COMPUTER VISION SIGNAL PROCESSING ON GRAPHICS PROCESSING UNITS

James Fung and Steve Mann

ECE Department
University of Toronto
10 King's College Road
{fungja,mann}@eecg.toronto.edu

ABSTRACT

In some sense, computer graphics and computer vision are inverses of one another. Special purpose computer vision hardware is rarely found in typical mass-produced personal computers, but graphics processing units (GPUs) found on most personal computers, often exceed (in number of transistors as well as in compute power) the capabilities of the Central Processing Unit (CPU). This paper shows speedups attained by using computer graphics hardware for implementation of computer vision algorithms by efficiently mapping mathematical operations of computer vision onto modern computer graphics architecture. As an example computer vision algorithm, we implement a real–time projective camera motion tracking routine on modern, GeForce FX class GPUs. Algorithms are implemented using OpenGL and the nVIDIA Cg fragment shaders. Trade–offs between computer vision requirements and GPU resources are discussed. Algorithm implementation is examined closely, and hardware bottlenecks are addressed to examine the performance of GPU architecture for computer vision. It is shown that significant speedups can be achieved, while leaving the CPU free for other signal processing tasks. Applications of our work include wearable, computer mediated reality systems that use both computer vision and computer graphics, and require realtime processing with low–latency and high throughput provided by modern GPUs.

1. INTRODUCTION

Computer vision algorithms can often be computationally intensive, and many applications require them to be run in real–time, at video framerates. This requirement can easily tax a computer's Central Processing Unit (CPU).

Modern graphics cards now incorporate a processor chip, commonly referred to as the Graphics Processing Unit, or GPU [1]. Typically, the GPU is a Single Instruction, Multiple Data (SIMD) processor which is now capable of performing arbitrary, programmable operations on data sent to it. This paper investigates how to efficiently map computer vision algorithms onto graphics cards so that they can run completely upon the GPU, leaving the CPU free for other tasks.

Under consideration is the NVIDIA NV35 chip which is the GPU found on the GeForce FX 5900 Ultra boards which we used in testing. Present day GPUs often have more transistors than any other part of a computer system, and are thus rapidly becoming the best place in a computer to implement high speed computation. The NV35 GPU, for example, has a transistor count of about 135 million, compared to the Xeon CPU that has only 108 million transistors, of which about two thirds of the CPU's transistors implement cache memory and not arithmetic. GPUs, such as the GeForce FX 5900, on the other hand, are primarily made up of a number of pixel pipelines implementing floating point arithmetic. Modern GPUs now provide standard IEEE Floating Point precision. This capability can thus be exploited by computer vision algorithms as well as for use in computer graphics.

Thanks to NSERC, SSHRC, Canada Council for the Arts, Ontario Arts Council, Toronto Arts Council, and Ontario Graduate Scholarships/Lewfam Foundation Scholarships in Science and Technology agency for support. Thanks to nVIDIA, ATI, and Viewcast for equipment donations.

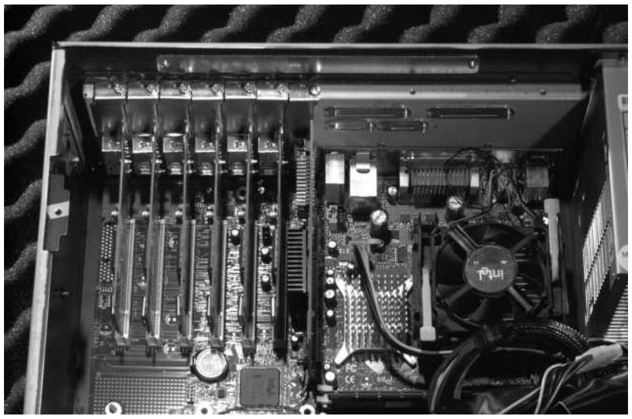

Fig. 1. A computer vision machine constructed in July 2002 with 6 capture cards designed for a simultaneous 6 channel capture application, which required fast processing to display projected versions of the images, using computer graphics hardware, prompting the investigation of applying computer graphics hardware to computer vision.

2. RELATIONSHIP BETWEEN GRAPHICS AND VISION

It has often been said that computer vision and computer graphics are closely related, being inverses of the same problem. Computer graphics can be considered image synthesis in that it takes a mathematical description of a scene and produces a 2D array of numbers, which is an image. Computer vision can be considered a form of image analysis, taking a 2D image and converting it into a mathmatical description. The mapping of the image processing and computer vision algorithms into computer graphics hardware explicitly and practically exposes the relationship between these operations. For instance, it has been shown that the process of image registration using a algebraic projective geometry is isomorphic to the process of projecting a texture mapped polygon under perspective projection in computer graphics [2].

In the same fashion, modern graphics requires a number of operations to be performed on an incoming fragment generated from a mathematical representation of a desired scene. These are operations such as geometric transformations, lighting, reflection, texture mapping and so on which are done in order to generate a final output pixel value. Similarly, for computer vision, a low–level algorithm will perform a number of operations on an input pixel value. After the processing is done, a final output is produced which characterizes the input image as a mathematical construct of some significance. Despite the inverse nature, these processes are both characterized by a high degree of local processing which must occur per pixel (or in a small region, achieved perhaps by filtering).

3. BACKGROUND

Some approaches to fast hardware solutions for computer vision have included focal plane processors [3], dedicated architectures [4], and FPGA solutions [5]. Computer graphics architectures have also been applied towards general purpose computing and matrix operations[6, 7] and image processing [8], as well as simulation. [6, 7] present frameworks for implementing matrix operations on graphics hardware, however, they did not have available at the time modern graphics architectures, and could only achieve limited accuracy. Computer graphics hardware has also been used to accelerate the calculations of morphological filters as used for in volume analysis for volume rendering applications [9].

However, full computer vision algorithms have not yet been implemented - rather only particular operations have been conducted on the graphics card.

4. ALGORITHM DESCRIPTION

VideoOrbits is an image registration algorithm which calculates a coordinate transformation between pairs of images of a static scene, taken with a camera that is free to pan, tilt, rotate about its optical axis, and zoom. The technique solves the problem for two cases: 1. images taken from the same location of an arbitrary 3-D scene, or 2. images taken from arbitrary locations of a flat scene [10].

This algorithm has proved useful in creating wearable, tetherless, computer mediated reality [11]. In a mediated reality, a user sees their environment through some kind of video eye–glasses like device which shows to the user a computer altered version of what they would otherwise see. By registering successive images seen by the user, computer generated information can be added into the environment and appear affixed to the environment, or computer information can be removed, or blocked from the user's view (e.g. removal of unwanted advertising). Thus the tracking must run in real–time, and at video framerates. A low latency, and high throughput solution is required.

VideoOrbits solves for a projection of an image to register it with another image as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}}{\begin{bmatrix} c_1 & c_2 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + 1} \qquad (1)$$

The algorithm is a repetitive, multiscale algorithm. The majority of the computational intensity is found in the calculation of an 8x8 square matrix and an 8 entry column matrix, whose entries are the sums of products of image derivatives and coordinates over the entire image. This 8x8 matrix is upper diagonal, which means that there are only 36 unique values. Thus, 44 unique sums of products must be calculated.

The operations of downsampling, low pass filtering, and forming a least-squares like approximation to determine a solution are also common operations found in a variety of computer vision algorithms. Thus, the techniques and analysis here can be applied to different problems.

5. TECHNIQUES FOR APPLYING GRAPHICS PROCESSING UNITS FOR IMAGE ANALYSIS

Modern graphics processors now support fragment shader programs. After the geometry of computer graphics primitives (triangles, and texture coordinates etc.) is calculated, and their position determined after transformation by the camera viewing matrix, the result is a fragment which is to be rendered onto the screen. Each fragment now has a known position on the screen (i.e. it has been rasterized). These fragments are then each processed by a fragment processor. The fragment processor is now implemented as a general purpose processor with an instruction set capable of carrying out typical mathematical functions. The fragment processor thus runs a fragment program on the incoming fragment. In graphics, this is the evolution of procedural shading [12, 13]. The same fragment program is run for each fragment to be displayed.

Fragment programs are a natural place to carry out image processing and computer vision algorithms because they operate directly on each pixel. The GeForce FX class of GPU provides fragment shaders which are capable of 32 bit (per each of R,G,B, and A components, for a total of 128 bit width) floating point calculations. They are also capable of doing texture lookups at arbitrary coordinates. The per–pixel functionality provided by the fragment shaders makes them a natural for image processing and computer vision.

5.1. Blurring and downsampling, and derivatives

The following Cg [14] code (unrolled by the Cg compiler) can be used to create a simple low–pass filtered image:

```
float4 diff = {0.0,0.0,0.0,0.0};
for( i=-1 ; i<=1; i++ ) {
 for( j=-1 ; j<= 1; j++ ) {
  diff+= texRECT(texture,Coord+float2(i,j));
} }
diff = diff/9.0;
```

In the case that the downsampled image is desired, the image can be texture mapped onto a quadrilateral which fills a window which is one quarter the size of the original image. In this way, the fragments displayed will be every other image coordinate, and the fragment program will only be run at the desired pixels.

Derivatives can be easily calculated in the fragment program as well. To do this, we use the multitexturing capabilities found on graphics cards. This allows more than one texture to contribute to the final value of a texture. Results are stored as 32-bit floating point numbers.

We require image derivatives D_x, D_y, and D_t (derivatives with respect to each axis and a temporal derivative between successive frames). By placing two successive frames in two different texturing units, we can access each. A simple image derivative is formed by the following:

```
float4 Dx =
  texRECT(tex1, Coord1 + float2(0.0,-1.0)) -
  texRECT(tex1, Coord1 + float2(0.0, 1.0));
float4 Dt =
  texRECT(tex1,Coord1)-texRECT(tex2,Coord2);
```

Again, these run on each fragment sent to the frame buffer, thus calculating derivatives for the whole image. Though graphics cards have register combiners which can perform color blending operations, these are not accessible for the full floating point precision, thus the temporal derivative must be explicitly calculated.

5.2. Projective Texture Lookups, and Lightspace conversion

Fragment shaders can perform arbitrary lookups. Thus, on a subsequent repetition of the algorithm, two images are compared, with one projected by the 8 parameters of equation 1 estimated in the previous repetition. The image value at the projected coordinate must be looked up.

This can be achieved in several ways: 1. By setting the appropriate OpenGL project matrix, 2. By performing a projective texture lookup, or 3. By explicitly calculating and looking up the desired texture. We implemented the latter option because it was the most direct way to ensure the GPU estimation and CPU estimation were as closely matched as possible. A 4x4 projection matrix can be sent to the fragment shader, and texture coordinates explicitly calculated according to equation 1

Lookup tables can be easily placed in a 1D texture, and used for easy imagespace to lightspace [15] conversion, and vice versa. Furthermore, arbitrary equations can be placed in the fragment shading program for more complex transformations on the domain.

5.3. Render to Texture

When calculated values are to be repeatedly used it is beneficial to store them in texture memory on the card. The image derivatives are re–used frequently in the algorithm and generating them involves a number of texture lookups, which requires fetching texture values the texture memory, taxing the texture bandwidth of the graphics card.

Instead, the processed image is rendered into the graphics card texture memory, avoiding making an AGP bus transfer. This texture is then rendered in subsequent passes.

5.4. Summations

Consider the summation:

$$\sum_{x=0}^{x=X}\sum_{y=0}^{y=Y}(x^2D_x + xyD_y)(xyD_x + y^2D_y) \qquad (2)$$

which is a summation found in the linear least squares step of VideoOrbits. x and y are image coordinates, and D_x, D_y, D_t are the image derivatives with respect to x y and time t.

Evaluation of the equation within in sums is straightforward in a fragment shader, since the texture coordinates are known, and the derivatives can be calculated as above. However, the above must be summed across the entire image. One solution would be to readback the entire framebuffer, and perform an addition. While this reduces the computation load on the CPU (since it only needs to sum the output of the fragment shader across the image), readback takes a considerable amount of time.

Because readback can be the slowest part of the computation, it is desirable whenever possible to reduce the amount of information which needs to be read back from the graphics card. When only a summation of the image is required, fragment shader programs can be used to conduct a partial summation. What can be done, is that only a fraction of the entire texture map need be displayed. At each texture coordinate, the suA of that coordinate, and a regular pattern of scanlines below (or nearby values) is calculated. Displaying only a portion of the texture map prevents unnecessary calculations by the fragment shader. After this fractional part of the texture has been rendered, it can then be read back into the CPU. Four unique summations can be readback each pass, placing them in each of the four framebuffer colour channels (red, green, blue, and alpha). Thus, the readback is much faster because less memory from the framebuffer is read. In cases where the complete summation cannot be carried out by the fragment shader (for instance, because of the limitation of the number of instructions which can be placed into a pixel shader program), the final summation can be carried out on the CPU, but on the already partially summed, smaller, buffer. This reduces the processing load for the CPU. This technique is similar to that found in [6], and used in parallel processing to maintain a low memory footprint. Such balancing of resources in multipass algorithms on GPU computation is also discussed in [16].

6. ARCHITECTURAL PERFORMANCE

The GeForce FX 5900 completes 145 estimations each second, where as the same process, running on the CPU (an AMD Athlon running at 2.0 GHz), takes 3.5 seconds to complete the same number of estimations. Thus, the speed up provided by the GPU is 3.5x. A GeForce FX 5200 card runs 41 estimations per second, which makes it equivalent to the processing power of an AMD Athlon 2.0 GHz.

Operation	FX5200	FX5900
Derivative Render	5.7 msecs	0.88 msecs
Render to Texture Memory	0.7 msecs	0.23 msecs
Fragment shaders (10 passes)	15.34 msecs	4.38 msecs
Readback (10 passes)	3.9 msecs	1.26 msecs

6.1. AGP and CPU Effects

The AGP bus is used when an image from the camera is sent to the GPU for processing, and then again when the results are read back from the graphics card. The AGP bus proved to not be a bottleneck for 20 frame per second 320x240 images. This was tested by running the program at the 4x and 8x AGP speed supported by the motherboard and video card. The performance was identical for both AGP speeds.

At 1x downsampling, an estimation rate of 135 repetitions per second was observed at 800 MHz, 145 at 1.4 GHz, and 148 at 2.0 GHz. It was concluded that the CPU speed made little difference on the algorithm performance. Thus, this shows that our implementation leaves the CPU free to run other tasks.

6.2. Fragment Program Length

From figure 2, a linear least squares approximation to the trend line gives a 2.7 millisecond overhead, with the time required to run increasing roughly linearly with more instructions. The overhead is considered due to the downsampling, and rendering to texture memory required between each pass of the algorithm. The additional instructions represent additional iterations of identical loops (thus there are the same number of multiplies and texture lookups added as more instructions are added).

The effect of the readback calls was also examined by removing the readback between passes. From the graph, it is seen that this resulted in the reduction of the estimation time by about 1.3 milliseconds, consistent regardless of the number of fragment program instructions. The implementation shows that 80 percent of the time is spent in the fragment shading process. Thus, the implementation is fragment limited, but that is not the only bottleneck.

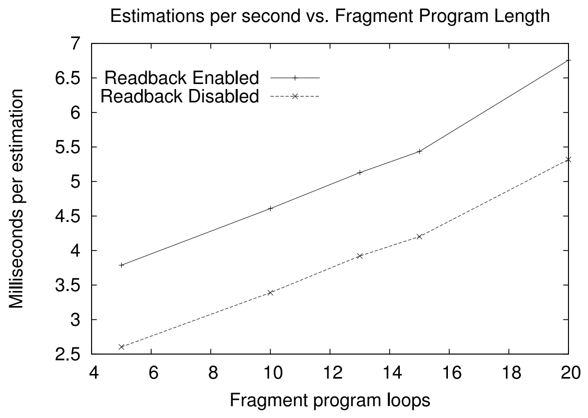

Fig. 2. Time per estimation vs. Fragment Program Length.

Because of the overhead, it is also likely that the fragment shaders are not being utilized all of the time, but lie idle during some parts of the rendering (during readback for instance).

7. PARALLEL COMPUTATION ON MULTIPLE GPUS

We have also designed and built various kinds of architectures that use multiple graphics cards in a single system. For example, we put together a system with six ATI video capture/display graphics cards, a system with seven nVIDIA cards (one AGP plus six PCI), and a system with six video capture (Viewcast) cards for doing computer vision [1]. In some of these systems we also implemented a "fuzzy bus" by using the I/O of the TV tuners on graphics cards to frequency-division-multiplex approximate array values onto a bus formed by connecting some or all of the TV tuners together, with each card assigned a different TV channel (e.g. TV channels 2 through 7).

The use of multiple (6) GPUs has advantages for parallel computation. Firstly, each card has its own RAM. Thus, they can access their memories in parallel which increases the overall memory bandwidth of the system. The GPUs do not contend with each other for access to a shared memory area. The fragment programs are also stored locally on each graphics card, and thus they can run relatively independently, requiring little supervision from the CPU. Additionally, OpenGL's display list functionality stores OpenGL calls on the graphics card, further reducing CPU overhead.

We have implemented an eigenspace image recognition system, and preliminary results have provided a 4.5x speedup when 5 PCI cards are used, over the single PCI case. Each PCI card runs at about 0.96x the speed of an AMD 2800+ CPU, and it was noted the primary bottleneck in the CPU case was the RAM (333 MHz DDR). Thus, much of the speedup likely results from the parallel RAM access provided by the multiple graphics cards.

8. CONCLUSION

Although other special purpose hardware systems could be used to provide hardware acceleration of computer vision algorithms, the low cost and widespread availability of computer graphics hardware will make hardware accelerated computer vision algorithms more accessible. Given the success of our computer mediated reality algorithms running in graphics hardware, we have shown that graphics processors provide new and useful ways of doing computer vision calculations.

[1] for the automation of a six-person column shower in a mass casualty decontamination facility (six cameras embedded in the shower column for user-tracking), August 29th, 2002 (http://deconference.com)

9. REFERENCES

[1] Erik Lindholm, Mark J. Kilgard, and Henry Moreton, "A user–programmable vertex engine," in *Computer Graphics, Proc. of SIGGRAPH 2001*, 2001, pp. 149–158.

[2] James Fung, Felix Tang, and Steve Mann, "Mediated reality using computer graphics hardware for computer vision," in *Proceedings of the International Symposium on Wearable Computing 2002 (ISWC2002)*, Seattle, Washington, USA, Oct. 7 – 10 2002, pp. 83–89.

[3] Shingo Kagami, Takashi Komuro, Idaku Ishii, and Masatoshi Ishikawa, "A real–time visual processing system using a general purpose vision chip," in *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*, Washington, DC, USA, May 2002, pp. 1229–1234.

[4] Peter J. Burt, "A pyramid-based front-end processor for dynamic vision applications," *Proceedings of the IEEE*, vol. 90, no. 7, pp. 1188–1200, July 2002.

[5] A. Darabiha, J.R. Rose, and W.J. MacLean, "Video rate stereo depth measurement on programmable hardware," in *IEEE Conference on Computer Vision & Pattern Recognition*, June 2003, pp. 203–210.

[6] Jens Krueger and Ruediger Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 908–916, 2003.

[7] C. Thompson, S. Hahn, and M. Oskin, "Using modern graphics architecture for general-purpose computing: A framework and analysis," in *SIGGRAPH/Internation Symposium on Microarchitecture*, Turkey, Nov. 2002.

[8] Anthony Sherbondy, Mike Houston, and Sandy Napel, "Fast volume segmentation with simultaneous visualization using programmable graphics hardware," in *To appear in IEEE Visualization 2003*.

[9] M. Hopf and T. Ertl, "Accelerating Morphological Analysis with Graphics Hardware," in *Workshop on Vision, Modelling, and Visualization VMV '00*. 2000, pp. 337–345, infi x.

[10] Steve Mann, "Wearable computing: Toward humanistic intelligence," *IEEE Intelligent Systems*, vol. 16, no. 3, May/June 2001.

[11] Steve Mann and James Fung, "Eye tap devices for augmented, deliberately diminished or otherwise altered visual perception of rigid planar patches of real world scenes," *Presence*, vol. 11, no. 2, pp. 158–175, 2002.

[12] K. Proudfoot, W. R. Mark, S. Txvetkov, and Hanrahan P., "A real-time procedural shading system for programmable graphics hardware," in *Proceedings of ACM SIGGRAPH (2001)*. The Eurographics Association 2002, 2001, pp. 159–170.

[13] Tiberiu S. Popa Michael D. McCool, Zheng Qin, "Shader metaprogramming," in *Proceedings of the conference on Graphics hardware 2002*. Eurographics Association, 2002, pp. 57 – 68.

[14] W. Mark, R. Glanville, K. Akeley, and M. Kilgard, "Cg: A system for programming graphics hardware in a c–like language," in *Proceedings of ACM SIGGRAPH. ACM Press, 2003*, July 2003, vol. 22.

[15] S. Mann, C. Manders, and J. Fung, "Painting with looks: Photographic images from video using quantimetric processing," in *ACM Multimedia 2002 (to appear in)*, Juan Les Pins, France, December 1-6, 2002.

[16] E. Chan, R. Ng, Pradeep Sen, K. Proudfoot, and P. Hanrahan, "Effi - cient partitioning of fragment shaders for multipass rendering on programmable graphics hardware," in *SIGGRAPH/Eurographics workshop on graphics hardware*, 2002, pp. 1–11.