Performance Evaluation of H.264/AVC Decoding and Visualization using the GPU

Bart Pieters[a], 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

ABSTRACT

The coding efficiency of the H.264/AVC standard makes the decoding process computationally demanding. This has limited the availability of cost-effective, high-performance solutions. Modern computers are typically equipped with powerful yet cost-effective Graphics Processing Units (GPUs) to accelerate graphics operations. These GPUs can be addressed by means of a 3-D graphics API such as Microsoft Direct3D or OpenGL, using programmable shaders as generic processing units for vector data. The new CUDA (Compute Unified Device Architecture) platform of NVIDIA provides a straightforward way to address the GPU directly, without the need for a 3-D graphics API in the middle. In CUDA, a compiler generates executable code from C code with specific modifiers that determine the execution model.

This paper first presents an own-developed H.264/AVC renderer, which is capable of executing motion compensation (MC), reconstruction, and Color Space Conversion (CSC) entirely on the GPU. To steer the GPU, Direct3D combined with programmable pixel and vertex shaders is used. Next, we also present a GPU-enabled decoder utilizing the new CUDA architecture from NVIDIA. This decoder performs MC, reconstruction, and CSC on the GPU as well. Our results compare both GPU-enabled decoders, as well as a CPU-only decoder in terms of speed, complexity, and CPU requirements. Our measurements show that a significant speedup is possible, relative to a CPU-only solution. As an example, real-time playback of high-definition video (1080p) was achieved with our Direct3D and CUDA-based H.264/AVC renderers.

Keywords: Direct3D, GPU, graphics hardware, H.264/AVC, motion compensation, CUDA, shaders, video coding

1. INTRODUCTION

Today, the high-pace business of the videogames market keeps pushing graphics card production and performance levels to the limits. With memory bus widths of up to 512 bit, high speed memory chips delivering up to 84 gigabytes per second, and high processor clock speeds, graphics cards deliver specifications never seen in consumer-level commodity hardware before. Graphics cards these days hold a Graphics Processing Unit, called the GPU. This is a powerful processor predominantly used in videogames to render realistic 3-D environments. Because of the very high parallelism and floating-point computational capability, performance levels are impressive with respect to the ability of the GPU to render realistic 3-D scenery with high dynamic color ranges. Along with these GPUs, the graphics bus, now PCI-Express, has evolved to a high-performance bus capable of delivering four gigabytes of data per second in full duplex.

While the first graphics cards provided a fixed function pipeline with no programmability at all, today's cards can be programmed using so-called shaders. Shaders are small programs running on the GPU, processing geometrical data or rasterizing 3-D scenes. The high-performance architecture and programmability of the GPU spurs scientists to seek out the GPU for a variety of computational tasks, so called General Purpose GPU (GPGPU) applications. However, the GPU has a distinct programming model. First, because of its original 3-D graphics purpose, programming the GPU is generally done by means of a 3-D graphics API, with Microsoft's Direct3D and OpenGL being the two most obvious ones. This implies that generic algorithms have to be casted in a graphics context, suitable for execution by a shader. Second, what the GPU is best at is processing a massive amount of uniform, lightweight shader programs without

[a] E-mail: bart.pieters@ugent.be, Telephone: +32 9 33 14985, Fax: +32 9 33 14896

Applications of Digital Image Processing XXX, edited by Andrew G. Tescher
Proc. of SPIE Vol. 6696, 669606, (2007) · 0277-786X/07/$18 · doi: 10.1117/12.733151

interdependencies, for example, element-wise additions. Without agreement to these two characteristics, processing speeds of implemented algorithms will see minor to no improvements over those of a CPU-based implementation.

With the last generations of GPUs, programmability has increased with the introduction of generic programming interfaces without the need for a graphics context. Examples are the ATI CTM (Close To the Metal) and NVIDIA CUDA (Compute Unified Device Architecture) platforms. These general purpose computing platforms allow users to directly address the processing power of the GPU, without the need for extensively rewriting the algorithms to a graphics context. While ATI CTM allows to directly access the GPU using assembler code, the NVIDIA CUDA platform introduces a C compiler, capable of compiling C code with platform specific annotations.

Most graphics cards also have powerful 2-D video processors on board designed to assist the system processor in decoding coded video streams. These chips are mostly vector processors specifically designed to accomplish a certain decoding task, for example, motion compensation. Although the latest 2-D video processors are programmable, lack of design flexibility and programming transparency makes users look for alternatives. H.264/AVC currently stands where the MPEG-2 Video standard was years ago, with gradual hardware support coming available to make commodity PCs able to decode H.264/AVC bitstreams. The question is whether dedicated on-board 2-D video decoders can provide enough flexibility to assist in handling today's and future complex decoding standards, while performance and flexibility of GPUs keeps advancing.

In this paper, we investigate whether GPU-based video decoders provide a more flexible solution than on-board dedicated 2-D hardware decoders for decoding H.264/AVC video bitstreams. We essentially introduce two GPU-enabled renderers capable of decoding part of an H.264/AVC bitstream. One renderer is based on Microsoft Direct3D, using the traditional programmable graphics pipeline, while the other one relies on the recently introduced NVIDIA CUDA programming framework.

The remainder of this paper is organized as follows. Section 2 provides an introduction to the H.264/AVC standard with an emphasis on the decoding phases we plan to execute on the GPU. Section 3 discusses the possibilities available for decoding H.264/AVC bitstreams using on-board dedicated 2-D video decoders. Section 4 briefly outlines previous work done in decoding video on the GPU. Section 5 introduces our own-developed Persephone framework used in the development of the presented accelerators. Section 6 and 7 each presents a GPU-based accelerator for decoding H.264/AVC bitstreams where the former section uses Direct3D and the latter section the new NVIDIA CUDA platform. In Section 8 the performance of both presented accelerators is compared and discussed. Finally, Section 9 concludes this paper and addresses future work.

2. DECODING H.264/AVC

This section introduces the H.264/AVC standard. All decoding steps are briefly discussed, emphasizing the decoding complexity of each step. Particular attention is given to the final decoding steps, i.e. Motion Compensation (MC) and reconstruction. Color Space Conversion (CSC) and visualization aspects are addressed as well.

2.1 H.264/AVC Overview

At the time of writing, H.264/AVC is the latest international video coding standard developed by the Joint Video Team (JVT), a collaboration of the Video Coding Experts Group (VCEG) of the ITU-T and the Moving Pictures Experts Group (MPEG) of ISO/IEC. By using state-of-the-art coding techniques, the standard provides enhanced coding efficiency and flexibility of use for a wide range of applications, including high-definition cinema. A trade-off between coding efficiency and implementation complexity was considered. The accomplished coding ratio has seen an average improvement of at least a factor two compared to MPEG-2 Video[1]. However, the state-of-the-art coding techniques used in this standard require a significant amount of processing power[2].

2.2 Decoding Steps

H.264/AVC divides pictures into slices, which are areas of the picture that can be decoded independently. Slices contain macroblocks with different macroblock subdivisions, determined using three iterative steps (tree-structured MC). First, slices are divided into macroblocks of 16x16 pixels. These macroblocks can be further divided into macroblock partitions of 8x16, 16x8, or 8x8 pixels. Finally, 8x8 macroblock partitions can in turn be divided in sub-macroblock partitions of 4x8, 8x4, or 4x4 pixels. It is clear that this tree-structured MC increases decoding complexity over MC techniques of previous video coding standards. For instance, a high-definition video sequence of 1920 by 1080 pixels may consist of 129 600 sub-macroblock partitions, which all have to be processed in real time. Decoding is done on a macroblock-by-macroblock basis.

2.2.1.1 Entropy Decoding, Inverse Quantization, and Inverse Transformation

The decoding process starts with parsing the bitstream and executing entropy decoding for the first macroblock received. H.264/AVC supports two advanced entropy coding methods. Both CAVLC (context-adaptive variable-length coding) and CABAC (context-adaptive binary arithmetic coding) use context-based adaptivity to offer more efficient entropy coding over previous standards. With CABAC being an arithmetic entropy coder, higher compression ratios can be accomplished, but at the expense of higher complexity and processing power requirements.

Using a per-macroblock quantization parameter, transform coefficients are acquired with inverse quantization (IQ). Afterwards, an inverse transformation similar to an Inverse Discrete Cosine Transform (IDCT) is executed. What makes this inverse transformation special is the use of 16-bit integer-based arithmetic, thereby avoiding inverse-transformation rounding errors on different hardware implementations. The use of integer-based arithmetic makes the inverse transformation more suited for fast execution on a consumer-level CPU than the use of floating-point-based arithmetic.

2.2.1.2 Prediction and Reconstruction

To eliminate redundancy in the spatial domain, macroblock sample values can be predicted either by intra or inter prediction. Intra prediction predicts sample values by using neighboring samples from previously-coded blocks in the current slice, located to the left and/or above the block that is to be predicted. Multiple intra prediction schemes are possible. A new feature not found in previous standards, like MPEG-4 Visual, allows intra prediction from neighboring areas that were not coded using intra coding. Macroblocks predicted using intra prediction are called I macroblocks.

Inter prediction uses decoded pictures present in the Decoded Picture Buffer (DPB) to predict macroblocks. A first possibility is to predict the macroblock based on one of up to 16 previously decoded reference pictures. An index is used to choose the desired reference picture from the Reference Picture List (RPL), together with a motion vector of a particular precision to retrieve the correct prediction. The macroblock predicted with this type of prediction is called a P macroblock. A slice containing only P and I macroblocks is called a P slice. A second possibility is to predict the macroblock based on two weighted averages of two motion-compensated prediction values. Slices that contain macroblocks with this type of prediction are called B slices. The pictures in the DPB are now organized into two distinct RPLs, *list 0* and *list 1*, and four types of predictions are possible: *list 0*, *list 1*, *bi-predictive,* and *direct.* MC can be performed at integer pixel level and sub-pixel level (e.g., half-pel and q-pel), hence with different pixel interpolation strategies. The calculation of half-pels for instance uses a 6-tap interpolation filter, as shown in Figure 1.

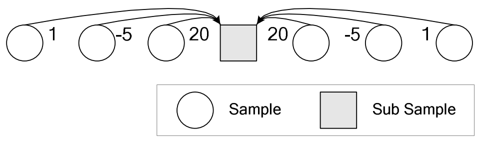

Figure 1: The 6-taps interpolation filter used in H.264/AVC.

After having executed intra or inter prediction, the obtained pixel values have to be corrected. This correction is done in the reconstruction phase where the residual data obtained from the inverse transformation are added to the prediction.

The sheer possibilities and number of intra and inter prediction modes prove the flexibility of the H.264/AVC standard. However, this flexibility also introduces additional implementation complexity and extra requirements concerning processing power.

2.2.1.3 Color Space Conversion and Visualization

The last step in order to display a decoded H.264/AVC video sequence is executing CSC and visualizing the result on the screen. CSC transforms the samples from the YCbCr color space to the RGB color space. Calculating this transformation involves per-pixel floating-point arithmetic.

Visualization implies that the current picture is uploaded to the memory of the graphics card, typically by use of either generic operating system calls (e.g. Windows GDI), DirectX Overlays, or the Direct3D Video Mixing Renderer (VMR). The Direct3D VMR is capable of using the YCbCr color space. This renderer uses the GPU to execute CSC. This means that YCbCr pixel values can be transferred instead of RGB pixel values. Subsampling is usually not applied in the RGB color space. This results in fewer bytes being uploaded to GPU memory.

2.3 Decoding Complexity

According to Lappalainen *et al.*[2], the most computationally-expensive tasks in H.264/AVC decoding and visualization comprise MC, reconstruction, CSC, and visualization, together claiming more than half of the processing time. These results are not surprising. For MC, this is partly because of the complex interpolation schemes used. For CSC, this is because of the per-pixel floating-point arithmetic used. As mentioned before, sub-pixel MC with half-sample and quarter-sample accuracy is realized by means of a 6-taps interpolation filter in H.264/AVC. With several interpolation strategies for both half-pels and q-pels, complexity increases further. For example, calculating a middle half-pel uses the 6-taps filter six times, accessing 36 samples, consequently loading the memory bus.

With the increasing popularity of high-definition video sequences, there is a greater need for sufficient processing power to decode these sequences.

3. DECODING H.264/AVC BITSTREAMS USING DEDICATED HARDWARE ON GRAPHICS CARDS

Graphics cards providing video decoding support for the H.264/AVC standard are already available. In recent graphics cards from manufacturer ATI, a programmable chip is present, named the Universal Video Decoding (UVD) engine. This chip is capable of decoding the entire bitstream, performing all decoding steps from entropy decoding to CSC. Graphics cards from NVIDIA have a similar decoding engine, which is a component of the PureVideo framework. How to use these decoding engines is discussed in the next two paragraphs.

Graphics hardware vendors have released software components delivering GPU acceleration to systems running Microsoft Windows. These components usually come as DirectShow[3] filters, generally accessible for all video decoding software using the DirectShow framework. These DirectShow filters communicate with the graphics card driver using a proprietary API. These API calls are then translated in some sort of micro code to be executed on the card.

With the new Windows Vista operating system, DirectX Video Acceleration[4,5] (DXVA) 2.0 adds support for H.264/AVC. DXVA is a Microsoft API specification for video decoding that utilizes 2-D hardware acceleration. The pipeline of DXVA enables software decoders to off-load certain CPU-intensive operations to hardware located on the graphics card. Typically off-loaded operations are IDCT, MC, and CSC. DXVA 1.0 includes support for key video coding formats like H.261, MPEG-1 Video, MPEG-2 Video, H.263, and MPEG-4 Visual. The second version of DXVA adds support for H.264/AVC decoding acceleration. Hardware support for the features of DXVA 2.0 comes with both NVIDIA and ATI hardware.

The use of on-board hardware video accelerators has a number of disadvantages. First, and already mentioned, the use of proprietary video acceleration APIs. No solutions are available that have reverse-engineered NVIDIA’s PureVideo or ATI’s AVIVO APIs. Second, DXVA 2.0 is only available for Windows Vista. Finally, but most importantly, these hardware solutions lack flexibility, as acceleration functions are partially hard-wired on the cards with no support for custom or less popular formats. As such, future video coding standards cannot be accelerated without requiring a new graphics card with hardware support specifically designed for that video standard.

To overcome these drawbacks, this paper will emphasize on hardware accelerated software decoders using more flexible GPU-based accelerators.

4. DECODING H.264/AVC BITSTREAMS WITH THE GPU: RELATED RESEARCH

Executing MC, reconstruction, and visualization using the flexibility of the GPU has been addressed by Shen *et al.*[6] for the proprietary WMV-8 codec of Microsoft Corporation. For WMV-8, MC and CSC together consumed more than 61% of the total decoding time. In the paper, the authors propose a decoding design to make CPU and GPU work in a pipelined fashion. In this design, the GPU performs MC, reconstruction, and CSC, while the CPU focuses on Variable Length Decoding (VLD), inverse quantization, and inverse transformation. Figure 2 shows the design as proposed by Shen *et al.* Achieving this objective for WMV-8, the authors report a significant speed-up of the decoding process of up to a factor of 3.16 for 720p video sequences.

One of our research goals is to investigate whether the same results can be achieved for the more complex and higher-demanding H.264/AVC standard, using both 3-D graphics APIs and newly introduced General Purpose GPU (GPGPU) programming platforms like NVIDIA CUDA. As such, one of the questions we would like to answer is whether the flexibility, transparency, and processing power of the GPU is sufficient to make it a complete alternative to dedicated 2-D video processors located on the graphics card.

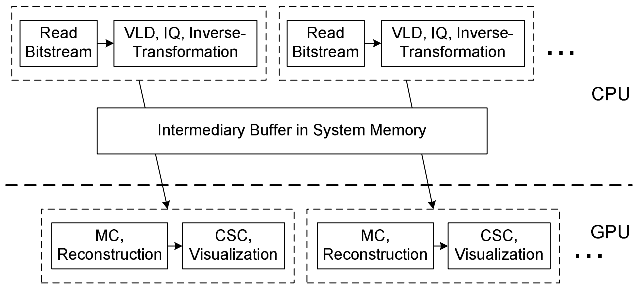

Figure 2: Decoding video bitstreams in a pipelined fashion as proposed by Shen *et al.*[6]

The use of the GPU for video encoding purposes for H.264/AVC has already been described by Ho *et al.*[7] Here, the 3-D graphics pipeline of the GPU was used to accelerate motion estimation, one of the most computationally complex phases in encoding a video sequence to a H.264/AVC bitstream.

5. THE PERSEPHONE FRAMEWORK

Our framework, called Persephone[b], allows easy utilization of accelerators in decoding video by construction of rendering pipelines. It is possible to use different software H.264/AVC decoders to perform part of the decoding process. The output of these producers, as they are called, is to serve as a basis for GPU-based producers. In their turn, the output of these producers can be the basis for GPU-accelerated renderers. Our plug-in system has been specifically designed for use with hardware-accelerated components with minimal overhead. Configuring and addressing the Persephone framework is done by XML-based pipeline descriptions. Figure 3 shows the design of the Persephone framework.

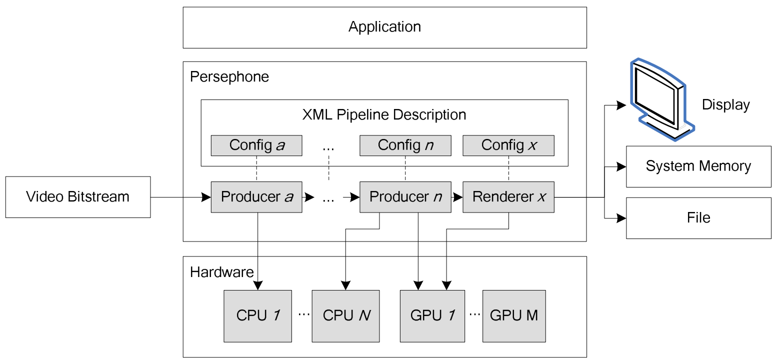

Figure 3: Persephone Framework.

This development framework significantly resembles with the DirectShow framework from Microsoft, and in particular, with the use of the Video Mixing Renderer architecture. However, the Persephone Framework lacks complexity on the one hand, and is designed specifically targeting GPU-based accelerators that require sharing of GPU resources on the other hand.

The two GPU-enabled H.264/AVC bitstream renderers presented in Section 6 and 7 are both integrated into Persephone.

[b] More information about the Persephone framework can be found on http://multimedialab.elis.ugent.be/GPU.

6. DECODING H.264/AVC BITSTREAMS WITH DIRECT3D

Our first GPU-based video accelerator for H.264/AVC decoding uses the Direct3D 3-D version 9c graphics API to steer the GPU. The implementation starts at the Intermediary Buffer in Figure 2, containing positions, dimensions, motion vectors, and residual data for all macroblocks, macroblock partitions, and sub-macroblock partitions.

6.1 Direct3D

Using the Direct3D graphics API[8], the GPU works on raw geometry data, i.e. vertices and triangles, which are positioned in a 3-D space. A triangle is formed by three vertices. Two triangles form a rectangle, called 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 buffers in the GPU memory. These buffers are known as 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, which makes it possible to use previous results in successive render passes.

Both vertex transformation and pixel rasterization are programmable since Direct3D 8.1. This makes it possible to use the GPU for advanced custom calculations, which are done by shaders that run on the GPU. Vertex transformation and pixel rasterization are done by vertex and pixel shaders respectively. Figure 4 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 in one texture. 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. By translating these texture coordinates, the source pixels for the pixel shader can be set.

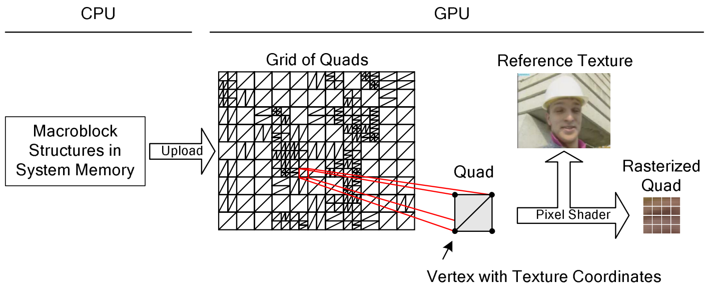

Figure 4: The GPU architecture used in decoding video bitstreams.

6.2 Proposed Design for Motion Compensation of P Slices

Using the GPU combined with Direct3D, we can execute our renderer on all hardware supporting Direct3D and shader model 2.0b. The obtained flexibility can, for instance, be used to apply advanced custom macroblock error correction methods.

Our implementation starts at the Intermediary Buffer in Figure 2, containing macroblock positions, dimensions, motion vectors and residual data. 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 in Section 6.1. The four vertices of each quad 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. Now the Direct3D pipeline is set-up to start rendering pixels.

3) Figure 5 shows the operational work flow for the rasterization phase. In Figure 5, textures (T*y*) are used as reference textures in subsequent render passes. The result is a new texture composed by a pixel shader. A motion-compensated picture is rendered in six render passes. Each render pass corresponds to a different interpolation strategy, using texture (T1) as a reference picture. Each render pass, the texture used to hold the

motion-compensated picture (T2) is filled a little bit more. This technique will be explained in further detail in Section 6.3.

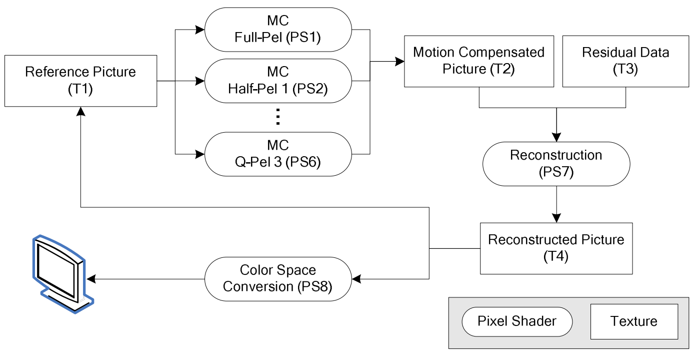

Figure 5: Flow chart of textures and pixel shaders for P slices.

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. The texture holding the reconstructed picture (T4) is sixteen times the size of one decoded picture. Therefore, up to 16 previously decoded pictures are stored for future motion compensation.
5) A similar process is done for the chroma pictures using only one interpolation strategy, resulting in two textures containing the reconstructed chroma pictures.
6) CSC is accomplished by using the reconstructed pictures in a render pass with a pixel shader (PS8) that transforms all pixel values; In case downsampling was used, the chroma pictures are upsampled. The result is presented on screen.

6.3 Sub-Pixel Motion Compensation

As previously mentioned, 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. Figure 6 illustrates this. By dispositioning quads out of the viewing frustum, the GPU will exclude them from the rasterization phase. This displacement is done by a vertex shader for quads that do not need to be filled with the selected interpolation technique. 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 is required for each picture; H.264/AVC only uses one type of interpolation for chroma pictures.

Pixel shaders execute the correct interpolation algorithm for calculating sub pixels. A pixel shader (PS1..6 in Figure 5) is selected corresponding to the desired interpolation technique to fill all remaining quads. Multiple pixel shaders are used as only a limited number of instruction slots are available for implementing a particular pixel shader and branching in pixel shaders can potentially stall the rendering pipeline.

Particular attention was paid to achieving drift-free MC by ways described by Van Rijsselbergen *et al.*[9] Also, the GPU shader architecture provides instructions working on normalized floating-point numbers while video decoding depends on integer calculations. Hence, integer calculations were simulated by introducing extra rounding instructions.

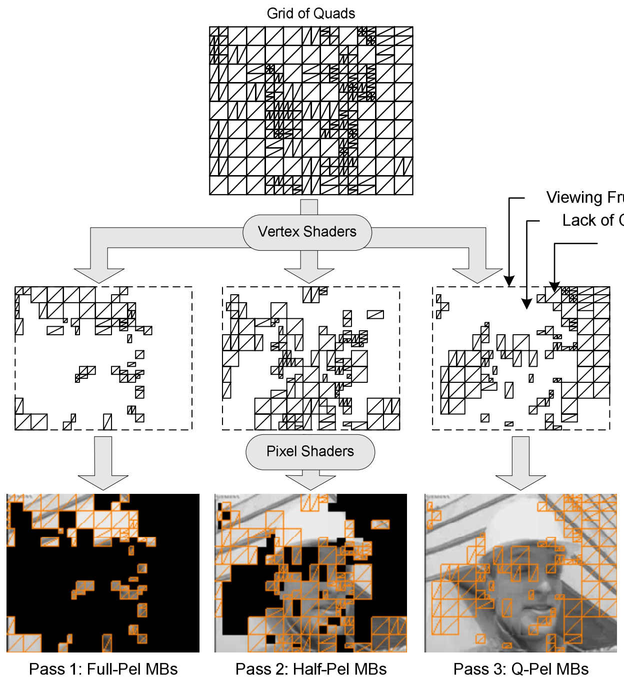

Figure 6: Using vertex shaders to select the required macroblocks for a specific pixel shader. In practice, six render passes are needed.

6.4 Motion Compensation of B Slices

The process of motion compensation of B slices is done similar to that of P slices. For each reference list, motion compensation as discussed in Section 6.2 is repeated using the reference frame from the current reference list. In case of a second reference list, motion-compensated output for that list is written to an additional texture and this texture is merged with the motion compensated picture of the first reference list in a new render pass using a pixel shader.

6.5 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. 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 for 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. However, it only requires little CPU processing time for data preparation. Indeed, today's PCI-Express graphics bus is fast enough to deliver the data in time.

6.6 Intra Prediction

In the presented implementation, intra prediction is still 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.

As mentioned in Section 2.2.1.2, intra prediction can be based on samples of neighboring P macroblocks. This implies that I macroblocks predicted by the CPU can use P macroblock samples located in GPU memory. This behavior could compromise the pipelined fashion CPU and GPU work together shown in Figure 2. The CPU could have to wait for the GPU to start decoding the next video picture. Therefore, Constrained Intra Prediction (CIP) was enabled during encoding of the video sequence, disabling the possibility of using P macroblocks for intra prediction.

7. DECODING H.264/AVC BITSTREAMS WITH NVIDIA CUDA

Our second GPU-driven video accelerator for H.264/AVC decoding relies on the NVIDIA CUDA computing architecture to steer the GPU. Again, the implementation starts at the Intermediary Buffer in Figure 2, containing positions, dimensions, motion vectors, and residual data for all macroblocks, macroblock partitions, and sub-macroblock partitions.

7.1 Introduction to the NVIDIA CUDA Platform

CUDA is a new computing architecture from graphics card manufacturer NVIDIA, targeted at using the GPU to accelerate the execution of computational problems. It enables the user to deliver a large amount of computations to the GPU, which is seen as a dedicated super-threaded co-processor in the CUDA programming model. The architecture allows a programmer to develop applications using standard C, extended with a number of annotations used to set memory locations and coordinate thread execution.

7.2 NVIDIA CUDA Programming Model

In the NVIDIA CUDA computing architecture[10], the GPU, which has its own DRAM, is viewed as a dedicated co-processor to the CPU or *host*. In this model, the GPU runs many jobs in parallel, called *threads*. Though the name suggests a similarity with traditional CPU bound threads, there are a number of differences, the most important one being the lightweight nature of the GPU threads. Because GPU threads come with little overhead (i.e., little creation time), the GPU can manage a significant number of such threads simultaneously. Consequently, context switching generates little overhead. This enables the GPU to execute thousands of threads in parallel while the CPU typically processes only a dozen of threads, not necessarily in parallel. Consequently, to fully utilize the power of the GPU, the user has to provide a sufficient number of jobs to be processed in threads.

To use the GPU in calculations, data-parallel portions of an application are executed on the device as *kernels* which run many threads in parallel. NVIDIA has chosen to structure and group together threads in entities called *blocks*. A kernel is executed as a *grid* of thread blocks. Figure 7a shows this model. All threads in a thread block share data memory space and can cooperate by synchronizing execution with each other. They can also share data through low-latency shared memory. Threads from two different blocks cannot communicate with each other.

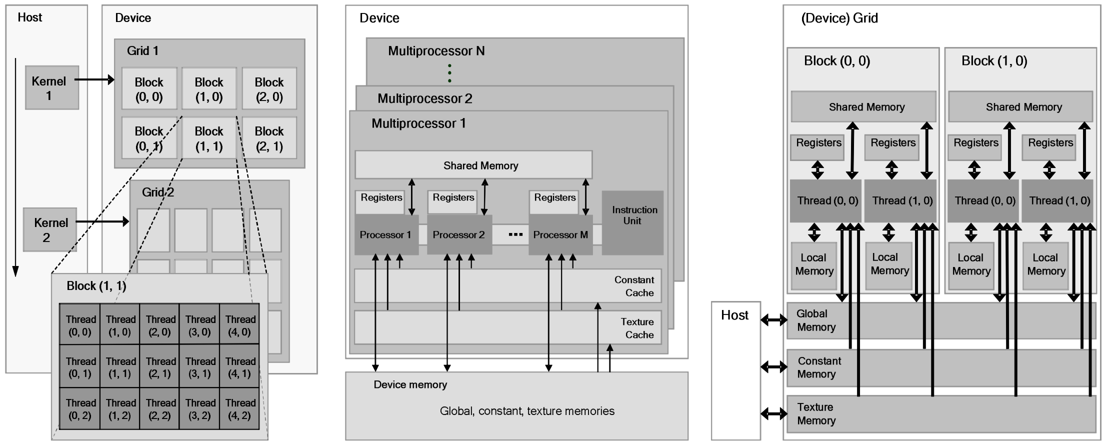

Figure 7: From left to right as stated in the CUDA Programming Guide[10]: (a) CUDA Programming Model; (b) CUDA Hardware Model; (c) CUDA Memory Model.

7.2.1 NVIDIA CUDA Hardware Model

Figure 7b shows the NVIDIA CUDA hardware model. The underlying hardware model differs from the programming model. Here, a number of multiprocessors are available, each having a fixed number of processors. Each multiprocessor is characterized by a Single Instruction, Multiple Data architecture (SIMD). Each processor of the multiprocessor executes the same instruction at a given clock cycle, but operates on different data. This concept is known as lock-step execution; it is illustrated in Figure 8. Consequently, when different processors follow different execution paths, processors not following the same branch are suspended. Different execution paths are consecutively executed until the threads can converge back to the same execution path. This means control flow instructions, and therefore heterogeneous program execution paths, have to be avoided or an optimal computational throughput cannot be reached.

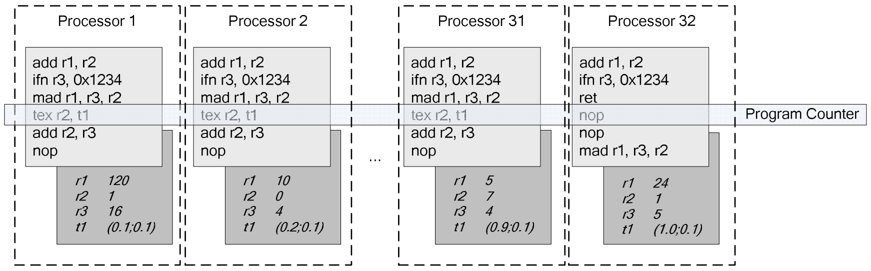

Figure 8: CUDA instruction level lock-step execution with 32 processors.

Each multiprocessor executes one or more blocks using time slicing. On a multiprocessor, each block is split into SIMD groups of threads called warps. At any given time, a single warp is executed over all processors in the multiprocessor. A thread scheduler periodically switches from one warp to another. This happens in a smart way, for example, when all threads from the warp execute a memory fetch and have to wait several clock cycles for the result. For the GeForce 8800 GTX graphics card, the warp size is 32. This implies that, at any given time, 32 threads are executed simultaneously on a single multiprocessor. The GeForce 8800 GTX has 16 multiprocessors. Consequently, at a given time, a minimum of 512 threads are executed in parallel. It is easy to see that if the number of threads in a block to be executed on a single multiprocessor is not a multiple of 32, a number of processors will have nothing to compute. Multiple kernels can be scheduled over one or more multiprocessors.

7.2.2 Memory Model

Figure 7b and Figure 7c show the CUDA memory model from respectively the hardware model and the programming model viewpoint. This memory model complicates programming CUDA because of its many layers but increases flexibility. On a given multiprocessor, each processor or thread has a number of registers available. Shared Memory, a very fast type of DRAM, is shared between processors in a multiprocessor. This memory is typically 16 KB in size. Global Memory is high-latency memory, available from all threads, but it is not cached. Constant Memory is a portion of the Global Memory to store constants accessible from threads. Texture Memory is a high-latency memory with some extra features specifically designed for texture filtering. Texture Memory offers for example bilinear filtering, as well as automatic texture address cropping. Both Constant Memory and Texture memory have a small cache enabling faster memory access for recurrent memory locations. It is up to the user to select the most appropriate memory location.

7.2.3 NVIDIA CUDA Driver Model

Although CUDA and Direct3D differ a lot, both platforms use the same underlying hardware. Direct3D communicates with an underlying Hardware Abstraction Layer (HAL) provided by the display driver, which is specifically optimized for a 3-D graphics context. CUDA, on the other hand, directly communicates to the CUDA dedicated driver, optimized for general purpose computation. This direct approach introduces less overhead than a 3-D graphics API. Figure 9 illustrates both CUDA and Direct3D driver models.

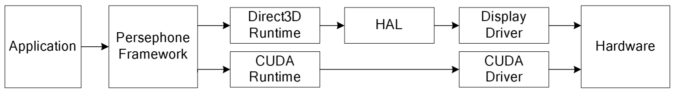

Figure 9: GPU programming abstraction layers.

7.3 Motion Compensation, Reconstruction, and CSC using the CUDA Platform

Using the CUDA programming model, MC, reconstruction, and CSC were implemented. With its many possible thread mappings and memory layers, several implementations were investigated. In this paragraph, a number of design choices will be outlined in more detail.

With motion compensation, multiple interpolation schemes pose a threat to the parallel execution model of the CUDA platform. Two conditions have to be met in order to profit from the fast parallel execution on CUDA hardware. First, all calculations have to be as uniform as possible. As mentioned before, threads are grouped together to form a warp, with instructions of threads in the warp to be executed in lock-step. The CUDA hardware model implies that if all threads of a warp do not follow the same execution path, then diverged control paths are serialized. This significantly increases the number of executed instructions.

The second condition that is to be met in order to efficiently execute computations on CUDA hardware involves the number of scheduled threads. This number has to be high and a multiple of the warp size, i.e. 256. As CUDA tries to execute as many threads as possible, the thread scheduler switches threads out of context when, for example, high-latency memory access occurs. To provide enough threads for efficient context switching, a high number of threads has to be scheduled for execution in parallel. Our implementation of H.264/AVC decoding for instance uses 256 threads.

7.3.1 Thread Mappings

In order to take optimal advantage of the provided hardware, computations have to be mapped to the CUDA thread model, taking into account previous thoughts. Three different thread mapping schemes are possible: *pixel-to-thread*, *macroblock-to-thread,* and *slice-to-thread*. The first mapping, *pixel-to-thread*, implies that every thread processes one pixel of the output picture. Because of the different interpolation schemes previously mentioned, as well because of the warp size of 32 threads, the execution paths of threads executed in the same warp are not guaranteed to be homogenous. Inter prediction of one sub-macroblock partition containing 16 pixels would only occupy half of the warp at a specific moment in time. The other half of threads of the warp would be occupied with interpolation computations for another macroblock, not necessarily using the same interpolation strategy as the first half of threads in the warp. Because of instruction level lock-step execution of threads, the computational throughput would not be optimal.

The second mapping, *macroblock-to-thread*, implies the use of a single thread for processing an entire macroblock, macroblock partition, or sub-macroblock partition. With the high granularity of macroblock interpolation schemes, coarser mappings mean more chance of inefficient occupation of the warp. Therefore, this second mapping can be considered inferior to the first one.

The third mapping, *slice-to-thread*, suggests that MC of every different slice would be executed on a single thread. Because of previous arguments regarding the warp size and lock-step execution, as well because of the limited number of coded slices usually available in a video coded picture, it is obvious that this mapping would not perform well.

7.3.2 Design Choices

The *pixel-to-thread* mapping was found to be the most efficient one in terms of execution speed. Each thread calculates one motion-compensated pixel. To minimize flow control instructions, all macroblocks were first grouped according to the required interpolation strategy for motion compensation. For each interpolation scheme, a kernel was executed. This way, minimal divergence of thread execution paths is accomplished.

The variable block-size motion compensation available in H.264/AVC conflicts with the CUDA programming model, which most efficiently processes uniform data. This forced us to address the motion compensation as uniform as possible. Indeed, when decoding an H.264/AVC video bitstream on the CPU, all macroblocks and macroblock partitions are represented using sub-macroblock partitions. The motion vectors and reference frame numbers are duplicated over all constructed sub-macroblock partitions.

Texture Memory was chosen to contain the video pictures used as reference pictures in the MC process. This allows for a straightforward implementation, without the need to explicitly cache pixels using shared memory and thread synchronization. All reference pictures are grouped in one texture as the current CUDA version does not allow texture indexing or three-dimensional textures yet.

7.3.3 CSC and Visualization

The motion-compensated and reconstructed picture has to be transformed from the YCbCr color space to the RGB color space. The uniformity of the calculations makes this an ideal task for the CUDA platform. Each thread can transform one sample. The next step, visualization, is done by a graphics API, here Direct3D. Although the (upsampled) RGB-transformed picture is resident in GPU-memory, the current beta status of the CUDA platform prohibits us from using CUDA-allocated memory directly in Direct3D other than as vertex buffers. To circumvent this limitation, the picture has to be downloaded to system memory, only to be uploaded again to GPU memory for use by Direct3D. It is obvious that this step needlessly stresses the graphics bus, consequently diminishing the performance. To minimize bus transfers as much as possible, CSC is done in D3D by use of pixel shaders. This is because of downsampling of the chroma pictures.

7.4 CUDA Limitations and Conclusions

The limitation discussed in the previous paragraph, in particular the inability to use CUDA memory as texture memory in Direct3D, is due to the current beta state of the CUDA platform. A second temporary limitation is the inability to execute CUDA API commands asynchronously. This means in the case of motion compensation, when a kernel is executed, memory not related to the kernel's execution cannot be downloaded or uploaded. This is in contrast to Direct3D, where, for example, residual data are uploaded at the same time that the motion compensation is executed on the GPU. This implies that the CPU is waiting continuously until each CUDA API call ends. This can potentially stall the pipelined model described in Section 4. NVIDIA has stated to remove these two restrictions in future versions.

Although the CUDA platform provides more flexibility over Direct3D, for MC specifically, the source data for the motion compensation process needs careful handling: where Direct3D can easily work with variable macroblock sizes represented by quads of different sizes, CUDA best works with uniform data structures.

8. COMPARISON

We compared both Direct3D and CUDA-based H.264/AVC decoders to a software implementation based on the motion compensation module in the libavcodec-library[11]. To focus performance-wise on the accelerators, three H.264/AVC sequences were decoded; intermediate outputs containing motion vectors and residual data were written to disk. This way, entropy decoding, IQ, and inverse transformation speed limits were omitted. The test sequences are *driving*, *shields,* and *blue skies*, with resolutions of respectively *720x480*, *1280x720*, and *1920x1088*. All progressive sequences use B slices and are coded in an *IBPB* slice-coded picture structure, using up to sixteen reference frames and one slice per picture. For all tests we used Direct3D 9.0c (February 2007 SDK), CUDA Toolkit version 0.8.1, and Windows XP SP2.

Table 1: Performance Results of CPU- and GPU-based Renderers.

System and Graphics Card	Resolution	CPU Renderer		Direct3D GPU Renderer		CUDA GPU Renderer	
		fps	cycles (%)	fps	cycles (%)	fps	cycles (%)
Intel Pentium D 930 GeForce 8800 GTX	720x480	90.1	98.4	215.6	52.3	144.5	98.3
	1280x720	36.9	98.0	78.9	63.1	60.9	97.1
	1920x1088	16.5	98.6	41.2	58.8	33.8	99.6

Table 1 shows the number of frames presented on screen (frames per second; fps), as well as the CPU processing power required by the renderers (processing cycles). The table clearly shows that the GPU-driven renderers outperform the CPU renderer with a factor of up to 2.4. While the CPU renderer fails to deliver 1080p images in real time, the GPU-driven renderers have no problem providing the necessary 30 frames per second. When comparing both GPU renderers with each other, we see the Direct3D renderer finishing with a head start.

The fact that the CUDA platform is still in a development phase can explain why the Direct3D renderer delivers more frames per second. As discussed in Section 7.4, temporary implementation limitations prohibit the pipeline of the CUDA renderer to work efficiently. Bus traffic due to the inability to use CUDA memory directly as Direct3D textures decreases performance, as well as the fact that CUDA API calls are handled synchronous. This last limitation is the cause for the high CPU utilization of the CUDA renderer in comparison to the Direct3D renderer. Despite these results, CUDA shows significant promises when these restrictions can be removed in future versions.

Our calculations show that decoding a B slice roughly takes two times the amount of processing time of that of decoding a P slice (due to additional interpolations and motion vectors and reference indices that need to be uploaded).

Note that the H.264/AVC deblocking filter was disabled. First, this filter is intuitively of little use when decoding high-definition video sequences with high bitrates. Second, the nature of the H.264/AVC deblocking algorithm is such that it is difficult to execute the filter in parallel[12]. Indeed, a preliminary GPU-based implementation of the deblocking filter for instance reduced the rendering speed to 2 frames per second for all renderers. For more information regarding limitations of H.264/AVC decoding on the GPU, we refer to Pieters *et al.*[13]

9. CONCLUSION AND FUTURE WORK

Several decoding tools of the H.264/AVC standard were implemented using GPU-accelerated software components, by means of Direct3D, as well as by means of the new NVIDIA CUDA platform. We achieved motion compensation, reconstruction, and visualization in real time for high-definition video (1080p) on a PC with an Intel Pentium D 930 and an NVIDIA GeForce 8800 GTX using our Direct3D and CUDA renderer. The CUDA renderer shows to be less performing, but it can be assumed that beta development status with its temporary implementation restrictions can be held responsible for this behavior. The Direct3D renderer uses significantly less CPU cycles than a CPU-only based solution (58.8% vs. 98.6%), while achieving higher frame rates (41.2 fps vs. 16.5 fps). The CUDA renderer performs better than a CPU-only based solution (33.8 fps vs. 16.5 fps). Fundamental issues and limitations of the architecture of the GPU were identified and discussed, such as the need for uniform calculations.

Future work will focus on further delegation of decoding steps to the GPU. The first candidate is intra prediction, which proves to be a challenge as its design conflicts with the GPU's programming model. Further work will also focus on investigating the feasibility of GPU-assisted decoding of scalable video content, as well as on the development of a video codec specifically designed for the GPU, aimed at lossless encoding and decoding of high-definition video sequences with high dynamic color range capabilities.

10. ACKNOWLEDGEMENTS

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. J. Sullivan and T. Wiegand, "Video Compression – From Concepts to the H.264/AVC Standard", *Proceedings of the IEEE*, vol. 93, no. 1, pp. 18–31, IEEE, 2005.
2. V. Lappalainen, A. Hallapuro, and T. D. Hämäläinen, "Complexity of Optimized H.26L Video Decoder Implementation", IEEE Circuits and Systems for Video Technology, **vol. 13**, no. 7, pp. 717–725, 2003.
3. M. D. Pesce, *Programming Microsoft DirectShow for Digital Video and Television*, Microsoft Press, 2003.
4. G. Sullivan and C. Fogg, "Microsoft DirectX VA: Video Acceleration API/DDI Version 1.01", January 2001.
5. Microsoft Corp., "DirectX Video Acceleration 2.0", http://msdn2.microsoft.com/en-us/library/aa965263.aspx.
6. G. Shen, G. Gao, S. Li, H. Shum, and Y. Zhang, "Accelerate Video Decoding With Generic GPU", IEEE Circuits and Systems for Video Technology, **vol. 15**, no. 5, pp. 685–693, 2005.
7. C.-W. Ho, O. Au, S.-H. Chan, S.-K. Yip, and H.-M. Wong, "Motion Estimation for H.264/AVC using Programmable Graphics Hardware", *Proceedings of the IEEE International Conference on Multimedia Expo (ICME)*, pp. 2049-2052, IEEE, 2006.
8. K. Gray, *The Microsoft DirectX 9 Programmable Graphics Pipeline,* Microsoft Press, 2003.
9. D. Van Rijsselbergen, W. De Neve, and R. Van de Walle, "GPU-driven Recombination and Transformation of YCoCg-R Video Samples", *Proceedings of the Fourth IASTED International Conference*, pp. 21–26, no. 531, ACTA Press, 2006.
10. NVIDIA Corp., "NVIDIA CUDA Compute Unified Device Architecture: Programming Guide", version 0.8.1, April 2007.
11. "The FFmpeg project", including libavcodec library, http://ffmpeg.mplayerhq.hu/.
12. S. Yang, S. Wang, and J. Wu, "A Parallel Algorithm for H.264/AVC Deblocking Filter Based on Limited Error Propagation Effect", *Proceedings of the IEEE International Conference on Multimedia Expo (ICME)*, IEEE, 2007.
13. B. Pieters, D. Van Rijsselbergen, W. De Neve, and R. Van de Walle, "Motion Compensation and Reconstruction of H.264/AVC Video Bitstreams using the GPU", *Proceedings of the Eighth International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS '07)*, p. 69, IEEE, 2007.
14. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat, "Brook for GPUs: Stream Computing on Graphics Hardware", ACM Transactions on Graphics, **vol. 23**, no. 3, pp. 777-786, 2004.