ACCELERATING VIDEO DECODING USING GPU

Guobin Shen[a], Lihua Zhu[b], Shipeng Li[a], Heung-Yeung Shum[a] and Ya-Qin Zhang[a]

[a] Microsoft Research, Asia
[b] Institution of Automation, Chinese Academy Science

ABSTRACT

Most modern computers or game consoles are equipped with powerful graphics processing units (GPUs) to accelerate graphics operations. There is a trend that the power of GPU outgrows that of CPU (central processing unit). However, the GPU engines are specially designed for graphics operations. Can we take advantage of the powerful GPU engines for more general operations other than pure graphics operations? The answer is positive. In this study, we present schemes that map other non-graphics operations into graphics engines with an example application of accelerating video decoding with the assistance of GPU. Our results show that significant speed-up can be achieved by leveraging the GPU power. Specifically, we have achieved real-time playback of high definition video on a PC with an Intel Pentium III 667 MHz CPU and an nVidia GeForce3 GPU.

1. INTRODUCTION

With the advance of silicon and computer graphics technologies, more and more inexpensive yet powerful graphics processing units (GPUs) can be found in mainstream personal computers and game consoles. GPUs are equipped with specialized processors designed just for 2D and 3D graphics operations and they indeed do an excellent job [1]. On the other hand, multimedia is the core of digital entertainment and it usually requires very high processing power especially for real-time applications. When real-time multimedia applications are implemented with a general purpose computer, CPU is usually heavily loaded and in many cases that CPU alone can not meet the real time requirement at all. For example, currently CPUs in most household PCs alone are not powerful enough to decode high definition (HD) video in real-time.

However, for non-graphics oriented applications, the GPU is usually idle while CPU is heavily loaded. A question comes along naturally: can we leverage the power of GPU to off-load the CPU for some tasks, especially when the GPU is idle? In this study, we will present schemes to map non-graphics operations onto graphics engines with an example application of accelerating digital video decoding with the assistance of GPU. We should note that, some today's graphic cards have a special hardware unit that can accelerate the video decoding process, thanks to the well established international video coding standards such as MPEG-1/2/4 [2] and the widely endorsed DirectX video accelerator (DXVA) specification [3]. However, such a hardware video decoding accelerator is only limited to certain standard video coding formats. They can not handle video that may be coded with other proprietary yet very popular video coding formats, such as Windows Media Video (WMV) [4] and RealVideo [5]. Moreover, even though almost all GPUs can help on video rendering (through overlay), they can only provide very limited flexibility in manipulating the video decoded.

In order to support more flexible and wider application scenarios, we can not rely on these non-standard specialized hardware accelerators. In this study, we investigate how the common DirectX-8 compatible graphics engines can be exploited to assist the CPU in video decoding. We choose DirectX-8 is because of its predominance and rich APIs as an industrial standard. Our study confirms that the GPU power can indeed be leveraged for non-graphics applications. Furthermore, since the video is handled by the graphics engine directly, it provides a more efficient way to incorporate video into computer graphics, which is of wide interest in today's gaming industry.

The rest of the paper is organized as follows. In Section 2, we briefly review the architecture of a modern GPU. Section 3 highlights the general procedure of video decoding and analyzes the complexity of the building blocks. We then present a solution of exploiting GPU to accelerate video decoding in Section 4. Some experimental results are given in Section 5 and Section 6 concludes the paper.

2. GRAPHICS ENGINE ARCHITECTURE

Recent years have witnessed dramatic increases in the GPU processing power at a speed even faster than Moore's law to CPU due to breakthroughs in computer graphics field, innovations in silicon design and advances in semiconductor technologies. One most significant step forward is the introduction of user-programmable geometry engine [6] and the pixel pipeline. The principal 3D APIs (DirectX and OpenGL) have evolved alongside graphics hardware. One of the most important new features in DirectX graphics is the addition of a programmable pipeline that provides an assembly language

interface to the transformation and lighting hardware (vertex shader) and the pixel pipeline (pixel shader).

Vertex shaders are small programs describing a procedure to be applied to polygon vertices in the scene. Pixel shaders are small programs describing operations to be applied to pixels in the frame buffer. Pixel shaders' function is similar to vertex shaders', except that they perform operations manipulating colors and textures, rather than geometry. Note that the vertex shader calculates effects on a per-vertex basis (i.e., polygon based rendering) while the pixel shader operates on a per-pixel basis. For legacy performance, most GPUs still keep the old fixed-function pipeline (the standard Transform & Lighting pipeline where the functionality is essentially fixed). A greatly simplified graphics pipeline is shown in Figure 1 [7].

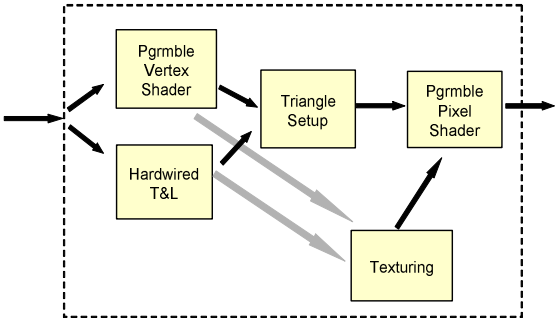


Figure 1. A greatly simplified graphics pipeline **[7]**

Modern mainstream GPUs are indeed very powerful, thanks to the complete fine-grained SIMD parallelism and pipelining. In Table 1, we list some performance metrics of nVidia GeForce3 [8].

Graphics Core	256-bit
Memory Interface	128-bit DDR
Fill Rate	3.2 Billion AA Samples/Sec.
Operations per Second	800 Billion
Memory Bandwidth	7.36GB/Sec.

Table 1. Performance of nVidia GeForce3 **[8]**

In summary, the programmable pipeline of GPU gives developers a lot more freedom to achieve special effects. The power of GPU ensures these special effects can be made into real time graphics applications.

3. TYPICAL VIDEO DECODER ARCHITECTURE

A typical video decoder consists of several building blocks, namely variable length decoding (VLD), de-quantization (IQ), inverse DCT (IDCT), motion compensation (MC), and color space conversion (CSC), as shown in Figure 2. Motion compensation is an efficient technology that exploits the temporal correlation between neighboring frames of a video sequence. Color space conversion module converts YUV to RGB for the display purpose.

Note that there is a feedback loop in a video decoder. Because of this feedback loop, the motion compensated signal has to exactly match that in the encoder. Any error introduced, albeit small, will be accumulated and propagated to future frames. This is called drifting. Drifting usually leads to quick video quality degradation and therefore must be prevented.

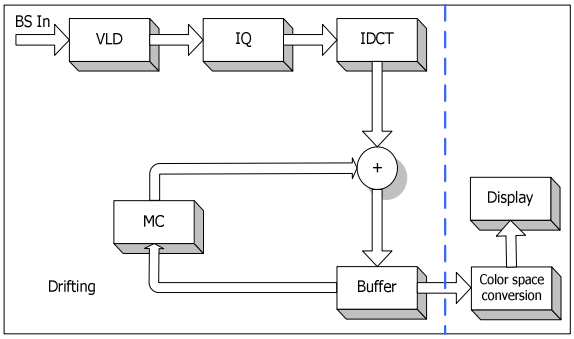
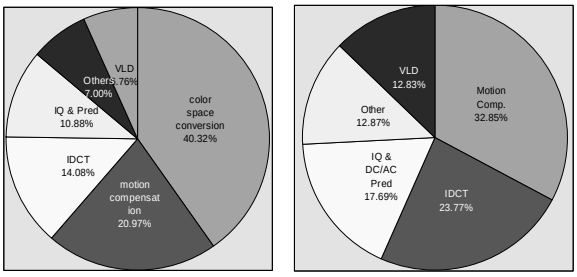


Figure 2. Block diagram of a typical decoder

The most computationally intensive parts, in a decreasing order, are CSC, MC, IDCT, IQ and VLD. Figure 3-(a) shows the load profiling result for a typical video decoder (for a proprietary MPEG-like video format) on Pentium III 667 MHz CPU when decoding an HD (1280×720) video sequence. Evidently, the CSC and MC consume most of the overall computation power (more than 60%). Since the CSC module can usually be handled by the GPU, we also performed the profiling that excludes the CSC module, as shown in Figure 3-(b). Clearly, the MC still occupies a significant portion of the whole computation load.



(a) (b)

Figure 3. Load profiling of building blocks in video decoding. (a) all modules, and (b) all modules except color space conversion.

According to the profiling results, it is very desirable that the color space conversion and motion compensation can be handled more efficiently. Ideally, they should be handled by some other processing units or hardware accelerator. As discussed in next section, this can be achieved by moving these two modules into the GPU.

4. GPU-ASSISTED VIDEO DECODING

4.1. Feasibility

Since the GPU is specially designed for faster graphics operations and better graphics effects instead of for assisting decoding video, there is no direct mapping of video decoding algorithms to the 2-D or 3-D graphics engines. On the other hand, the per-vertex and per-pixel operations of GPU may still be utilized to handle partial of the video decoding task. That is, we can use the GPU to take over some video decoding stages that involve only per-vertex and per-pixel operations in nature.

We analyze the nature of each building block of a typical video decoder as shown in Table 2. In this table, block-wise means that the operations are performed on a block by block basis. A block is a simple polygon and therefore their vertices can be handled by the vertex shader. Clearly, the most computationally complex MC and CSC modules are intrinsically suitable for the GPU to process. Since IQ and IDCT are not per-pixel operations and VLD is a purely sequential operation, they have to be handled by the host CPU.

Module	Block-wise	Per-pixel
VLD	x	x
IQ	√	x
IDCT	√	x
MC	√	√
CSC	√	√

Table 2. Nature of each module of a video decoder

Read-back from GPU memory to that of CPU is very expensive, due the asymmetric design of AGP bus. As a result, such read-back must be avoided in a practical design. This is achievable by moving the whole feedback loop to GPU. In other words, the CPU will not access the data after submitted to the GPU.

4.2. Constraints

Even though GPU is very powerful, it still has many constraints. These constraints are more visible when exploiting GPU for non-graphics oriented applications. Some main constraints are as follows:

- The memory bandwidth between CPU and GPU is limited.
- The internal precision of pixel shader is limited. For example, the nVidia GeForce3 GPU's internal precision is only up to 8 bits.
- The instruction set is small. Most instructions are specially designed for graphics operations.
- The code line count for pixel shader programming is limited. Only up to 8 arithmetic instructions are allowed in any rendering pass.

4.3. Proposed solution

Based on the analysis above, our solution is to move the whole feedback loop that consists of motion compensation (including a padding process which is used to handle motion vectors that point outside the reference picture) and color space conversion to the GPU.

For motion compensation, we use the vertex shader to handle the motion vectors. That is, vertex shader is used to compute the target block positions and the source (reference) texture addresses for the macroblocks to be motion-compensated. Motion compensation can be performed at different precisions for INTER blocks such as integer pixel level and sub-pixel level (e.g., half-pel and quarter-pel). For sub-pixel level MC, some interpolation processes are generally involved. There is a special type of macroblocks called INTRA block. No motion compensation is needed for these blocks. We adopted a divide-and-conquer approach to handle different MC types and precisions, as shown in Figure 4. Note that there are two ways to handle INTRA blocks. One way is to handle them via a separate rendering pass, the other way is to treat them as integer-pel INTER blocks with pseudo motion vectors that point to some zero area. We adopted the second method.

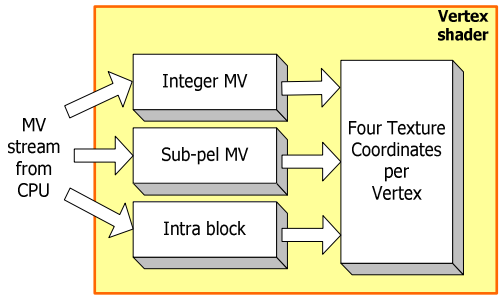


Figure 4. Divide-and-conquer approach for MV handling.

Now let's briefly go through the working flow of MC and CSC inside the GPU. The motion vector is first transferred from CPU to GPU and processed by vertex shader. The vertex shader generates the target block positions for triangle setup and the texture addresses for sampling the textures. The pixel shader will then use the texture addresses generated by the vertex shader to sample the texture, perform necessary arithmetic operations to obtain the motion compensated result, and render the result to the target positions. After the motion compensation is done, the CPU will transfer the difference data to GPU. The GPU will then reconstruct a new picture by adding the difference to the motion compensated reference. The picture will go through color space conversion and be sent for display. The picture will also be used as the reference of the next frame.

Because the geometric operations of MC and CSC are extremely simple, the main tasks are at the pixel shaders. There are totally five main steps that involve pixel shaders. Among them, four are mandatory and one is optional, see Figure 5. In this figure, a round-cornered rectangle represents a pixel shader. The first pixel shader restores the difference picture by unpacking the packed difference data. This is an optional step

though to reduce the bandwidth of transferring difference picture from CPU memory to GPU memory. The second pixel shader prepares a padded reference for the subsequent motion compensation process that is to be handled by the third pixel shader. The fourth pixel shader then adds the difference data to the motion compensated reference to form the final reconstructed picture. The fifth pixel shader converts the YUV format of the reconstructed picture to RGB format. Note that this pixel shader directly renders to the back buffer, and there are no extra steps needed to display the final decoded picture.

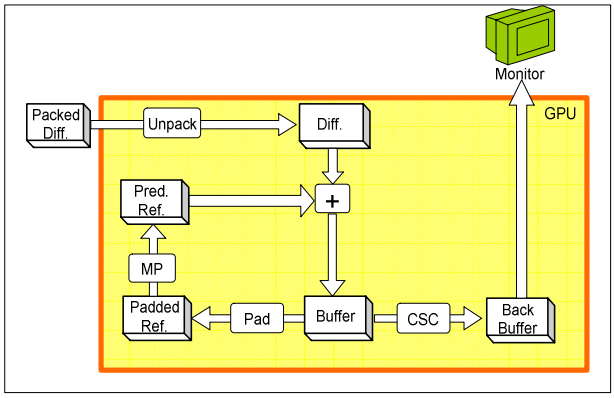


Figure 5. Flow chart of MC and CSC in GPU

All the steps mentioned above are relatively straightforward since MC is basically a translation operation of textures. The difficulty lies in the drifting prevention considering the limited internal precision of a GPU. It is just such a drifting effect that essentially distinguishes the video decoding from normal graphics operations. Extra care is needed to avoid the potential drifting problem especially for sub-pixel motion compensation.

According to DirectX-8 specifications, the only render target format that suits for the video decoding application is the 32-bit D3DFMT_A8R8G8B8 format. However, the dynamic range of the difference data is usually from -255 to 255, which only needs 9 bits to represent. Therefore, it would be very beneficial if we could pack three 9-bit difference values with one 32-bit pixel because this effectively reduces the memory bandwidth requirement by two thirds. To further reduce memory bandwidth, the CPU transfers only non-zero difference blocks to GPU.

5. EXPERIMENTAL RESULTS

We have proposed and implemented a solution to decode the proprietary MPEG-like video. We also performed extensive tests on a PC with an Intel Pentium III 667 MHz CPU and an nVidia GeForce3Ti200 GPU. The test sequences used are Football, Total, and Trap. The Football sequence is a standard MPEG test sequence in SIF format (320x240) with very high motion. The Total sequence is a concatenation of several standard MPEG test sequences (such as Carphone, Stephan, Silence, Akiyo, etc.) in CIF format (352x288). The Trap sequence is a high definition version (1280x720) of the movie trailer of “The Parent Trap” (Disney, 1998). We encoded the SIF and CIF sequences at 2 Mbps and the HD sequence at 5 Mbps, respectively.

We compare the video decoding speed achieved using CPU only against that achieved using CPU and GPU, as shown in Table 3. Clearly, the speed is significantly improved by leveraging the power of GPU. It is interesting to observe that the speed-up of the Total sequence is much higher than Football, while the speed-up of the Trap is by large the most significant one. The reason mainly lies in the bandwidth requirement. For the same bit rate, due to its high motion nature, Football produces a lot more non-zero difference blocks than the Total does. On the other hand, the Trap sequence at 5 Mbps leads to a high percentage of zero difference blocks. As a result, the memory bandwidth requirement is greatly reduced since we do not transfer any zero block.

Sequence \ Decoder			CPU	CPU +GPU
Football	SIF (320x240)	2 Mbps	81.0 fps	135.4 fps
Total	CIF (352x288)	2 Mbps	84.7 fps	186.7 fps
Trap	HD (1280x720)	5 Mbps	9.9 fps	31.3 fps

Table 3. Experimental results of GPU assisted video decoding on PC with an Intel Pentium III 667 MHz CPU and an nVidia GeForce3Ti200 GPU.

6. CONCLUSION

In this paper, we proved that GPU can indeed do more than just graphics. We have demonstrated that GPU can help CPU to accelerate video decoding. We achieved real-time decoding of high definition video on PC with a slow CPU and a powerful GPU, which is definitely impossible otherwise. It is definitely worthwhile to further investigate how a GPU can accelerate other non-graphics oriented operations.

7. REFERENCES

[1] http://www.siggraph.org/s2002/conference/papers/papers8.html
[2] http://www.mpeg.org/MPEG/video.html
[3] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/graphics/hh/graphics/dxvaguide_48x3.asp
[4] Windows Media Video, http://www.windowsmedia.com
[5] RealVideo, http://www.realnetworks.com
[6] Erik Lindholm, Mark J. Kilgard, Henry Moreton, “A user programmable vertex engine”, ACM SIGGRAPH 2001.
[7] http://www.nvidia.com/docs/lo/67/SUPP/vertexshaders.pdf
[8] http://www.nvidia.com/view.asp?PAGE=geforce3