# Motion Compensation and Reconstruction of H.264/AVC Video Bitstreams using the GPU

Bart Pieters, 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 bus 201, B-9050 Ledeberg–Ghent, Belgium
Email: {bart.pieters, dieter.vanrijsselbergen, wesley.deneve, rik.vandewalle}@ugent.be

***Abstract*— Most modern computers are equipped with powerful yet cost-effective Graphics Processing Units (GPUs) to accelerate graphics operations. Although programmable shaders on these GPUs were designed for the creation of 3-D rendering effects, they can also be used as generic processing units for vector data. This paper proposes a hardware renderer capable of executing motion compensation, reconstruction, and visualization entirely on the GPU by the use of vertex and pixel shaders. Our measurements show that a speedup of 297% can be achieved by relying on the processing power of the GPU, relative to the CPU. As an example, real-time playback of high-definition video (1080p) was achieved at 62.0 frames per second, consuming only 68.2% of all CPU cycles on a modern machine.**

## I. Introduction

Nowadays, consumers expect to be able to decode high-definition video sequences on their personal computers, though even the newest CPUs have difficulties achieving this. With the H.264/AVC standard, high-quality video can be compressed at low bitrates with a minimal impact on the visual quality. However, a significant amount of computational power is required to achieve this task. Most modern PCs have a powerful floating-point co-processor at hand, located on the graphics card. These GPUs, as they are called analogously to the CPU, can be used to offload some of the tasks from the CPU. They can execute parts of the decoding process parallel to the CPU. However, the GPU lies dormant in most cases when decoding video. This paper aims at showing how to activate the GPU to decode part of an H.264/AVC video bitstream.

The remainder of this paper is organized as follows. Section II briefly addresses previous work done in this rather new research area. Section III provides an introduction to Motion Compensation (MC) and reconstruction in the H.264/AVC standard. Sections IV and V give an introduction to the GPU architecture and how to accomplish MC using the GPU. Sections VI and VII show the results and limitations of our proposed solution. Finally, Section VIII concludes this paper.

## II. State-of-the-Art

The need for offloading MC from the CPU has been addressed in [1] for the proprietary WMV-8 codec from Microsoft Corporation. In the presented profiling, MC and Color Space Conversion (CSC) together consumed more than 61% of the total decoding time. A similar behavior has been observed for the H.264/AVC standard in [2]. The desired goal is to make CPU and GPU work in a pipelined fashion so that the GPU can perform the MC, reconstruction, and CSC, while the CPU focuses on Variable Length Decoding (VLD), Inverse Quantization (IQ) and Inverse Transformation. Achieving this objective will result in a significant speed-up of the decoding process, as long as the setup cost of the GPU is small. Fig. 1 shows this design as proposed by Shen *et al.* in [1].

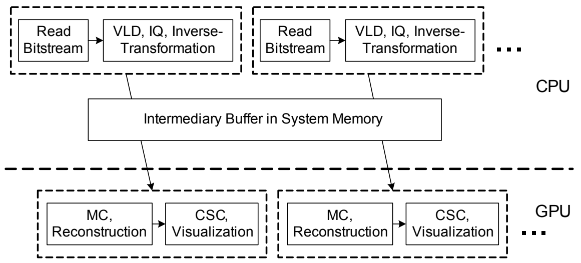

Fig. 1. GPU and CPU decoding video in a pipelined fashion[1].

MC and reconstruction on the GPU have already been implemented for Microsoft's proprietary WMV-8 in [1]. One of our goals is to investigate whether the same results can be achieved for the more complex and higher-demanding H.264/AVC standard [2].

## III. Motion Compensation in H.264/AVC

H.264/AVC divides pictures into macroblocks using three iterative steps (tree-structured MC). First, into macroblocks, these macroblocks are divided in macroblock partitions and these are in their turn divided in sub-macroblock partitions. To eliminate redundancy, these macroblocks can be predicted either by intra or inter prediction. Macroblocks are grouped together to form slices, which are parts of the picture that can be decoded independently. In this paper, we suppose that a picture consists of one slice.

Intra prediction uses pixels from previously decoded adjacent macroblocks in the same slice as a basis for prediction. These adjacent macroblocks can be encoded as either intra-predicted or inter-predicted macroblocks.

Inter prediction uses decoded pictures present in the Decoded Picture Buffer (DPB) to predict macroblocks. One possibility is to predict the macroblock based on up to one of 16 previously decoded reference pictures. An index is used to choose the desired reference picture from the Reference Picture List, together with a motion vector of a particular precision to retrieve the correct prediction. This type of macroblock is called a P macroblock. A slice containing only P macroblocks is called a P slice.

MC can be performed at integer pixel level and sub-pixel level (e.g. half-pel and quarter-pel), hence with different pixel interpolation strategies. The calculation of half-pels for instance uses a 6-tap interpolation filter. In one interpolation strategy, this filter is used six times to calculate a middle half-pel, which implies that more than 36 samples may be used in this process. These interpolations are therefore costly in term of CPU usage. They are the main reason why MC is a time-consuming process for a decoder. For more information regarding MC in H.264/AVC, the reader is referred to [3].

## IV. Representing Video on the GPU

The GPU uses raw geometry data, vertices and triangles, positioned in a 3-D space. A triangle is formed by three vertices. Two triangles form a quad. These vertices are transformed - e.g., projected on a plane - and then converted to a pixel-based raster. The latter is called the rasterization phase and establishes the values of the pixels. These values - e.g., colors - can be either calculated or looked-up in memory blocks in GPU memory called textures. The resulting colored pixels are finally written to a screen buffer and can subsequently be viewed on a screen. Alternatively, the results can be rendered to a texture. This makes it possible to use previous results in successive render passes.

Both vertex transformation and pixel rasterization are programmable since Direct3D 8.1, an API from Microsoft used to program the GPU. This makes it possible to use the GPU for advanced custom calculations, done by small programs, called shaders that run on the GPU. Vertex transformation and pixel rasterization are done by vertex and pixel shaders respectively.

Fig. 2 shows how this architecture can be used to manipulate video. A grid of vertices is constructed conform to the macroblock structure of the video, aligned on a 2-D plane. Every macroblock, macroblock partition and sub-macroblock partition is represented by four vertices forming a quad. The vertices have texture coordinates related to their position in 3-D space. All reference pictures are resident in GPU memory as textures. A pixel shader is used to fill the quads with pixels and pixel values are calculated from looked-up texture elements (texels) using the texture coordinates of the vertices. This way, the top left quad is filled with pixels from the top left of the reference picture for example.

## V. H.264/AVC Motion Compensation on the GPU

### A. Proposed Design

By using the GPU combined with Direct3D, we can execute our solution on all hardware supporting Direct3D and certain shader models. The obtained flexibility can for instance be used to apply advanced custom macroblock error correction methods.

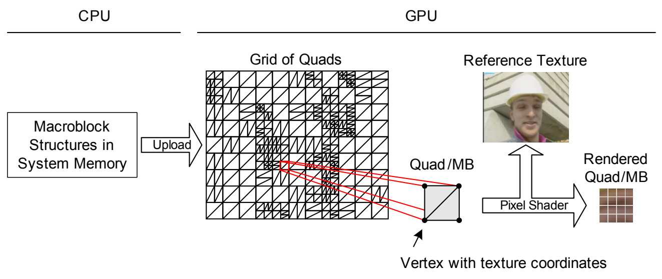

Fig. 2. The GPU architecture used in manipulating video.

Our implementation starts at the Intermediary Buffer in Fig. 1, containing macroblock descriptions, motion vectors and residual data. Fig. 3 shows the operational work flow. In the Figure, textures (T$x$) are used as reference textures in render passes. The result is a new texture composed by a pixel shader. The work flow for luminance pictures is organized as follows.

1) A grid of quads is constructed representing the macroblock structure of the video picture as addressed previously. All four vertices of the quads contain duplicates of the motion vector of the corresponding macroblock or (sub-)macroblock partition (as vertex data are private).
2) A vertex shader translates the texture coordinates of all vertices according to their stored motion vector. This way, full-pel MC is accomplished.
3) The motion-compensated picture is rendered in 6 render passes as shown in Fig. 3, corresponding with the different interpolation strategies, based on the reference picture texture (T1). The texture used to hold the motion-compensated picture (T2) is filled each render pass with certain quads selected by a vertex shader. Fig. 4 illustrates this process. For each interpolation technique, the following is done:
   - a vertex shader displaces all quads that do not need to be filled with the selected interpolation technique;
   - a pixel shader (PS1..6) is selected according to the desired interpolation technique to fill all remaining quads. Multiple pixel shaders are used as few pixel shader instruction slots are available.
4) Meanwhile, the CPU has uploaded the residual data to a texture (T3) on the GPU. The motion-compensated picture is selected as a reference texture to start the reconstruction phase. A pixel shader (PS7) renders a single quad with the dimension of the video, using the motion-compensated picture and the uploaded residual data as textures. The two are added together to create the reconstructed picture (T4), which remains in GPU memory.
   A similar process is done for the chroma pictures using only one interpolation strategy, resulting in two textures containing the recontructed chroma pictures.
5) CSC is accomplished by using the reconstructed pictures in a render pass with a pixel shader (PS8) that transforms all pixel values. The result is presented on screen.

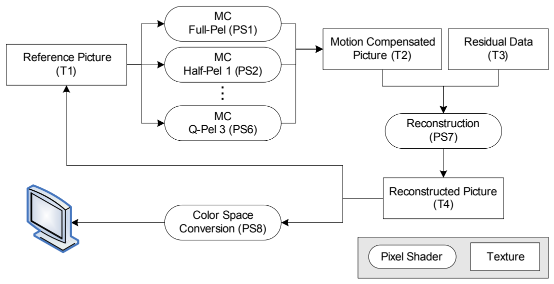

Fig. 3. Flow chart of textures and pixel shaders.

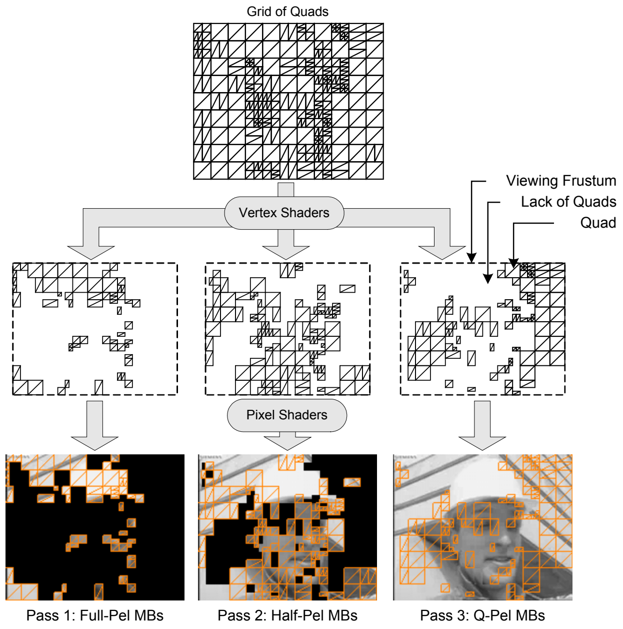

Fig. 4. Using vertex shaders to select the required macroblocks for a specific pixel shader. In practice, 6 render passes are needed.

## *B. Sub-Pixel Motion Compensation*

As illustrated in Fig. 4, the motion compensated luminance picture is established over six render passes. Each of these passes fills a part of the picture with a different interpolation technique, calculated by pixel shaders. By dispositioning quads out of the viewing frustum, the GPU is smart enough to exclude them from the rendering phase. This way, the CPU is relieved of the burden of deriving motion vector types and grouping similar quads. In case of chroma pictures, only one pass for each picture is required as H.264/AVC only uses one type of interpolation.

Pixel shaders execute the correct interpolation algorithm for calculating sub pixels. Attention was paid to achieving drift-free MC by ways described in [4]. The GPU shader architecture provides instructions working on floating-point numbers while video decoding depends on integer calculations. Hence, integer calculations were simulated by introducing extra rounding instructions.

TABLE I
PERFORMANCE RESULTS

| System and | | CPU Renderer | | GPU Renderer | |
|---|---|---|---|---|---|
| Graphics Card | resolution | fps | cycles (%) | fps | cycles (%) |
| Intel | 720x480 | 79.8 | 98.3 | 155.9 | 39.5 |
| Pentium D 930 | 1280x720 | 36.9 | 98.1 | 71.2 | 37.2 |
| GeForce 6800 GT | 1920x1080 | 13.4 | 97.2 | 27.6 | 48.6 |
| Intel | 720x480 | 83.6 | 98.5 | 299.5 | 75.2 |
| Pentium D 950 | 1280x720 | 35.5 | 98.1 | 160.7 | 75.5 |
| GeForce 7800 GTX | 1920x1080 | 15.6 | 98.7 | 62.0 | 68.2 |

## *C. Reconstruction of the Motion-Compensated Pictures*

After inter prediction, a texture is resident in GPU memory containing the motion-compensated picture. In a next step, the motion-compensated picture needs to be reconstructed by adding residual data to all pixels. This is again a process that is very well suited for the GPU.

Residual data contain values in the range [-255..255] in order to correct all predicted 8-bit pixels. This means 9-bit values need to be stored in GPU memory. The GPU however does not support any texture formats efficiently storing 9-bit values. To resolve this issue, 16-bit textures where used where each texel holds a single 9-bit value. This approach may seem inefficient as 7 bits are wasted, but requires very little CPU processing time for data preparation. Indeed, today's PCI-Express graphics bus is powerful enough to deliver the data in time.

## *D. Intra Prediction*

This paper focuses primarily on inter prediction. Intra prediction is done by the CPU. Therefore, all predicted intra macroblocks need to be transferred to the GPU. This is accomplished by packing reconstructed intra macroblocks in the residual data that are to be uploaded to the GPU. This way, no extra render pass is needed to draw intra-predicted macroblocks. That is, if the texture to receive the motion-compensated picture is cleared to zero first. Another advantage is that no vertices describing an intra macroblock need to be uploaded to the GPU.

## VI. RESULTS

For our implementation, the DirectX9c API was used to program the GPU. All shaders were written in the High Level Shader Language (HLSL) in the DirectX Effects Framework. The performance of the implemented renderer only utilizing the GPU was compared to that of an own-developed renderer, only using the CPU. The results of both GPU and CPU renderer are shown in Table I. A video sequence containing one I slice and 127 P slice coded pictures was looped 10 times and the frame rates were noted. The P slice coded pictures used one reference picture and one of two CPU cores was disabled. Video sequences used where driving, shields and blue sky[1].

The results show that the GPU renderer outperforms the CPU renderer with a factor of up to 3.9, rendering 62.0 1080p frames per second (fps) on an Intel Pentium D test machine

[1]For a demo, the reader is referred to *http://multimedialab.elis.ugent.be/gpu/*

TABLE II
PERFORMANCE RESULTS OF GENERIC VERTEX GRID

| System and Graphics Card | Resolution | CPU Renderer fps | CPU Renderer cycles (%) | GPU Renderer fps | GPU Renderer cycles (%) |
|---|---|---|---|---|---|
| Intel | 720x480 | 79.8 | 98.3 | 80.7 | 32.4 |
| Pentium D 930 | 1280x720 | 36.9 | 98.1 | 33.4 | 30.2 |
| GeForce 6800 GT | 1920x1080 | 13.4 | 97.2 | 13.9 | 48.4 |
| Intel | 720x480 | 83.6 | 98.5 | 169.6 | 57.7 |
| Pentium D 950 | 1280x720 | 35.5 | 98.1 | 64.0 | 53.3 |
| GeForce 7800 GTX | 1920x1080 | 15.6 | 98.7 | 27.9 | 47.5 |

with an NVIDIA GeForce 7800 GTX. This enables decoding of H.264/AVC-sequences of 1080p frames in real time.

More important is the CPU processing power required by both renderers. The table shows how the frame rates by the GPU renderer were achieved with relativly low CPU activity. When processing 720p pictures, only 75.5% of all available CPU cycles were needed. The CPU renderer used, evidently, up to 98.1% of all CPU cycles. If we limit the output of the renderer to 60 fps, the expected frame rate for real-time high-definition video, the amount of CPU cycles the GPU renderer consumes drops to 27.1%. This means the GPU renderer can successfully take over more than half of the decoding process with low CPU demand [2].

### *A. Influence of Motion Complexity*

The amount of data to upload to the GPU is correlated to the complexity of motion in the video sequence. The more motion in a video sequence, the higher the subdivisions in macroblock partitions and sub-macroblock partitions. This means more macroblock descriptions and motion vectors are to be uploaded to the GPU. To examine the influence of motion complexity, a second GPU-based renderer has been developed. This renderer uses a vertex grid that represents a video picture composed of only sub-macroblock partitions. This means that, for example, a macroblock will be represented in GPU memory by 16 quads, all containing duplicates of the original motion vector. This generic vertex grid is uploaded once, and every next picture, motion vectors in the vertex grid are updated. Consequently, the amount of data to be uploaded is invariant. Hence, this renderer implements a worst-case upload scenario.

Table II shows the results when using the generic vertex grid with the video sequences that were used in the previous experiment. The results show that even in this case, performance gains are significant.

## VII. LIMITATIONS TO THE CURRENT DESIGN

There are a number of limitations to our current design. The CPU must be able to predict all intra macroblocks independent of P-macroblocks, as the latter are located on the GPU. Read back of these pixels could compromise the pipelined fashion in which CPU and GPU work together. By using Constrained Intra Prediction (CIP) during encoding of the video, the problem can be avoided. The ideal solution would be to predict the intra macroblocks on the GPU. However, intra prediction does not translate well into a GPU model. In this prediction mode, each pixel is dependent on pixels above and to the left of it. This makes it difficult to calculate pixels in parallel and does not allow an efficient solution only using the GPU. Future research will focus on removing this constraint.

Another limitation in the current implementation is the inability to execute the MC process with more than one reference picture. A minor performance drop is expected if more reference pictures are used as more render passes to compose the motion-compensated picture become necessary. To minimize the impact, the CPU can arrange constructed quads in the buffer such that quads are never rendered multiple times with different reference pictures.

Finally, no in-loop deblocking filter is used. Just as intra prediction, this filter does not translate well in a GPU-enabled design because of inter-pixel dependences. In practice however, the deblocking filter is mostly turned off in case of high-quality, high-definition video (as blocking effects are minimal).

## VIII. CONCLUSIONS AND FUTURE WORK

We achieved motion compensation, reconstruction and visualization in real time for high-definition video (1080p) on a PC with an Intel Pentium D 950 and an NVIDIA GeForce 7800 GTX. Our implementation uses significantly less CPU cycles than a CPU-only based solution (68.2% vs. 98.7%), while achieving higher frame rates (62.0 fps vs. 15.6 fps). Fundamental issues and limitations of the architecture of the GPU were identified and discussed. A significant speed-up may be expected when the developed design is integrated in an H.264/AVC decoder compared to a CPU-only based solution.

With this design and implementation, the first steps were set to develop an H.264/AVC decoder, utilizing the GPU through Direct3D. Future work will focus on integration of the renderer in a decoder and GPU-assisted decoding of scalable video content.

## ACKNOWLEDGMENTS

The research as described in this paper was 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. Shen, G. Gao, S. Li, H. Shum, and Y. Zhang, "Accelerate Video Decoding With Generic GPU," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 5, pp. 685–693, May 2005.

[2] V. Lappalainen, A. Hallapuro, and T. D. Hämäläinen, "Complexity of Optimized H.26L Video Decoder Implementation," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 717–725, July 2003.

[3] G. J. Sullivan and T. Wiegand, "Video Compression - From Concepts to the H.264/AVC Standard," *Proc. the IEEE, Special Issue on Advances in Video Coding and Delivery*, vol. 93, no. 1, pp. 18–31, January 2005.

[4] D. Van Rijsselbergen, W. De Neve, and R. Van de Walle, "GPU-driven Recombination and Transformation of YCoCg-R Video Samples," in *Proc. of the Fourth IASTED International Conference*, J. Silva-Martinez, Ed., no. 531. Anaheim, Calgary, Zurich: ACTA Press, 11 2006, pp. 21–26.

[5] G. Sullivan, P. Topiwala, and A. Luthra, "H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions," in *Proc. of SPIE annual meeting 2004: Signal and Image Processing and Sensors*, vol. 5558, Denver, USA, 2004, pp. 454–474.