Using Graphics Processor Units (GPUs) for Automatic Video Structuring

Peter Kehoe and Alan F. Smeaton
Centre for Digital Video Processing and Adaptive Information Cluster
Dublin City University, Glasnevin, Dublin 9, Ireland.
Alan.Smeaton@dcu.ie

Abstract

The rapid pace of development of Graphic Processor Units (GPUs) in recent years in terms of performance and programmability has attracted the attention of those seeking to leverage alternative architectures for better performance than that which commodity CPUs can provide. In this paper, the potential of the GPU in automatically structuring video is examined, specifically in shot boundary detection and representative keyframe selection techniques. We first introduce the programming model of the GPU and outline the implementation of techniques for shot boundary detection and representative keyframe selection on both the CPU and GPU, using histogram comparisons. We compare the approaches and present performance results for both the CPU and GPU. Overall these results demonstrate the significant potential for the GPU in this domain.

1. Introduction

Recently, researchers have increasingly become interested in the potential of Graphics Processor Units (GPUs) for a variety of computational tasks beyond graphics rendering. The motivation is the promise of high speedups compared to commodity desktop CPUs, thanks to the very high parallelism employed by GPUs and the massive advantage this gives the GPU in floating point computational capability. Using the GPU effectively also presents unique challenges, as they have a distinct programming model.

This work examines the potential of the GPU compared to the CPU for structuring video – specifically in shot boundary detection and representative keyframe selection. Shot boundary detection is the process of segmenting a video into its component camera shots, which may be delineated by an opening and closing cut. This is commonly used as the first step in automated video content analysis. Aside from the shot boundary detection itself, a subsequent technique called representative keyframe selection may be used to identify a single frame that will represent a given shot. Here we examine implementations of both techniques on the CPU and on the GPU, and compare their performances.

2. Related Work

Our shot boundary detection and keyframe selection techniques work with decompressed video frames. Thus the process of taking a compressed video file and decompressing it is a first step in this process which takes a significant proportion of the overall time required for the entire process, and represents a good candidate for acceleration on the GPU. This has already been undertaken by a number of graphics processor vendors, such as in nVidia's PureVideo technology, and ATi's Avivo. Both companies' technologies offload a number of the most computationally intensive aspects of MPEG decoding to the GPU, in order to speed up the process over the CPU alone so we will concentrate our wok on the post-decoding phases.

There has been a significant amount of research into shot boundary techniques and the proceedings of the TRECVid conferences [3] over the last five years or so present a good body of knowledge in the field. Many techniques exist for shot boundary detection, including pixel and histogram comparisons and other statistical differences between frames. Some approaches focus on different types of shot boundary, from hard cuts to gradual transitions e.g. fades and dissolves.

3. Graphics Processor Units

GPUs, or Graphics Processor Units, emerged in the PC space in response to the growing demands placed on rendering capability, driven primarily by the videogames market and the breakneck pace in which advances are expected therein. In recent years there has been a growing interest in using GPUs for tasks beyond rendering mainly because of a steady increase in GPU capability and programmability. Initially, GPUs were fixed-function pieces of silicon that simply took input via an API, and produced a picture as output, with little scope for programmer control in the process.

Eight International Workshop on Image Analysis for Multimedia Interactive Services(WIAMIS'07)
0-7695-2818-X/07 $20.00 © 2007 IEEE

2001 saw the introduction of the first programmable GPU which was quite limited in its capability, and required programs to be written in an assembly-like language. However, since then we have the emergence of high level languages, including nVidia's Cg ('C for Graphics') and Microsoft's HLSL (High Level Shading Language). These languages are quite similar to C in syntax, with support for branching, loops, and a wide variety of data types.

A modern GPU today boasts a much higher capability in floating point calculations than commodity CPUs. To look at an example from a couple of years ago, the nVidia Geforce FX 5900 Ultra GPU of the time could manage 20 billion floating point multiplies per second compared to a 3Ghz Pentium 4 which peaks at 6 billion floating point multiples [2]. The GPU also can claim much higher bandwidth to its local memory with as much as 512MB of RAM locally, with up to 50GB/s of bandwidth to that memory. By comparison, todays high-end CPU has up to 8.5GB/s of main memory bandwidth. These numbers make a compelling case for using the GPU, however, the GPU is not suitable for every task and embodies a different programming model than that on the CPU, with the key limitation being limited output. On the CPU, a programmer can write to any location in memory at any time, known as a *scatter* capability. On the GPU the number of outputs is limited to at most one RGBA colour value (i.e. a pixel), which is fixed and pre-determined. Input is read into a GPU from 2D arrays of data called textures. In graphics rendering applications, these are used to apply 2D images to 3D surfaces, to give the appearance of texture, but data can be stored in textures for the purposes of general computation. The readback of results from the GPU to the CPU is slow due to the relatively narrow bus between the CPU and GPU, which is one of the biggest limitations on effective GPU performance. Similarly, passing input data to the GPU during computation is slow and ideally only a low amount of traffic on the bus between the CPU and GPU would be required during computation.

In the work reported here we used a machine with an AMD 3800+ CPU, 512 MB RAM, an nVidia 7800 GT (450Mhz, 24 Pixel Shaders) GPU which had 256MB RAM and a bandwidth of 32GB/s. The API we used was OpenGL, and the shader language chosen was nVidias Cg. All other programming was in C++. For performance tests, timings were averaged over 10 runs.

4. Shot Boundary Detection: Techniques and Implementations

In our implementations of shot bound detection we focussed on a histogram-based approach, popular due to its performance and accuracy [4]. Our focus was on hard cut detection rather than gradual shot transitions such as fades or dissolves. Our technique firstly calculates a colour histogram for each decoded frame of video, secondly computes the difference between adjacent frames based on vector distance and finally identifies candidate shot bounds by comparing adjacent frame similarities against a threshold. This straightforward technique has been evaluated by us in TRECVid in 2001 [1] and performs about average compared to others. Since 2001 there have been many refinements on this technique suggested and evaluated but with relatively minor improvements in precision and recall.

The implementation of a CPU approach to shot boundary detection is reasonably straightforward. We use a simple histogram class with functions for generating a histogram based on a provided array of frame data, and for calculating the distance between the frame and a second frame passed to it. This simplicity of implementation is due to the CPU's competency with both gather and scatter operations. In contrast, a GPU shader[1] lacks any scatter capability meaning the location the shader writes to in memory is preset and cannot be changed within the shader.

After a number of different approaches, we achieved an efficient GPU implementation using an approach which leverages the capability to query the GPU based on a shader that is executing. This querying is exposed by the API, and can be used in rendering graphics to determine, for example, if an object is occluded or not (the application would execute a shader to draw a proxy object in place of the actual more complex version, and use a query to determine if the object was drawn or discarded). This capability can be applied to histogram computation by addressing each bin of the histogram in turn. For each bin, the shader takes the frame as input, and draws it unchanged to another buffer, but first it checks if the pixel to be drawn is within the range of the current bin, whose minimum and maximum values are passed as parameters. If the pixel is within the given range it is drawn, if not it is discarded. The query over this shader simply counts the number of pixels drawn, effectively computing the value for the current bin. Note that in this approach we are still passing over every frame n times for n bins. However we can pass the bin's minimum and maximum values to the shader directly as parameters with each pass, obviating the need for computation of the minimum and maximum values within the shader, without sacrificing parallel speedups.

We calculate the difference between adjacent frames' histograms in one shader pass by packing all the histograms into two textures, one containing all textures from 0 to $m-1$, where m is the number of histograms, and the other contains all textures from 1 to m. So in short, the second texture is the same as the first, except shifted one histogram to the left. This allows the shader to access a frames

[1] The word shader has a dual meaning here, referring to processors in the GPU hardware itself, and to software programs that run on them.

histogram and its neighbours. Our process for computing frame distances on the GPU uses multiple shaders in a sequence, namely:

- Shader 1 takes the two input textures and subtracts them resulting in the vector between each histogram which we subsequently calculate the length of to get the distance between the histograms. Since it is easy to do so at this point, we also square the result, which takes care of part of the Euclidian norm calculation.
- Shader 2: Takes the output texture from Shader 1 and uses row reduction to sum the values in each row of the texture (each histogram) and reduce the texture to one column.
- Shader 3: Takes the output buffer from Shader 2 and simply calculates the square root of each element.

After these 3 passes, we are left with a column of values containing the vector distances between adjacent frames.

# Frames	500 (20s)	1000 (40s)	2000 (1m 20s)
CPU	9.679s	18.743s	35.443s
GPU	2.889s	5.303s	10.283s

Table 1. Histogram-based Shot Boundary Detection (32 bins)

The two implementations (CPU and GPU) were tested on video with runs averaged $\times$ 10, to measure not the accuracy of shot boundary detection but its speed of execution. The results for shot boundary detection, excluding video decoding, are presented in Table 4 and show a clear performance advantage for the GPU, roughly 3 times faster than the CPU in each case. Performance scales roughly linearly on both the CPU and GPU with increasing numbers of frames.

5. Representative Keyframe Selection: Techniques and Implementations

A process often used subsequent to shot boundary detection is keyframe selection. With a set of video shots, each delineated by its opening and closing cuts, it is often desirable to select a keyframe to represent each shot. The cheapest and most common way to do this is to simply select the frame in the middle of the shot, however, this frame may not actually be representative. A more desirable approach is to select the frame that is most like every other frame in the shot and one way to implement this is to calculate the difference between each frame and every other frame in the shot, using histograms, and to average the difference for each frame. The frame with the lowest average distance between itself and every other frame can then be selected as the keyframe. While this has intuitive appeal, the disadvantage of this approach is that it incurs much greater computational cost, of the order $n \times n$, where n is the number of frames in the shot, but it yields a keyframe which is truly representative of the shot. We implemented this computation on the CPU and the GPU and refer to it as *representative* keyframe selection.

Assuming a prior step of shot boundary detection, histograms for every frame in a given shot should already be computed and available for use in representative keyframe selection. The selection process itself is simple and can be coded in a very straightforward manner on the CPU with a double nested for loop.

The GPU approach is again significantly different from that taken on the CPU, using some techniques that are similar to those used in the shot boundary detection GPU implementation. Again, how input is passed to the GPU is a key factor in performance, so we look at this first. As with the first step of shot boundary detection, we calculate the vector distances between histograms, but in this case we need to find the distance between a given histogram and every other histogram, and repeat this process for every frame. A logical extension of the technique used in the shot boundary detection approach might see input of the form of two textures, one containing every frame's histogram (with each histogram on one 'row' of the texture, as before), and one containing one frame's histogram copied across a texture of the same size as the first. This seems sound initially, as it requires only one texture to be prepared for each frame being considered, and the texture containing every frame's histogram need only be prepared and transferred to the GPU once and used repeatedly for subsequent frames. However the cost of preparing and transferring even one texture for each frame is extremely high relative to the amount of computation to be performed, and scales linearly with the number of frames in the scene because of the slow bandwidth in transferring to the GPU. Having tested this approach, it is significantly slower than even the CPU implementation, highlighting the need to pay attention to data transfer to and from the GPU in order to maximise its performance.

To address the shortcomings of the naive approach we developed an alternative. The texture we were preparing and transferring to the GPU for each frame is simply one histogram repeating n times, where n is the number of frames in the shot. Intuitively this seems wasteful — do we really need to transfer all of this data to the GPU when only 1 row in the texture is actually unique ? The answer, luckily, is no. Packing the histogram into the texture n times has the benefit of allowing a $1:1$ mapping between the texture coordinates accessed in each of the two input textures, making it easy to access the input data without altering the interpolated texture coordinates automatically passed into the shader. With some relatively straightforward manipula-

Table 2. Histogram-based Representative Keyframe Selection (32 bins)

Shot length in frames (& duration)	500 (20s)	1000 (40s)	2000 (1m 20s)	3000 (2min)	4000 (2m 40s)
CPU	0.077s	1.089s	4.515s	9.729s	17.472s
GPU	0.648s	0.559s	1.234s	1.584s	1.926s

tion of texture coordinates within the shader, we can access the same histogram repeatedly as if it were a circular buffer. This means we only need to transfer the histogram once, in a $1 \times n$ texture (where n is the number of bins), which is vastly cheaper than transferring a $1 \times m$ texture (where m is the number of frames). This has the added bonus of being a constant cost regardless of the number of frames in the shot.

For the computation itself, as before we split it into a number of passes with different shaders. A breakdown of the shaders and what they do follows:

- Shader 1 subtracts each histogram in the first texture from the single histogram in the second texture, producing a buffer of the same size as the first texture. It also squares the result.
- Shader 2 takes the output of shader 1 and uses row reduction to sum the values in each row of the texture i.e. in each histogram, reducing the buffer to a column vector.
- Shader 3 takes the output of shader 2 and takes the square root of each element giving the vector distance between the current histogram and every other one.
- Shader 4 uses column reduction to sum every distance, reducing our column vector to simply one value.

The single output value of the final pass is then read back to the CPU, and subsequent processing is performed there. The value is divided by the number of frames to get the average vector distance between the current frame and every other frame. The average difference is stored in an array, and once every frame has been processed, we find the minimum of these values exactly as on the CPU, and mark its associated frame as the representative keyframe for the shot.

The results for keyframe selection on a variety of different shot lengths are presented in Table 2 showing performance for increasing numbers of frames in the shot being analysed (results averaged over 10 runs). As we can see, the results bear some interesting characteristics. For shorter shots, like 500 frames, the GPU is significantly outperformed by the CPU because there is a constant amount of once-off setup work that needs to be performed before any computation can take place on the GPU, including loading shader programs, and any initial texture input, etc.

However, as the shot length increases we see some dramatic improvements for the GPU. With 1000 frames and beyond, the GPU is easily faster, with a speedup of $\times 9$ for shots of 2 min 40 sec. This is because the keyframe selection algorithm is $O(n \times n)$ where n is the number of frames and the amount of computation required scales quadratically with more frames. This means that a GPU implementation of true *representative* keyframe selection is well-suited to rushes or raw video content, characterised by long shots with not much happening.

6. Conclusions

In this paper we have presented execution time performance figures for two video analysis and video structuring applications, shot boundary detection and representative keyframe selection, implemented on a conventional CPU and on a Graphics Processing Unit. Our findings show that for shot bound detection the GPU offers clear performance advantages while for keyframe selection the technique we use is well-suited for log shots such as found in rushes or stock footage. Indeed our GPU implementation yields keyframe selection in $1/80$ of real time for shots just under 3 minutes in length.

Our future work will involve testing the GPU implementation of these operations using even more recent GPUs compared against faster CPUs.

Acknowledgement: Part of this work was supported by Science Foundation Ireland under grant 03/IN.3/I361.

References

[1] P. Browne, C. Gurrin, H. Lee, K. M. Donald, S. Sav, A. F. Smeaton, and J. Ye. Dublin City University Video Track Experiments for TREC 2001. In *TREC 2001 - Text REtrieval Conference*, MD, USA, 2001. National Institute of Standards and Technology.

[2] I. Buck. *A Toolkit for Computation on GPUs*. Addison-Wesley, 2004.

[3] A. F. Smeaton, P. Over, and W. Kraaij. Evaluation Campaigns and TRECVid. In *MIR 2006 - 8th ACM SIGMM International Workshop on Multimedia Information Retrieval*, 2006.

[4] H. Zhang, A. Kankanhalli, and S. Smoliar. Automatic partitioning of full-motion video. *Multimedia Systems*, 1(1):10–28, 1993.