Each vertex is also assigned an additional texture coordinate, indicating the interval represented by that vertex. This new geometry is illustrated in Figure 6.

The additional texture coordinate can easily be generated by means of a vertex shader. Since the number of vertices to be processed is substantially lower than the number of pixels to be rasterized, processing time can be saved by moving functionality from pixel shading to the vertex processing phase. Below, the new vertex shader is listed. The input vertex position and texture coordinate, both calculated by the Persephone renderer at startup, are passed through to output. The additional second texture coordinate is used to store the vertex's interval and is also returned. This vertex shader requires only five instructions, two multiplications and three trivial moves.

```
VS(float4 pos: POSITION, float3 tex0: TEXCOORD0) {
    VSOutput_2tex outv;
    outv.pos = pos;
    outv.tex0 = tex0;
    outv.tex1 = float2(tex0.x * uvImageWidth/8.0, 0);
    return outv;
}
```

For every rasterized pixel, the GPU will interpolate the exact texture coordinate, using the coordinates assigned to the vertices of the triangle. This interpolation, inherent to the graphics pipeline and essentially without additional cost, provides the pixel shader with exactly the information it requires. The interpolated texture coordinate will contain the pixel's interval as its integer part. The fractional part of that texture coordinate represents the position of the pixel in the interval. This extended geometry and vertex processing approach reduces the pixel shader by two instructions.

4.2 Bit lookups

Since current generations of GPUs have no support for bitwise integer operations, such as AND or bit shifts, there is no way to carry out the fourth step of the chroma recombination, i.e., the extraction of collector bits, using an elegant instruction set.

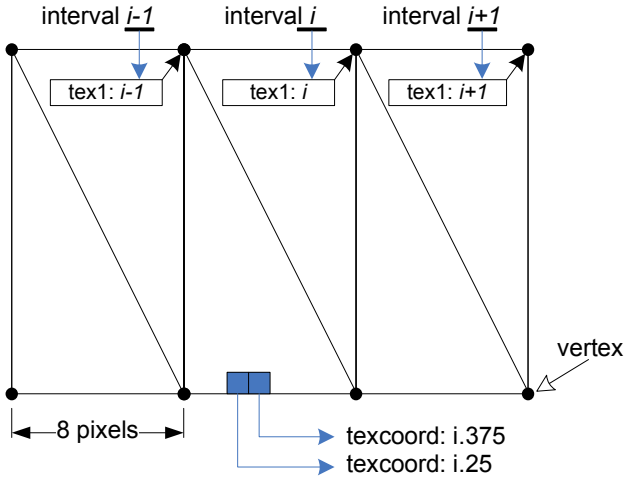

Figure 6. New geometry consists of different quads and adds a texture coordinate

In order to determine the value of a bit, two approaches can be taken. The first option is to arithmetically calculate the value of the bit (indexed starting from least significant bit 0) of a value x, as follows.

$$bit(x,i) = round\left(frac\left(\frac{x}{2^{i+1}}\right)\right) \quad (8)$$

The pixel shader can use (8) to determine the correct bit value. Notice that such a function can process both chroma collectors concurrently.

```
float2 lookupBit(float2 input, float bit) {
    float t = exp2(floor(8*bit)+1);
    return round(frac(input/t));
}
```

A Lookup Table (LUT) can serve as an alternative to the arithmetic bit calculation. LUTs are commonly used in GPU image processing operations such as color corrections [6]. Their use avoids expensive calculations of a (preferably continuous) function for each element in its domain, by storing discrete pairs of elements and respective function values. Figure 7 illustrates the LUT used in this case. The LUT is implemented as a texture of 256 by 8 elements and is filled according to equation 8. As such, the table contains one row of elements for each possible byte value, i.e., $2^8 = 256$ combinations.

The bit collector serves as the first coordinate of the LUT. The bit index obtained from the vertex shader serves as the other. As this interpolated fractional value lies, naturally, within the [0,1] interval, it can be employed immediately for addressing the LUT.

The lookup table incurs an additional texture read, but considering the limited size of the table (2048 entries) a texture cache possibly handles most reads directly. However, if the arithmetic function compiles (and optimizes) to a limited number of instructions, the actual calculation

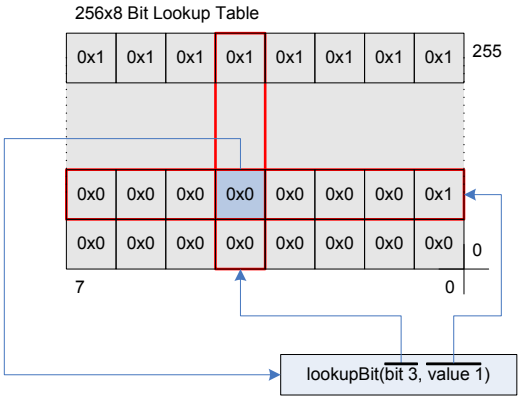

Figure 7. Retrieval of the fourth bit from byte value 1, by means of a Lookup Table

could still be cheaper than a texture memory fetch, especially in cases where the texture access misses the cache. Both approaches are evaluated in Section 5.

Once the correct most significant bits are obtained for the Co and Cg channels, by means of calculus or the LUT, these values are multiplied by 256 (i.e., the equivalent of shifting 8 positions to the left). This product is added to the previously fetched least significant chroma bytes in order to complete the reconstruction of the chroma samples.

4.3 The pixel shader

The implementation of all steps in the chroma sample recombination, including the optimizations from Sections 4.1 and 4.2, results in the pixel shader listed below. This is the shader version that uses the LUT. The arithmetic shader is identical except for the use of the `lookupBit` function from Section 4.2, for assigning `co9thbit` and `cg9thbit`. The LUT shader compiles to 38 instructions, while the arithmetic shader compiles to 45 instructions. This is roughly twice the length of the preliminary shader.

When reading the samples from texture memory and rescaling to the [0,255] range, a small corrective factor is added to ensure that no floating point input values are less than their nearest integer value. This avoids incorrect rounding and image artifacts in the transformed image, caused by *floor* operations in the color space transformation. The statement before returning the result limits all color channels to [0,255]. This is an equivalent of the required bit masking operation mentioned in Section 2. Due to Direct3D-specific texture addressing, the collector coordinate needs to be augmented by a half pixel in order to exactly map the texture elements to output pixels.

```
float4 PS_11(VSOutput_2tex inp) : COLOR {
    float t;
    float3 r = 0; ycocgr = 0;
    float3 f = {255,255,255}, e = {0.5,0.5,0.5};
    float collectorco, collectorcg;

    float interval = floor(inp.tex1);
    float bit = frac(inp.tex1.x);

    float2 coord = float2(
        (inp.tex0.x+interval.x/uvImageWidth)/9*8,
        inp.tex0.y
    );

    ycocgr.x = tex2D(ySampler, inp.tex0);
    ycocgr.y = tex2D(uSampler, coord);
    ycocgr.z = tex2D(vSampler, coord);

    ycocgr = ycocgr * f + e;

    float2 collectorcoords =
        float2(
            (8.5+interval*9)/(uvImageWidth/8.0*9.0),
            inp.tex0.y
        );

    collectorco = tex2D(uSampler, collectorcoords);
    collectorcg = tex2D(vSampler, collectorcoords);

    float co9thbit =
        tex2D(lookupSampler, float2(bit, collectorco));
    float cg9thbit =
        tex2D(lookupSampler, float2(bit, collectorcg));

    ycocgr.yz += float2(co9thbit, cg9thbit) * 256;

    // Actual YCoCg-R color space transform:
    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)*(256.0/255.0);
    return float4(r,1);
}
```

The correctness of this shader (and the rest of the implementation), was verified using an exhaustive test, comparing all permutations of input values to the transformed output from the GPU.

5 Performance measurements

The performance of the GPU-based transformation was tested on two machines; an AMD Athlon XP 2800+-based system with AGP-bus and an Intel Pentium D 820-based PC equipped with PCI-Express. Two GPU boards were used in each test system: an NVIDIA FX5900 and 6800 in the Athlon, and an ATI X600 and X1900 in the Pentium. At the time of writing, the FX5900 and X600 can be considered lower-end GPUs, the 6800 is midrange and the X1900 is a high-end chip. The test consisted of the upload, transformation and visualization of 300 YCoCg-R pictures. All pictures were buffered in memory in order to avoid suffering from a hard disk throughput bottleneck. The picture material used for this test was the high definition VIPER test sequence, which was especially made available by FastVDO for high-quality experiments.

The performance results are listed in Table 1 and Table 2. While the basic GPU implementation outperforms the CPU version in all situations, the same can not be said about the new approach. The extra burden placed on the GPU loads the X600 and FX5900 to their maximum processing capacity. The reduction in frame rate compared to both the CPU-only and first GPU implementation shows

that both lower-end chips form the bottleneck in recombining and transforming YCoCg-R frames. The CPU spends more time waiting for the GPU to finish processing than performing useful tasks. In both cases, the first GPU implementation offers a better balance between CPU and GPU workload. The FX5900 favors the arithmetic calculation of bit values, while the X600 is faster looking up those values from the LUT. As a matter of fact, the arithmetic bit calculation on the X600 suffered from artifacts due to its lower internal precision of 24 bits (all other chips support a 32-bit floating data type). This phenomenon was not observed in the faster LUT implementation, where the transformation was performed in a lossless fashion.

The tests performed using the 6800 and X1900 clearly benefit from the greater processing power of these higher-end chips. While the CPU still spends 25% time waiting for the 6800, the X1800 processes a frame in the time the next frame is being uploaded, forming a balanced division between CPU and GPU work. The LUT and arithmetic bit calculation perform almost identical on both higher-end GPUs.

The fact that the new implementation runs slower on the low-end chips does not deny a benefit for the CPU. If targeting real-time applications, this implementation still leaves more spare CPU time, as long as frames can be transformed within real-time constraints (typically between 30 fps and 60 fps) by the GPU. Instead of spin-locking the driver until the GPU is ready, the application can poll the GPU state at regular intervals while performing useful tasks in between.

Table 1. Performance results on Athlon XP 2800+

GPU	Resolution	CPU-only (fps)	GPU Basic (fps)	GPU New (LUT) (fps)	GPU New (fps)
FX5900	768x576	75.2	**148.2**	64.8	87.2
	1024x576	55.0	**109.4**	47.8	65.4
	1280x720	35.4	**70.3**	30.8	41.2
	1920x1080	15.7	**30.3**	13.7	18.9
6800	768x576	75.3	149.7	**247.9**	247.5
	1024x576	54.9	110.1	**188.9**	188.8
	1280x720	35.6	70.8	**116.2**	114.6
	1920x1080	15.9	31.5	**49.5**	48.9

Table 2. Performance results on Pentium D 820

GPU	Resolution	CPU-only (fps)	GPU Basic (fps)	GPU New (LUT) (fps)	GPU New (fps)
X600	768x576	137	**150**	110	97.6
	1024x576	104	**113.5**	83	73
	1280x720	67	**77**	53.5	47
	1920x1080	29.7	**34**	24	21.5
X1900	768x576	137	160	**304.5**	302.8
	1024x576	104	122	**238**	236.2
	1280x720	67	78.5	**155**	150.7
	1920x1080	29.7	36.4	**73.5**	69.8

6 Conclusions and future work

In this paper, an improvement of the previous implementation of GPU-assisted YCoCg-R color space transformation was demonstrated. The entire recombination and processing of YCoCg-R samples is now performed by the GPU, resulting in faster processing and double speed in specific cases. Aside from transferring image bytes from file or a memory buffer to the graphics memory, the CPU remains available for other decoding tasks. It must be noted that this new implementation favors fast GPUs and is not necessarily the best option for older GPUs in terms of pure processing speed. The successful realization of the chroma sample recombination proves that current GPUs provide the necessary flexibility to handle nontrivial data layouts. However, the limited and specific processor instruction set often requires using non-straightforward techniques and optimizations. Future work will focus on the implementation of other common video decoding steps (e.g., motion compensation) on the GPU, within the context of H.264/AVC and lossless encoding solutions. Attention will also be paid to further development of the Persephone Engine.

Acknowledgements

The research activities as 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.

References

[1] G. Sullivan, P. Topiwala, and A. Luthra, The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions. *Proceedings of SPIE annual meeting 2004: Signal and Image Processing and Sensors*, volume 5558, Denver, USA, 2004, 454-474.

[2] G. Shen, G. Gao, S. Li, H.-Y. Shum, and Y.-Q. Zhang, Accelerate Video Decoding with Generic GPU. *IEEE Transactions on Circuits and Systems for Video Technology*. 15 (5), 2005, 685–693.

[3] W. De Neve et al., GPU-Assisted Decoding of Video Samples Represented in the YCoCg-R Color Space, *Proceedings of the 13th ACM International Conference on Multimedia*, Singapore, 2005, 447–450.

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

[5] R. Duncan, A Survey of Parallel Computer Architectures, *IEEE Computer*, 23 (2), 1990, 5–16.

[6] J. Selan, Using Lookup Tables to Accelerate Color Transformations, in M. Pharr (Ed.), *GPU Gems 2* (Boston, Addison Wesley, 2005) 381–392.