Accelerating Image Based Scientific Applications using Commodity Video Graphics Adapters

Randy P. Broussard
Systems Engineering Department, U.S. Naval Academy
Annapolis, MD 21402, USA

and

Robert W. Ives
Electrical and Computer Engineering Department, U.S. Naval Academy
Annapolis, MD 21402, USA

ABSTRACT

The processing power available in current video graphics cards is approaching super computer levels. State-of-the-art graphical processing units (GPU) boast of computational performance in the range of 1.0-1.1 trillion floating point operations per second (1.0-1.1 Teraflops). Making this processing power accessible to the scientific community would benefit many fields of research. This research takes a relatively computationally expensive image-based iris segmentation algorithm and hosts it on a GPU using the High Level Shader Language which is part of DirectX 9.0. The selected segmentation algorithm uses basic image processing techniques such as image inversion, value squaring, thresholding, dilation, erosion and a computationally intensive local kurtosis (fourth central moment) calculation. Strengths and limitations of the DirectX rendering pipeline are discussed. The primary source of the graphical processing power, the pixel or fragment shader, is discussed in detail. Impressive acceleration results were obtained. The iris segmentation algorithm was accelerated by a factor of 40 over the highly optimized C++ version hosted on the computer's central processing unit. Some parts of the algorithm ran at speeds that were over 100 times faster than their C++ counterpart. GPU programming details and HLSL code samples are presented as part of the acceleration discussion.

Keywords: Image processing, DirectX, GPU, graphics card.

1. INTRODUCTION

Research has been performed to utilize the computational power within video graphics cards for scientific computing tasks such as sparse matrix solutions [1], linear algebra operations [2], fast Fourier transforms [3], discrete wavelet transforms [4], [5], and image-based relighting [6]. This research will focus on the field of iris recognition to demonstrate GPU acceleration. The iris is currently believed to be one of the most accurate biometrics for human identification. Error rates of one in ten million have been achieved in production systems [7].

Iris processing demands

Current iris identification algorithms execute quickly on images that are controlled with respect to lighting, resolution, orthogonality and occlusion. When images are acquired under non-ideal conditions, additional image processing is often required. Non-orthogonal iris images (viewed from an angle other than perpendicular to the iris) require extra processing to transform the image to a viewing angle that is compatible for comparison to a stored orthogonal database [8]. Advanced image processing techniques that detect and remove the effects of lighting (glint) and occlusion (eyelids and eyelashes) can also introduce a processing delay in a real-time identification system. High resolution systems such as "Iris on the move" and "Iris at a distance" require the real-time processing of high resolution images. As processing demands grow, so does the need for increased computational power. This research takes a relatively computationally expensive iris segmentation algorithm and hosts it on a GPU using the High Level Shader Language (HLSL) which is part of DirectX 9.0. The goal of this research is to demonstrate that an image processing-based scientific algorithm can be greatly accelerated using commodity graphics adapters. The selected segmentation algorithm uses basic image processing techniques such as image inversion, value squaring, thresholding, dilation, erosion and a computationally intensive local kurtosis (fourth central moment) calculation to identify the pupil and limbic boundaries of the iris.

The graphics processing unit

One approach to accelerate an image-based scientific algorithm is to move the processing into dedicated hardware such as a math coprocessor or a Field Programmable Gate Array (FPGA). The powerful graphics processing unit (GPU) found in commodity video graphics cards provide a low cost and widely available alternative to these dedicated solutions. The fact that the speed of modern graphics hardware has grown at a rate of 3.0-3.7 every 18 months, while CPU speeds have only grown by a factor of 2.2 makes GPUs even more appealing [9]. A GPU is a special purpose processor that is optimized for graphical processing of triangle vertices and individual pixels. State-of-the-art GPUs claim theoretical computational performance in the range of 1.0-1.1 trillion floating point operations per second (1.0-1.1 Teraflops). When multiple GPUs are placed within standard computer systems, the processing power of a single computer can approach super computer levels. Another advantage of the video graphics card is that video memory bandwidth is often greater than the memory bandwidth within the host computer. The memory bandwidth of the fastest consumer video card (GeForce GTX 285, circa 2009) is currently 159 Gigabytes per second while the fastest 64-bit computer memory (DDR3-1600) has a bandwidth of 12.8 Gigabytes per second [10].

The Teraflop processing power of the GPU combined with the Gigabyte bandwidth of the video graphics card can provide tremendous acceleration for scientific applications. The GPU's operation is not coupled to the computer's CPU, thus both can run in parallel. GPU code that conforms to the DirectX specification (or some other high level interface) could automatically take advantage of advances in GPU performance as they appear [11]. Some code would not even require a recompile.

A GPU contains multiple pipelines which perform many graphics operations such as coordinate transformations, lighting effects, triangle texturing and pixels rendering. Only the pixel rendering pipeline was evaluated in this research

DirectX

DirectX is a Microsoft Windows-based Application Programming Interface (API) which offers programming functions that can access the graphical processing capabilities within a video graphics card. In DirectX, three-dimensional objects are formed using multiple triangles (facets). These triangles represent the surface area of the object. By manipulating the location, orientation and size of these triangles, the object can be moved to any location and orientation within a three-dimensional space. By manipulating the texture and color within these triangles, many lighting and visual effects can be produced. The entire DirectX framework is based on scaling and rotating a set of triangles, and geometrically applying lighting and texture to those triangles. In current video graphics cards, these functions are accelerated in hardware. Once the triangle manipulation is complete, the three-dimensional objects are projected onto a two-dimension plane which represents the output screen. The final stage of the DirectX pipeline is a high speed Arithmetic Logic Unit (ALU), called a pixel shader, which is used to manipulate the output image on a pixel-by-pixel basis.

The pixel shader

The pixel shader (or fragment shader), is the primary source of the graphical processing power utilized to perform this research. When stored in video memory, a pixel is defined to contain four color components; the colors red, green, blue and an extra component. Many GPUs have processing pipelines that are 128-512 bits wide. These wide pipelines allow all four pixel components to be processed simultaneously. Pixel components can range from 32-bit to 128-bit floating point values. If desired, this parallelism could also be used to simultaneously process four grayscale images by loading each image into a separate color plane. Current GPUs have as many as 240 pixel shaders that operate in parallel. This means a GPU can process 240 pixels simultaneously. Many pixel shaders have multiple arithmetic logic units and can perform multiple mathematical operations in parallel [12]. Figure 1 illustrates some of the processing properties of a pixel shader.

The GPU simultaneously executes identical instructions on each available pixel shader to process individual image pixels. This simultaneous execution of multiple pixel shaders forms a Single Instruction Multiple Data (SIMD) architecture [11], [13]. This architecture is highly parallel, but also introduces several significant restrictions on algorithm flow [12]. A pixel shader

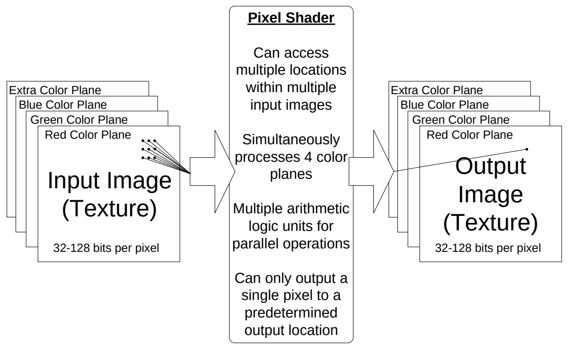

Fig. 1. The high level architecture and functionality of a hardware pixel shader contained within a graphics processing unit.

can operate on multiple input pixels, but the output value is always placed in a separate output image. This means the pixel shader is highly suitable for neighborhood operations such as filtering and morphology [14]. Since the output image is separate from the input image, no in-place processing can be performed. The pixel shader also does not have access to the output of other pixel shaders, thus no global image processing can be performed in a single pass. This restriction causes the GPU to be less suitable for global image calculations such as determining the mean or standard deviation of an image.

2. APPROACH

To demonstrate the acceleration afforded by using a video card's GPU, portions of a computationally intensive iris segmentation algorithm were implemented in the GPU. This segmentation algorithm uses image inversion, value squaring, dilation, erosion and a computationally intensive local kurtosis calculation to identify the pupil and limbic boundaries of the iris [15]. The general steps within the algorithm can be seen in Fig. 2. Additional details about the algorithm can be found in [15]. As a proof of concept, only the portions of the algorithm that were easily ported and were suitable to processing in the GPU were attempted. It is possible that the overall algorithm could be modified to enhance parallelism, but no attempt was made.

To measure acceleration, sample images were processed using the original CPU hosted functions and the video graphics card

1. Find pupil boundary:
   a. Invert and square pixel values.
   b. Apply statistical threshold.
   c. Dilate 15 pixels.
   d. Erode 15 pixels.
   e. Group connected pixels into objects.
   f. Select object that is most pupil-like.

2. Find limbic boundary:
   a. Compute local kurtosis of image.
   b. Find areas of low kurtosis and convert to a binary image
   c. Fit an annulus the the arcs that correspond to the location of the limbic boundary

Fig. 2. General steps performed to segment iris.

hosted functions. Average execution times for each version of the function are presented and compared to determine algorithm acceleration. The computer system clock was used to measure execution time. Since the available C language clock function had a resolution of 15 milliseconds, each function was executed 1000 times and the average execution time was used for comparison. All CPU executed video graphics code was written using the DirectX 9.0 interface. The DirectX High Level Shader Language for pixel shader version 2.0b was used to compose all GPU executed code. All CPU code was executed on an AMD Athlon X2 3800+ dual core system with 4 gigabytes of memory. The GPU code was executed on a NVidia GeForce 7900 GT video card containing 512 Mbytes of memory. Both the computer system and the graphics card were near state-of-the-art in mid-2006, thus representing comparable technologies.

3. PROGRAMMING THE PIXEL SHADER

To upload an iris image to the video graphics card, the image is copied to a user-defined texture map that is created in video memory. The first step is to create an object that occupies the entire output region of the DirectX pipeline. This is done by locating two triangles in three-dimensional space to represent a rectangle, that when projected onto the output screen will exactly cover the output screen.

The pixel shader is used map the input texture to the triangles located in the output image. Figure 3 depicts the mapping of the texture (iris image) to the two triangles. A pixel shader will be called once by the GPU for each pixel that lies within the defined triangles. With each call to a pixel shader, the GPU passes the x and y location of a single output pixel to the shader. The pixel shader is expected to produce a four component (red, green, blue, extra) value for that pixel. To perform image processing, the pixel shader can access pixels from one or more input images (called textures), process that information and pass the result back to the GPU for storage in the output image. This process is repeated, by the GPU, for each output pixel within the defined triangles.

The pixel shader can access multiple pixels from one of many input images, but has no access to the output image. The output triangles can be mapped to any location in the input image. Locating the triangle so they only cover a portion of the input image causes only that portion of the input image to be processed. This negates the need for image cropping and can accelerate processing.

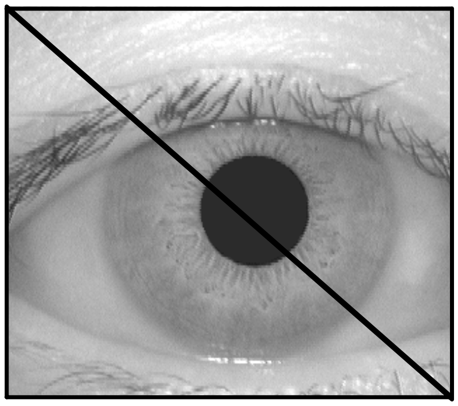

Fig. 3. Two triangles are used to define the image region processed by the DirectX rendering pipeline [16].

The pixel shader can be programmed in assembly language, or in a High Level Shader Language which syntactically is nearly identical to the C programming language. Looping constructs and logical tests are supported, but can have a negative impact on performance. Loops that have been unrolled (restructured as a finite sequence of sequential steps) provide better performance. Pixel shader code should be short and simple to enhance the GPU compiler's ability to optimize the code for the available hardware. Fig. 4 shows an example of the pixel shader code used to perform a four-connected one-pixel morphological dilation (eight connected dilation was used in this research). The float2 and float4 data types are arrays of two and four floating point values respectively.

The GPU processes all pixel values using 32-bit floating point math and normalizes them by default. Note the floating point offsets used to access neighboring pixels. The pixel shader normalizes all image coordinates to be within the range 0.0 to 1.0 (inclusive). Initially, achieving precise pixel alignment was a significant challenge when mapping a texture to triangles [14]. See the DirectX documentation for more information on this topic [17].

Since the pixel shader does not have access to the output image, many algorithms will need to be executed in discrete sequential stages. To execute a multi-step algorithm in the GPU, a technique known as ping-ponging is used [14]. A traditional GPU program would process each screen pixel once and render the output to the viewer's screen. To use this output as input to another processing stage, the GPU is configured to render to a texture map instead of the screen. By using two textures and alternating which is input and which is output, the GPU can

```
float4 PixelShaderDilate(float2 PixelCoord :
                         TEXCOORD0) : COLOR
{
    float4 output;
    const float Dx=1.0f/1280.0f;
    const float Dy=1.0f/960.0f;

    // sample neighborhood in texture
    PixelCoord.y += Dy; // check pixel above
    output = tex2D(BaseTex, PixelCoord);

    PixelCoord.y -= 2*Dy; // check pixel
below
    output += tex2D(BaseTex, PixelCoord);

    PixelCoord.x += Dx; // check pixel to
    PixelCoord.y += Dy; // the right
    output += tex2D(BaseTex, PixelCoord);

    PixelCoord.x -= 2*Dx; // check to left
    output += tex2D(BaseTex, PixelCoord);

    // binarize and return output value
    output = saturate(output);
    return output;
}
```

Fig. 4. An example of the pixel shader code used to perform a four-connected one-pixel morphological dilation.

perform multi-step image processing. Both textures exist in the graphics card memory, thus the GPU can process the data at full speed.

Multiple pixel shader programs can be compiled and passed to the GPU in any order at run time. Images processed by the GPU can be retrieved by locking the video card's memory and copying the image data back to system memory. The GPU's execution runs in parallel with and is decoupled from the CPU. Function calls to the GPU place a task request in the GPU's queue and return immediately. This means the system CPU can perform other tasks while the GPU executes its tasks. The only time the two processing units are synchronized is when the GPU's memory is locked for a data transfer. The locking mechanism will wait until the GPU has finished modifying the render target before giving the CPU access.

Processing a 1280 x 960 image			
Processing Step	***CPU***	***GPU***	***Acceleration***
Invert and Square Values	58 mS	1 mS	58.0 times
Threshold Values	13 mS	1 mS	13.0 times
Dilate (x15)	328 mS	8 mS	41.0 times
Erode (x15)	830 mS	8 mS	103.8 times
Transfer Image to/from VGA	n/a	17 mS	n/a
Convert Image type	140 mS	n/a	n/a
Total	1369 mS	35 mS	39.1 times

Table 1. Execution times for the CPU- and GPU-based functions, and the acceleration achieved.

4. RESULTS

All image processing functions were accelerated using the video graphics card. As can be seen in Table 1, the acceleration ranged from 13 times the original speed to 103 times the original speed. The longer, more complex functions, such as erosion, were accelerated to a greater degree than the shorter, less complex functions such as thresholding. As the number of steps in an algorithm increase, the possibility for parallelism and optimization also increase. The overall iris segmentation algorithm was accelerated 39 times the speed of the CPU-based algorithm. Transferring the image to and from the video card took nearly as much time as the total combined operations performed within the video card. Due to the overhead of transferring images between system memory and video memory, longer, more complex algorithms would gain greater benefit from GPU acceleration than shorter, less complex ones. Note that when image transfer time is included, performing thresholding within a video graphics card takes longer than performing this function on the CPU.
While conducting the experiments, it was found that performing a large morphological operation in many small steps was faster than performing the entire operation in one step. For example, it was much faster to perform 15 one-pixel dilations than to perform a single 15-pixel dilation. The authors theorize that the slow down is due to the size of the pixel shader's program cache or due to the inability of the GPU compiler to optimize the more complex looping code required for the larger dilation. It was also found that logical tests introduced a speed penalty. It was faster to dilate an entire image than it was to omit processing of "on" pixels using an if(…) statement.

Acceleration as image resolution increases			
Resolution	***CPU***	***GPU***	***Acceleration***
320 x 240	130 mS	4 mS	31.0 times
640 x 480	414 mS	11 mS	37.6 times
1280 x 960	1369 mS	35 mS	39.1 times
1920 x 1440	3113 mS	76 mS	40.8 times
2560 x 1920	7123 mS	134 mS	53.0 times

Table 2. The effects of image size on execution time and acceleration.

To determine the effect of image size on acceleration, all steps listed in Table 1 were performed on images of various sizes. The various sized images were scaled versions of the 1280x960 resolution images used in the previous experiment. Table 2 shows how acceleration is affected by image size. For smaller images, the processing time for both the CPU and GPU code increased linearly as image size increased. As image size grew, the processing time for the CPU code grew at a faster rate than the processing time for the GPU code. Thus, acceleration increases as image size increases.

Figure 5 shows a plot of the processing time vs. image size. Note the line representing CPU performance has two distinct slopes. The reason the line has two slopes is that the CPU contains an internal two megabyte (Mbyte) level two memory cache. When image size exceeded two Mbytes, image processing could no longer be performed entirely within the CPU. Accessing system memory introduced a performance penalty resulting in the distinct second slope for images larger than two Mbytes. The GPU contains no internal cache and directly accesses video memory which is typically faster than system memory. As can be seen in Fig. 5, the GPU processing time scaled linearly as image size increased. It should be noted that all experiments were performed using only one of the four color planes within each pixel. If the image were divided into four and placed into all four planes, the GPU execution times should theoretically decrease by a factor of four. Accelerations of 10 to 100 times the CPU based algorithm speed have also been cited in [18] and [19]. This indicates that the current

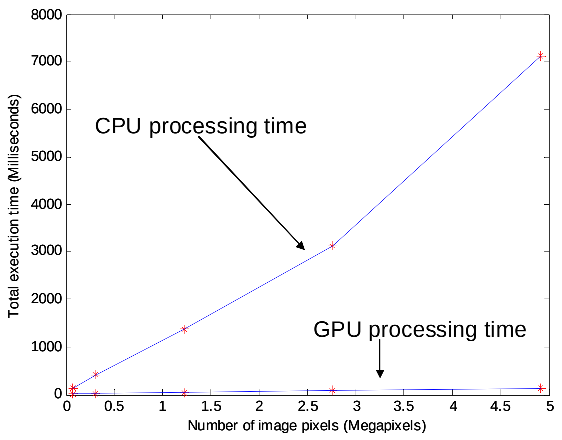

Fig. 5. Plot of the effects of image size on execution time.

function implementations are taking advantage of the inherent parallelism in the GPU architecture.

5. CONCLUSION

Using a video graphics card can accelerate image-based scientific algorithms by a factor of 10 to 100 times the speed of a CPU based algorithm. The acceleration achieved on video graphics cards is largely unaffected by and scales linearly with image size. Some longer, more complex algorithms will execute more quickly if the algorithm is divided into many small steps versus performing the entire operation in one step. The DirectX pipeline is complex and highly parallel which presents many technical challenges when performing global image processing functions such as summation and average value computation. Overall, commodity video graphics adapters have proven to be a useful tool in accelerating the performance of computationally intensive algorithms.

6. REFERENCES

[1] J. Bolz, I. Farmer, E. Grinspun, P. Schreoder, “Sparse matrix solvers on the GPU: Conjugate gradients and multigrid”, in **ACM Trans. Graphics**, 2003.

[2] J. Kruuger, R. Westermann, “Linear algebra operators for GPU implementation of numerical algorithms”, in **ACM Trans. Graphics**, 2003.

[3] K. Moreland, E. Angel, “The FFT on a GPU”, in **Proc. HWWS**, 2003.

[4] T.-T. Wong, C.-S. Leung, P.-A. Heng, J. Wang, “Discrete Wavelet Transform on Consumer-Level Graphics Hardware”, **IEEE Transactions on Multimedia**, Vol. 9, No. 3, pp. 668-673, April 2007.

[5] C. Tenllado, et. al., “Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting”, **IEEE Transactions on Parallel and Distributed Systems**, March 2008 (Vol. 19, No. 3) pp. 299-310.

[6] T.-T. Wong, S.-H. Or, C.-W. Fu, “Real-time relighting of compressed panoramas”, in **Graphics Programming Methods**, J. Lander, Ed. New York: Charles Rivers Media, 2003, pp. 375-388.

[7] J. Daugman, "Probing the uniqueness and randomness of IrisCodes: Results from 200 billion iris pair comparisons." **2006 Proceedings of the IEEE**, vol. 94, no. 11, pp 1927-1935.

[8] L.R. Kennell, R.N. Rakvic, R.P. Broussard, R.W. Ives, "Segmentation of Off-Axis Iris Images", published as a chapter in the **Biometrics Encyclopedia**, Springer publishing, winter 2008.

[9] M. Wloka, “Batch, Batch, Batch: What does it really mean?”, **Game Developers Conference**, http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.pdf.

[10] http://www.nvidia.com/object/product_geforce_gtx_285_us.html, accessed 31 July 2009.

[11] V. Moya, et. al., “Shader Performance Analysis on a Modern GPU Architecture”, **Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture**, 0-7695-2440-0/05, 2005

[12] WO02103638: Programmable Pixel Shading Architecture, technical document from NVIDIA CORP., December 27, 2002.

[13] A. Purde, et. al., “Pixel shader based real-time image processing for surface metrology”, **2004 Instrumentation and Measurements Technology Conference**, Como, Italy, May 18-20, 2004.

[14] J.L. Mitchel, “Image Processing with 1.4 Pixel Shaders in Direct3d”, an excerpt from **ShaderX: Vertex and Pixel Shader Tips and Tricks**, Wordware Publishing Inc., 2002, ISBN 1-55622-041-3,

[15] L.R. Kennell, R.W. Ives, R.M. Gaunt, “Binary morphology and local statistics applied to iris segmentation for recognition,” **Proceedings of the 13th Annual International Conference on Image Processing**, Oct. 2006, in press.

[16] [2] Monro, D. M., Rakshit, S., and Zhang, D, University of Bath, U.K. Iris Image Database, http://www.bath.ac.uk/elec-eng/pages/sipg/irisweb.

[17] http://msdn.microsoft.com/en-us/directx/default.aspx.

[18] J.D. Owens, et. al., “A survey of general-purpose computation on graphics hardware”, **Computer Graphics Forum**, vol. 26, 2007.

[19] B. Han, B. Zhou, “High Speed Visual Saliency Computation on GPU”, **2007 IEEE International Conference on Image Processing**, San Antonio, TX, September 16-19, 2007.