OpenVIDIA: Parallel GPU Computer Vision

James Fung, Steve Mann, Chris Aimone
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada
{fungja,mann}@eecg.toronto.edu, aimone@eyetap.org

ABSTRACT

Graphics and vision are approximate inverses of each other: ordinarily Graphics Processing Units (GPUs) are used to convert "numbers into pictures" (i.e. computer graphics). In this paper, we propose using GPUs in approximately the reverse way: to assist in "converting pictures into numbers" (i.e. computer vision). The OpenVIDIA project uses single or multiple graphics cards to accelerate image analysis and computer vision. It is a library and API aimed at providing a graphics hardware accelerated processing framework for image processing and computer vision. OpenVIDIA explores the creation of a parallel computer architecture consisting of multiple Graphics Processing Units (GPUs) built entirely from commodity hardware. OpenVIDIA uses multiple Graphics Processing Units in parallel to operate as a general-purpose parallel computer architecture. It provides a simple API which implements some common computer vision algorithms. Many components can be used immediately and because the project is Open Source, the code is intended to serve as templates and examples for how similar algorithms are mapped onto graphics hardware. Implemented are image processing techniques (Canny edge detection, filtering), image feature handling (identifying and matching features) and image registration, to name a few.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special–Purpose and Applications–Based Systems—*signal processing systems*; I.4.0 [**Image Processing and Computer Vision**]: General
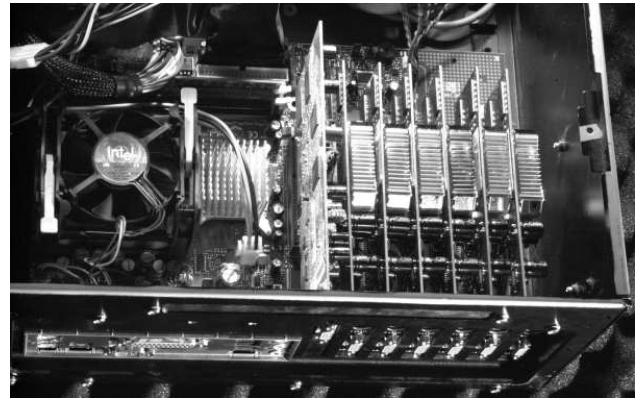
General Terms

Algorithms, Performance, Design

Keywords

Computer vision, GPU, hardware accelerated computer vision, computer graphics, computer architecture, Radon Trans-

Figure 1: **A computer vision machine with 6 PCI graphics cards, and 1 AGP graphics card. Each graphics card runs pattern recognition and computer vision tasks in parallel, creating a cheap, powerful, and easily constructible parallel architecture well suited for pattern recognition and computer vision.**

form, OpenVIDIA, chirplet transform, mediated reality

1. INTRODUCTION

OpenVIDIA is a programming framework that uses single or multiple graphics cards to operate as a general-purpose parallel computer architecture for fast computer vision and image processing. Realtime image processing and computer vision algorithms can be computationally intensive, exceeding the capabilities of the CPU. OpenVIDIA utilizes the GPU that is present on modern graphics hardware to expand the computational resources that are available to these algorithms. Furthermore, the design of the GPU architecture provides higher performance for many vision algorithms than the CPU.

One may consider computer graphics and computer vision to be, in some way, "inverses" of one another since the task of graphics is the task of image synthesis whereas image processing can be considered the task of image analysis.

The OpenVIDIA project explores the notion of using computer graphics hardware "in reverse" to accelerate the computer vision task of image analysis even though graphics hardware was initially designed for rendering images, or image synthesis. Furthermore, OpenVIDIA explores a parallel "graphics for vision" architecture created by placing multi-

ple computer graphics cards on a single motherboard. This creates a low cost, commodity, architecture for hardware accelerated computer vision and signal processing.

The GPU has been applied to other calculations beyond graphics. A survey of applications is given by Owens et al [7][1]. This paper will discuss the OpenVIDIA design and API and show some techniques OpenVIDIA uses for GPU computer vision.

2. OPENVIDIA
2.1 Design and Intent

OpenVIDIA is designed as a library that is called from within user written OpenGL programs. OpenVIDIA provides an API that implements the OpenGL calls that are needed for vision processing. Essentially, this API is implementing a set of mappings of computer vision algorithms onto graphics hardware. For instance, the OpenVIDIA API provides an interface for creating and applying image filters that use fragment shading hardware on the GPU, resulting in hardware accelerated image filtering operations. This paper will discuss both the API of OpenVIDIA as well as the methods it uses to perform some common computer vision tasks on graphics hardware.

OpenVIDIA is built with C/C++ and OpenGL. Additionally, it interfaces with libraries common to computer vision tasks. For example, it interfaces with an IEEE 1394 camera interface library, allowing video to be brought into an OpenGL vision processing framework in a simple fashion.

The API provides some function calls to run common vision processing algorithms. These can be used "as is" when they suit the application. Alternately, as the project is Open Source, the implementations can serve as template examples, which can then be altered to create similar but more specific functions as needed by users. Included in the OpenVIDIA package are image filtering operations (many vision algorithms include sequences of image filtering operations), feature detection and tracking, image registration, stereo vision, and Hough transformations to name a few.

3. APPLICATIONS AND USAGE
3.1 Image Processing and Filtering

Many computer vision operations can be considered sequences of filtering operations. Fragment programs can be considered short programs run on each pixel of an image that implement the desired filter. The architecture of the GPU allows many pixels to be processed in a parallel fashion, providing significant hardware acceleration.

OpenVIDIA provides an interface for filtering with sequences of fragment shader programs. To apply these fragment programs to input images, the input images are initialized as textures and then mapped onto quadrilaterals. These quadrilaterals are displayed in appropriately sized windows, to ensure that there is a one-to-one correspondence of image pixel to output fragment. When the textured quadrilateral is displayed, the fragment program then runs, operating identically on each pixel of the image. Filters are written in Cg [6], a simple C-like program executed at each pixel in the image.

The results of a filtering pass are saved as a texture, which can then be used as the input to a next pass. Complete

computer vision algorithms can be created by implementing sequences of these filtering operations.

3.2 A GPU Hough Transform: An example of using the full graphics pipeline for Vision

OpenVIDIA also demonstrates a mapping of a computer vision algorithm that uses the vertex processor, rasterizer, and fragment processor on the GPU. Figure 2 shows a GPU accelerated Hough line detection program. The GPU implementation of the Hough Transform is used to detect lines and curves in images, following the common formulation as: $(\theta, \rho) = x_0 \cos(\theta) + y_0 \sin(\theta)$ which takes an image point (x_0, y_0) and maps it to a set of points (θ, ρ) (for some given θ), which are the parameters of the equation of a line given by its perpendicular angle to the origin, θ and distance of its normal to the origin, ρ. The set of points $(\theta, \rho), \theta \in [0, \pi]$ describes a family of lines passing through (x_0, y_0).

First, an edge detection is performed using filtering operations on the GPU. A Canny filter implementation OpenVIDIA is discussed in [2]. The resulting edge pixels (edgels) are read back to the CPU. The image coordinates of the edge pixels are noted and then sent as an array of vertices (geometric points) to the GPU.

The GPU Hough Transform is performed on this set of input coordinates (vertices) by setting the graphics hardware projection matrix to perform a Hough Transform on this set of input coordinates for a particular angle according to equation 1. Equation 1 includes scaling and translation to utilize the full drawing area provided by the graphics pipeline. In the graphics processing pipeline, the output of vertex processing is typically the vertex position in a Normalized Device Coordinate System (NDCS). NDCS coordinates range in $(x_{ndcs}, y_{ndcs}) \in [-1, 1]$, and the sides of this area are then mapped to sides of the graphics window drawing area.

$$
\begin{pmatrix}
0 & 0 & 2\theta/\pi & -1 \\
2\cos(\theta) & 2\sin(\theta) & 0 & -1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\ y \\ 1 \\ 1
\end{pmatrix}
=
\begin{pmatrix}
\theta_s \\ \rho_s \\ 1 \\ 1
\end{pmatrix}
\tag{1}
$$

The rasterizer then quantizes the result to lie at the nearest pixel location.

Each edgel contributes a continuous sinusoidal curve in Hough space. For computation, the Hough space is discretized and these continuous curves are approximated. We do this by approximating the curve as line segments between quantized values of θ. Thus, for each input vertex, we would like to calculate its mapping at θ_i and the next increment, θ_{i+1}, and use these as the two vertex endpoints of a line to be drawn. The rasterization hardware will then draw a line which interpolates positions for $\theta \in (\theta_i, \theta_{i+1})$, creating a contiguous mapping.

This is easily achieved by creating an array of vertices on the CPU, doubling up the coordinates. Equation 1 gives us the transformed location of one end of the line segment at $\theta = \theta_i$, and we can send a similar transformation matrix for $\theta = \theta_{i+1}$. When a vertex arrives for transformation, the vertex processor determines which endpoint θ_i or θ_{i+1} the vertex corresponds to and chooses the appropriate matrix. To do this, the input vertices are sent twice each a and 1 or 0 is placed in the z coordinate to act as a "flag" field informing the vertex processor which transformation to use. This method makes use of the programmability of the vertex

---

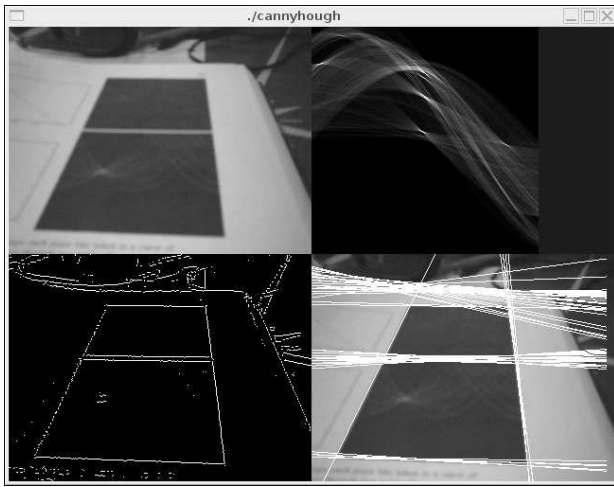[1]also see http://www.gpgpu.org for and active listing

Figure 2: Screen grab of a GPU Accelerated Radon/Hough transform that uses the vertex (geometry) processing, rasterizer (line/primitive drawing) and fragment processing (per-pixel operations) accelerated by the GPU pipeline.

processing stage to examine this flag and determine which transformation to use. While vertex processing supports branching on current hardware, it is a costly operation. It was found that a better solution was to avoid branching and implement a method whereby the vertex program calculates one endpoint and an increment to the next is calculated. Then the flag bit is used as a multiplicative mask and the increment added to the first endpoint.

It is also possible to create an equivalent Hough image by performing the transformation at each quantized θ and ρ. However, this would run the vertex program once for each point on the approximated curve. Instead, by using the line method, we only run a vertex program for each segment endpoint and the remaining points are interpolated by the rasterizer. Line segment GPU processing is also preferable to GPU point processing because it can be conducted with more parallelism due to the nature of the hardware. Our experience was that line vertices were processed roughly three times faster than and equivalent number of point vertices. The line method allows the quantization resolution to be changed by scaling the resolution of the rendering area and then issuing the exact same commands.

3.3 Realtime Chirplet Transform

The chirplet transform [5] is a signal representation that uses a family of localized chirp functions. A chirp is a signal that changes in frequency, as a function of time (or changes in time as a function of frequency), such as the sound made by a bird, bat, or slide whistle (changing the length of the resonant column while blowing into the whistle). When chirp functions are windowed (localized in time) the result is a chirplet (analogous to wavelets which are time-localized waves).

The chirplet transform is widely used in applications such as radar, medical signal processing (e.g. processing of heart sounds), analyzing bird and bat sounds, and processing EEG (brainwave) signals [1].

The one major problem that has kept the chirplet transform from being widely used in many different industries, is the computational complexity required. Although there

```
//create the feature tracking object. give it
//the size of the images it will operate on.
ft = new featureTrack(sourcewidth, sourceheight );
glGenTextures(1, &tex);        // make a texture
glBindTexture(GL_TEXTURE_RECTANGLE_NV, tex);
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_RGBA,
             sourcewidth, sourceheight, 0, GL_RGB,
             GL_UNSIGNED_BYTE,NULL );
while(1) { //rendering loop
  //update the image frame...
  glTexSubImage2D(GL_TEXTURE_RECTANGLE_NV, 0, 0,0,
                  sourcewidth, sourceheight, GL_RGB,
                  GL_UNSIGNED_BYTE, sourcedata );
  // retrieve features from the image in tex.
  Scene *s = ft->getScene(tex);
}
```

Figure 3: OpenVIDIA API performs a GPU feature detection and vector calculation. A simple API incorporates it into an OpenGL program.

exist Fast Chirplet Transforms (FCTs) based on FFTs (Fast Fourier Transforms) as well as some based on Hough's implementation of the Radon Transform, the FCT is still computationally intensive.

Within the OpenVIDIA project, there are two implementations of the Chirplet Transform (CT):

1. Direct implementation of the CT.

2. FCT by way of first computing the Wigner Ville Distribution (WVD), and then computing the Radon Transform (by way of Hough's fast implementation).

3.4 Image Registration on the GPU

We have also developed algorithms to estimate the projective coordinate transformation between successive pairs of images in a video sequence, in order to "stitch together" multiple pictures of the same subject matter. (See for example, http://wearcam.org/orbits as well as http://comparametric.sourceforge.net.) The OpenVIDIA library is particularly well-suited to this kind of computer vision and image processing task [3].

3.5 Locating and Tracking Features

OpenVIDIA provides an algorithm that performs image feature detection on the GPU. This algorithm also calculates unique appearance based feature vectors computed on the GPU. These vectors can then be re-calculated for each video image, and matched to track features in video. Figure 3 shows how to use the OpenVIDIA API calls which create the library tracker object, and match features in sequential frames of video. Feature detection in OpenVIDIA also provides an example of how to use the GPU for local histogramming and to create multiple outputs for single points.

Feature detection in OpenVIDIA begins using a Harris corner detector, implemented using the filtering pipeline technique discussed in section 3.1. The result of the feature detection stage is essentially a binary image, with feature points flagged. This is then read back to the CPU which places the locations into a 1-D array which is sent back to the GPU as a 1-D texture. The unique feature vectors are then calculated.

3.5.1 One-to-Many Outputs on the GPU

The feature vectors which uniquely identify features are an array holding 128 floating point values which identify the

feature. This requires that for a single feature point, 128 output elements be appropriately calculated on the GPU. Thus far, the filtering methods discussed produce a single output pixel per input pixel. OpenVIDIA employs a method whereby the coordinates of feature points are placed in a texture to be used as a lookup-table by a fragment shader which calculates the feature vector. A line holding 128 elements is drawn for each feature point, and the lookup table is textured onto it, along with appropriate neighbourhood offsets used in the feature computation. The feature calculation in OpenVIDIA can be used as one example of how to do such calculations where multiple outputs are needed for point inputs. This is a slight variation on the point method discussed in [2] which can be referenced for more details.

3.5.2 Histogramming in Fragment Programs

OpenVIDIA also provides an example of how to create small, local histograms using the fragment shaders on the GPU. This is of interest, since many fragment programs do not support data dependent array indexing nor pointer arithmetic, which are methods typically used to create histograms on the CPU. While an exhaustive "if else" type structure can be used to determine a given value's placement in a histogram, this does not necessarily lead to efficient fragment programs across all architectures, especially those that do not support true logical branching. A better solution is to carry out histogramming through a vector operation.

In order to accomplish this, we use the cosine function provided by fragment shaders. In our application, we are interested in histogramming into gradient directions in a small region into eight bins. To create the histogram, each gradient in question (a number lying in $[0, 7]$), is subtracted from a vector containing sequence of integer numbers $[0, 7]$. This places a '0' in the location corresponding to the observed direction and numbers of increasing absolute value on either side. Recalling that $\cos(0) = 1$, we see that taking a $\cos()$ of the values yields a $\cos()$ function centered at the desired location. Adding an offset then squaring it yields a well centered impulse around the bin location. The original gradient magnitude scalar is then multiplied by this cosine function and accumulated by vector addition to a histogram. The $\cos()$ function can be squared repeatedly to yield a more localized impulse. Furthermore, the spread is actually beneficial since it distributes gradient directions which lie between quantized bins, reducing quantization noise.

3.6 Mediated Reality

Part of our research includes implementing the Open-VIDIA project on a battery-powered body-borne computer connected to an EyeTap (http://eyetap.org) device, that basically causes the eye itself to, in effect, simultaneously function as both a camera and a display. In particular we designed and built a wearable OpenVIDIA computer system and EyeTap system around an nVIDIA 5200 system. Once we have scene understanding, camera egomotion, and the like, implemented in OpenVIDIA, we can begin to construct a real-time system for computer-mediated reality.

Mediated reality can be used for *augmented reality* and *virtual reality* as well as some new capabilities, such as implementing a visual filter to assist the visually impaired (*diminished reality* through scene simplification).

4. HISTORICAL CONTEXT: OUR FIRST APPLICATION

Originally, we had a hexagonal column shower that we wished to automate for water savings, so we outfitted it with a six-channel computer vision system to automate the control of the six shower nozzles around the column. The idea was to construct a three dimensional volume of each of the six or fewer users, and have the sensor operated nozzles only come on when the frustum of the cone of water intersected entirely with the body of a user. The goal was to not waste even a single drop of water. We also implemented a fuzzy data bus, using analog video input/output capabilities of the six graphics cards, to pass approximate quantities as image arrays between GPUs or graphics cards.

5. CONCLUSION

OpenVIDIA provides a library and API for using single or multiple graphics processing units to accelerate computer vision and image processing. Full source code and documentation on OpenVIDIA are available through the project web page at http://openvidia.sourceforge.net.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] J. Cui, W. Wong, and S. Mann. Time-frequency analysis of visual evoked potentials using chirplet transform. *IEEE Electronics Letters*, 41(4):217–218, 2005.

[2] J. Fung. Chapter 40: Computer vision on the gpu. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, pages 649–665. Addison-Wesley, 2005.

[3] J. Fung and S. Mann. Computer vision signal processing on graphics processing units. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004)*, pages 83–89, Montreal, Quebec, Canada, May 17–21 2004.

[4] E. Lindholm, M. J. Kilgard, and H. Moreton. A user–programmable vertex engine. In *Computer Graphics, Proc. of SIGGRAPH 2001*, pages 149–158, 2001.

[5] S. Mann and S. Haykin. The chirplet transform: A generalization of Gabor's logon transform. *Vision Interface '91*, pages 205–212, June 3-7 1991. ISSN 0843-803X.

[6] 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*, volume 22, July 2003.

[7] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005.