Image Processing on the GPU: Implementing the Canny Edge Detection Algorithm

Yuko Roodt
Highquest, Johannesburg
yuko@highquest.co.za

Willem Visser
University of Johannesburg
glasoog@gmail.com

Willem A. Clarke, Member IEEE
University of Johannesburg
willemc@uj.ac.za

Abstract

In this paper we present a detailed Graphics Processing Unit (GPU)-based implementation of the well known Canny edge detection algorithm. The aim of the paper is to provide an overview on our approach to implement the Canny edge detection algorithm, as it encompasses a set of image processing techniques. The result is an algorithm that can be applied in real-time applications. A basic understanding of the hardware architecture and available tools are required to successfully map computer vision and image processing techniques to the GPU. We describe some of the benefits associated with this platform while looking at possible development pitfalls, solutions and performance results.

1. Introduction

Processing images require a substantial amount of computational resources due to the ever increasing size of images. Therefore, image processing tasks often consume a large number of processing time on the Central Processing Unit (CPU), especially if multiple operations have to be performed on an image. This problem becomes even worse when the input is not just a single image, but a video stream.

Until recently image processing tasks were generally done on the CPU, but when GPUs with 32 bit floating point arithmetic and programmable fragment and vertex shaders were introduced in 2003 [1], it became possible to do image processing on the GPU. Suddenly higher computational power became widely available at a lower cost [2]. This is shown in **Table I**.

Table I: *Computational power comparison between CPU and GPU [2]*

	3.0 GHz Intel Core2 Duo	NVIDIA GeForce 8800 GTX
Computation	48 GFLOPS	330 GFLOPS
Memory bandwidth	21 GB/s	55.2 GB/s
Price (2007)	$874	$599

It is expected that this advantage will not diminish soon. The average annual GFLOPS growth of GPUs is 1.7× (pixel shaders) to 2.3× (vertex shaders) compared to the 1.4× annual growth of CPUs [2]. The growth from 2003 to 2007 can be seen in Fig. 1.

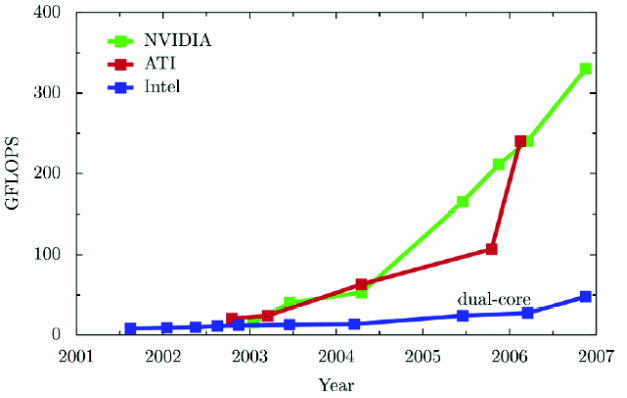

Figure 1: *Computational power growth [2]*

It is clear that there may be an advantage to adapt image processing algorithms, (be it existing or new) for the GPU. In this paper we provide an introduction to image processing on the GPU by implementing the Canny algorithm step-by-step. As far as we are aware, this is the first published implementation of this algorithm.

GPUs were originally developed for games processing, with mostly visual rendering as the main objective and within the ambit of games programmers. However, it did not take long for researchers to discover that the GPU as stream processing unit could be applied to more than just gaming. General purpose computing on GPUs (GPGPU) has found its way into fields as diverse as oil exploration, scientific computing [14], image processing [23,13], medical and biological processing, audio and signal processing, database processing, and even stock options pricing determination [10]. The web site referenced in [10] provides a good starting point for interested readers, as well as the book references in [5-7].

A caveat in the general acceptance of GPU's as general purpose stream processors have been the different skills set that is required of programmers, and the fact that to fully realise the processing speed benefits, existing algorithms need to be adapted for a massive parallel streaming architecture. However, as the architectures of both NVidia and ATI have opened up, more and more applications have appeared in the press, the latest commercial application being a virus detection programme [11], with reported speed increases of 21 times compare to a top-of-the range CPU. Further proof that GPGPU will become more prevalent, is the fact that NVidia has launched the Tesla range specifically for GPGPU with more than 1.8 TFLOPS of raw processing power [12].

Several papers have appeared that consider the GPU as a general purpose processor. In [22], a general application overview is provided, while a good architecture and application overview is provided in [1]. In most reported applications, processing speed increases up to 100 times are possible, provided the application is suitable to the architecture. Examples of specific algorithm implementations include Genetic Programming [15], Frequency and histogram estimations (FFT) [17, 18, 19], Linear algebra operators [20], Dense Matrix Algebra [21] and Shortest path problems [24]. This is by no means a exhaustive list as a casual glance at the SIGGRAPH proceedings will testify. An interesting application in the telecommunications domain is that presented in [16] to accelerate high-fidelity network simulations.

In the next section we will provide a high level discussion of the GPU architecture. Thereafter, we provide a summarised overview of the original Canny edge detection algorithm. A detailed discussion of our GPU Canny algorithm implementation will follow next. We present speed and operational results for this implementation. The paper will finish with concluding remarks.

2. GPU Architecture

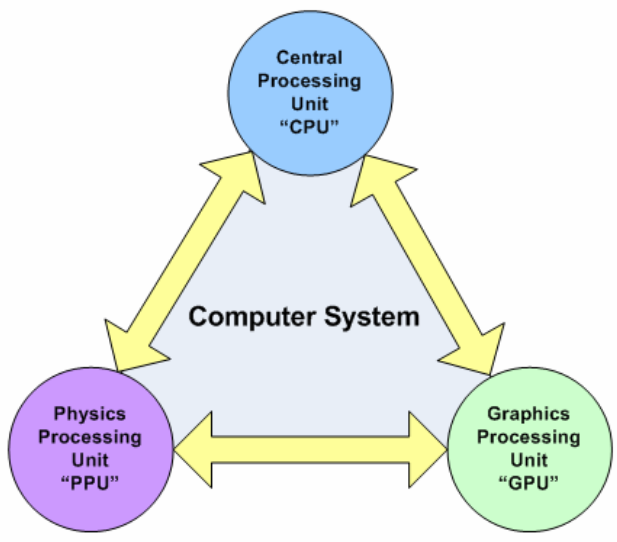

Figure 2: *Balancing processing Power*

The graphics processing unit (GPU) or dedicated graphics microprocessor is an essential integrated part of the computer system; its purpose is to lighten the workload of the central processing unit (CPU) and the physics processing unit (PPU) where applicable since the PPU is only an add-on processor responsible for all physics calculations performed in Computer games and scientific applications.

Fig. 2 depicts the interconnectivity of the available processing resources of a general computer system. In real-time and scientific applications this communication and sharing of workload is of utmost importance to provide maximum throughput and execution of complex problems. Balancing and distributing the processing work over all available hardware resources promotes the efficient use of hardware.

One important programming design consideration in the above architecture is the moving around of large amounts of data between the dedicated GPU memory and CPU memory. Sometime one may accept slower execution speed for a specific part of an algorithm rather then incurring the cost of moving a large image between different memories.

The Graphics Processing Unit (GPU) is a dedicated processing device that allows for fast parallel execution of complex high-end rendering operations and algorithms. In the past, these powerful processors were dedicated solely for graphical rendering operations, but since then the introduction of programmable pipelines provided the ability to override the fixed functionality of the hardware. A new application concept has arisen which turns the massive floating-point computational power of modern graphics accelerators' shader pipelines into a general purpose graphics processing unit.

A parallel processing pipeline is well suited for the rendering process, because it allows the GPU to function as a stream processor. This is because all vertices, geometries and fragments can be thought of as independent of each other. This allows all stages of the pipeline to be used simultaneously for different vertices or fragments as they work their way through the pipeline. Multiple vertices, fragments and operations can be processed at the same time.

GPU hardware can be integrated into the motherboard but the most powerful class of GPUs typically interface with the motherboard by means of expansion slots such as PCI Express (PCIe) or an Accelerated Graphics Port (AGP). The GPU can usually be replaced or upgraded with relative ease, assuming the motherboard is capable of supporting the upgrade.

One does not have to be limited to the use of only one graphics card, because using technologies such as NVIDIA's Scalable Link Interface (SLI) or ATI's competing Crossfire Technology, multiple cards can be configure to run together. This in turns allows for increased processing power and the ability to divide the work balance. In Fig. 3 the architectural diagram of a typical GPU is shown.

The stream processor of the GPU consists of three concurrent processing phases, i.e. the vertex shader, geometry shader and fragment shader. Each of these can be programmed individually by custom short programs called "*shader programs*" which can include (except for the input image texture) additional image textures and attributes as input.

The *vertex processor* is responsible for computing vertex shader programs. It replaces part of the geometry stage of the graphics pipeline by providing the ability to transform and manipulate individual incoming vertices [3] or polygon corners.

The *geometry processor* differs from other well known shader processors which replace well known fixed pipeline tasks. This processor processes complete geometric primitives that form the basis structure of an object or model in your virtual environment.

The *fragment processor* discards, declares or determines what colour a pixel fragment should be by processing the corresponding fragment program for each rendering pixel [4]. A fragment is considered to be all data needed in the process of generating a pixel in the frame buffer.

The new generation of GPUs consist of a number of processors which are multipurpose processors in nature and can

run the same set of shader program pairs on multiple vertices, primitives and fragments simultaneously.

During the rendering phase of applications, the image processing tasks starts out as models or geometric shapes. These shapes can be broken down into vertices, facets and their connectivity information. This data is used to reconstruct or render the virtual world. Firstly, vertices are processed, translated and transformed by the vertex processor, which in turn allows the geometric processor to statically or dynamically link related vertices. Finally the fragment processor takes over and declares each pixel fragment's colour information. After this the information is passed to a frame buffer.

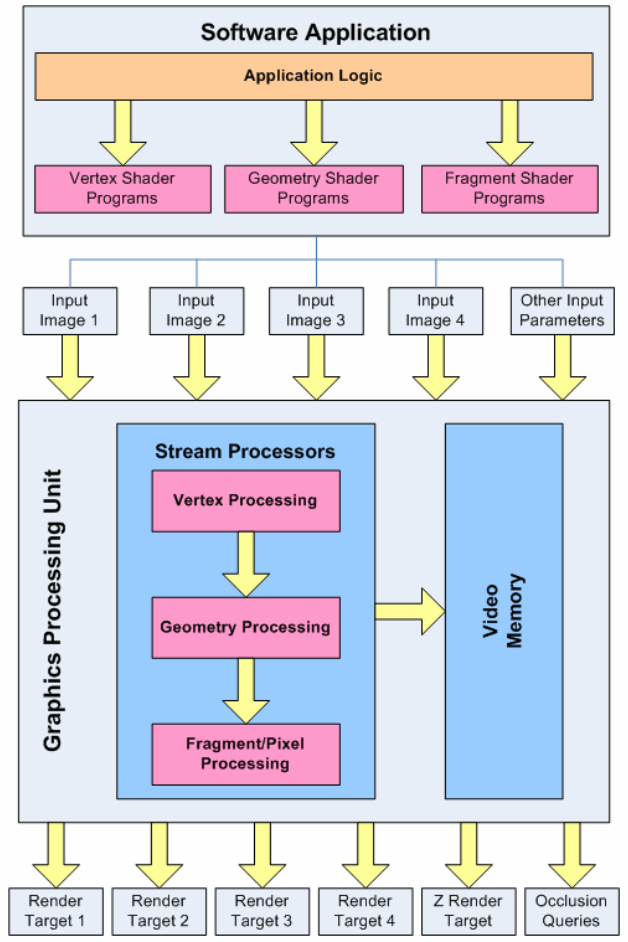

Figure 3: *Architectural overview of the GPU*

The instruction set of a graphics processing unit consists of special mathematical operations usually used in the high-end rendering process, but which is still applicable to other scientific fields. To harness the stream processing power of the GPU, one must first break down and map complex real world problems to the rules and constraints of the GPU architecture.

Some problems and algorithms map better to the GPU argitecture, two key attributes of computer graphics computations are parallelism and independence. Streams of vertices and pixels are processed simultaneously while the computations on elements have little or no dependence on each other. GPU data communication promotes the "gathering" of data rather then the "scattering" of information since "gathering" only needs the ability to randomly read from data stores or textures while "scattering" requires the ability to write to random data locations which is currently not possible on the current GPU architecture [6].

The GPU shader pipelines can be programmed using three different high level shading languages eg. GLSlang (GLSL), C for Graphics (CG) and High level shading language (HLSL). GLslang was developed by the OpenGL ARB to provide engineers and developers more control and flexibility over the graphics pipeline without resorting to hardware specific languages or assembly languages. GLSL shaders do not compile to standalone application but are code strings that are passed to the hardware for interpretation and execution. These small hardware programs are not platform specific and can run on a various number of platforms such as cell phones, gaming consoles and personal computers that support the GLSL API framework [8]

Results and output can be obtained from the GPU by rendering to a single image texture, screen, depth target and by doing an occlusion query.

3. Canny Edge Detection Algorithm

The canny edge detection algorithm was developed by John F. Canny [9] as a means to detect edge lines and gradients for the purpose of image processing. This algorithm provides good detection and localization of real edges while providing minimal response in low noise environments. This algorithm is well-known and explained in any introductory text on image processing. The main stages of the Canny Algorithm are as follows:

- Noise reduction by filtering with a Gaussian blurring filter;
- Determining the gradients of an image to highlight regions with high spatial derivatives;
- Relate the edge gradients to directions that can be traced;
- Tracing valid edges; and
- Hysteresis thresholding to eliminate breaking up of edge contours.

4. Implementation

A SHADER PROCESS EXAMPLE

(a)

```
//CPU Example - RGB to Black & White
vec3 current_Color;
float luminance;
for (int h=0;h<image_Height;h++) {
   for (int w=0;w<image_Width;w++) {
      current_Color=image_Array[h][w];
      luminance=dot(vec3(0.299,0.587,0.114),current_Color);
      if (luminance>threshold)
         image_Array[h][w]=vec4(1.0);
      else
         image_Array[h][w]=vec4(0.0);
   }
}
```

(b)

```
//Vertex Shader Example - RGB to Black & White
void main() {
   gl_TexCoord[0]=gl_MultiTexCoord0;
   gl_Position=ftransform();
}
```

(c)

```
//Fragment Shader Example - RGB to Black & White
uniform sampler2D input_Image;
uniform float threshold;

void main() {
   vec3 current_Color=texture2D(input_Image,gl_TexCoord[0].xy).xyz;
   float luminance=dot(vec3(0.299,0.587,0.114),current_Color);
   if (luminance>threshold)
      gl_FragColor=vec4(1.0);
   else
      gl_FragColor=vec4(0.0);
}
```

(d)

Figure 4: *(a) Black and white filter process overview; (b) CPU implementation; (c) GPU vertex shader example; (d) GPU fragment (pixel) shader example*

In Fig. 4 (a) is shown the process of converting a color image to its black and white presentation. An input image and the threshold are passed to the GPU. These input parameters will then be used by the Vertex and Fragment shaders to process the resulting black and white image. The end results are obtained by rendering to a texture of the same dimensions.

A CPU implementations is shown in Fig 4 (b) of this filter process. Comparing the CPU implementation with the corresponding GPU implementation in Fig. 4 (d), one finds that the GPU version does not have any loops to process every pixel in the image. Each fragment programme is automatically executed for every pixel.

To process a pixel coordinate, it needs to be enabled in the vertex shader programme. The vertex fragment program in Fig. 4 (c) enables the 1st texture coordinate channel and transforms each individual vertex that was queued for rendering to the .

Fragment or pixel processing takes place in Fig 4 (d). The current pixel's color is extracted from the input image of Lena and then converted to a luminance value. This value is compared to the threshold and the appropriate output fragment is chosen and rendered.

CANNY SETUP

In Fig. 6 the Canny edge detection algorithm is broken down into smaller processing steps, a daisy chain approach is followed where the output of one step is passed to another step for processing. This allows complex problems to be solved by only processing simple individual steps to reach the end results.

```
//Vertex Shader - Default
void main() {
   gl_TexCoord[0]=gl_MultiTexCoord0; //Enable Texture Channel
   gl_Position=ftransform();         //Manualy Convert Position
}
```

Figure 5: *Default vertex shader program*

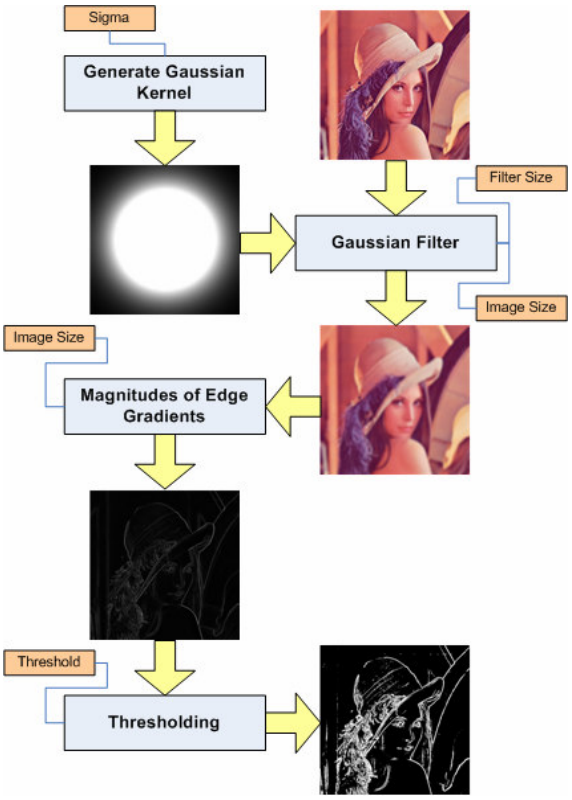

Figure 6: GPU implementation of the Canny edge detection algorithm

The small vertex program in Fig. 5 was executed in conjunction with the preceding fragment shader programs. This program is only responsible for the activation of the first texture coordinate channel and the transformation of the vertex to clip space where vertices that are not visible or out of view from the view frustum are clipped away to reduce the processing of invalid vertices, its function is similar to the fixed vertex functionality of the OpenGL API. This vertex shader must be executed for every fragment shader to follow without any change.

GENERATE GAUSSIAN KERNEL

```
//Fragment Shader - Generate Gaussian Kernal
uniform float sigma;
const float PI=3.1415;
const float center_Pos=0.5;
const float s=2.0*pow(sigma,2.0);
const float q=1.0/(PI*s);

void main() {
   gl_FragColor=vec4(q*exp(-(pow(gl_TexCoord[0].x-center_Pos,2.0)+
                     pow(gl_TexCoord[0].y-center_Pos,2.0))/s));
}
```

Figure 7: *Fragment shader program that generates a Gaussian kernel*

The Gaussian kernel generation shader in Fig. 7 uses the input sigma attribute to calculate the corresponding Gaussian kernel using the equation

$$x \mapsto \frac{1}{\sqrt{2 \cdot \pi} \cdot \sigma} \cdot e^{-\frac{x^2}{2 \cdot \sigma^2}} \qquad (1)$$

and store it in an output texture. Where this would normally be a

5x5 or 7x7 kernel window, it is now stored to a texture of a pre-selected size. This will be used as the Gaussian kernel in the next step.

This texture can be changed according to the accuracy required for the filter kernel. A large texture is not necessarily required, since automatic interpolation occurs for filter coefficients requested between neighbouring pixels.

The OpenGL command *gl_TexCoord* returns the current pixel coordinate. The resulting pixel colour in the output image is stored with the command *gl_FragColor.*

FILTER WITH GAUSSIAN KERNEL

Again, the following fragment shader programme is executed for each pixel in the input image (*input_Image*) using the filter kernel (*filter_Image*).

```
//Fragment Shader - Filter with Gaussian Kernal
uniform sampler2D input_Image;
uniform ivec2 image_Size;
uniform sampler2D filter_Image;
uniform int filter_Size;

const vec2  tex_Offset=1.0/vec2(image_Size);
const int   halfFilter_Size=filter_Size/2;
const float filter_Offset=1.0/float(filter_Size);
const float center_Pos=0.5;

void main() {
  vec3 current_Color,total_Color=vec3(0.0);
  float filter_Coefs,filter_Total=0.0;
  for (int w=-halfFilter_Size;w<=halfFilter_Size;w++) {
    for (int h=-halfFilter_Size;h<=halfFilter_Size;h++) {
      //Exstract filter Coefs
      filter_Coefs=texture2D(filter_Image,center_Pos+
                   vec2(float(w),float(h))*filter_Offset).x;
      filter_Total+=filter_Coefs;
      //Exstract Color
      current_Color=texture2D(input_Image,gl_TexCoord[0].xy+
                    vec2(float(w),float(h))*tex_Offset).xyz;
      total_Color+=current_Color*filter_Coefs;
    }
  }
  total_Color/=filter_Total;
  gl_FragColor=vec4(total_Color,1.0);
}
```

Figure 8: *Fragment shader program that filters an image using a filter kernel*

An input image has coordinates of 0 to 1 in the *x* and *y* directions with all coordinate references internally represented as single precision floating point values. Therefore, a value (*tex_Offset*) is required to represent the width (and height) of a single pixel. The size of the input image is required as an input parameter to derive this pixel size value.

The OpenGL command *texture2D* retrieves the pixel colour at a coordinate.

DETERMINING EDGE GRADIENT MAGNITUDE

```
//Fragment Shader - Determin Magnitude of Edge Gradient
uniform sampler2D input_Image;
uniform ivec2 image_Size;
const vec2 tex_Offset=1.0/vec2(image_Size);

void main() {
  vec2 pixelRight_Coord =gl_TexCoord[0].xy+vec2(tex_Offset.x,0.0),
       pixelLeft_Coord  =gl_TexCoord[0].xy+vec2(-tex_Offset.x,0.0),
       pixelTop_Coord   =gl_TexCoord[0].xy+vec2(0.0,tex_Offset.y),
       pixelBottom_Coord=gl_TexCoord[0].xy+vec2(0.0,-tex_Offset.y);
  vec2 gradient=vec2(length(texture2D(input_Image,pixelRight_Coord).xyz
                    -texture2D(input_Image,pixelLeft_Coord).xyz),
                    length(texture2D(input_Image,pixelTop_Coord).xyz
                    -texture2D(input_Image,pixelBottom_Coord).xyz));
  gl_FragColor=vec4(length(gradient));
}
```

Figure 9: *Fragment shader program that determines the magnitude of the largest edge gradient*

The edge gradients are calculated in Fig. 9, the magnitude of the gradient is considered to be the edge response and the algorithm was slightly modified to allow the detection of edges on colour images without converting to a greyscale image first [9].

THRESHOLDING

The final stage of the edge detection algorithm is the thresholding of the edge magnitudes; this can be accomplished in several ways depending on the application's needs. In this implementation, only a simple black and white thresholding schema was used to discard small edge magnitudes while maintaining and enhancing selected edges.

5. Results and Discussion

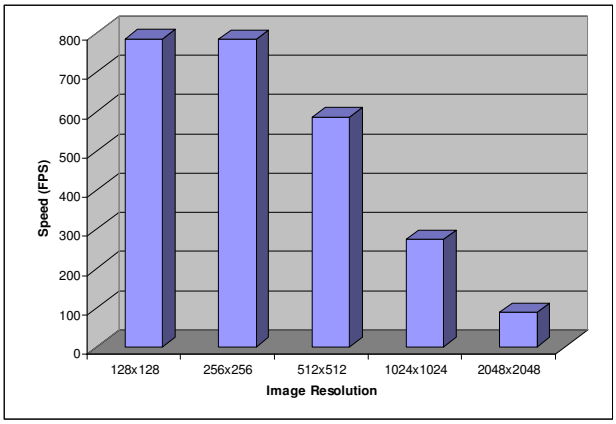

Figure 10: *Speed of Canny algorithm on GPU*

Fig. 10 depicts the average speed in frames per second (FPS) for this algorithm on a NVIDIA Geforce 8800 GTX graphics processing unit. Even on the highest tested resolution the results were still enough to allows the Canny edge detection algorithm to be computed approximately 80 times per second. This allows for real-time processing of high definition video applications.

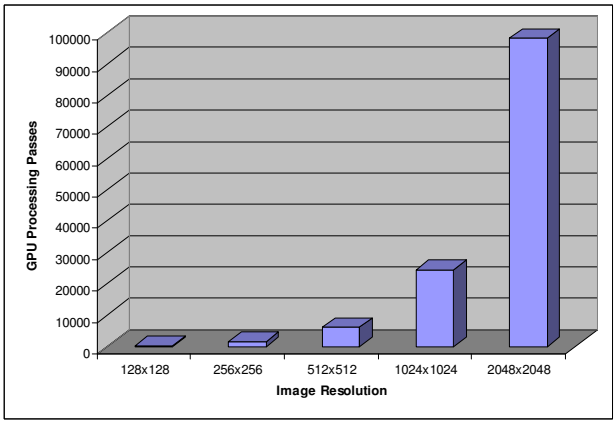

Figure 11: *GPU Processing passes for Canny algorithm on NVIDIA 8800 GTX*

The GPU hardware we used for testing purposes had 128 stream processors available which it used to simultaneously compute 128 individual pixels at a time for each processing pass. Fig. 11 emphasises the dramatic increase of processing complexity due to the enlarging of image resolutions.

The processing complexity was calculated by the mathematical function depicted in the following formula:

$$\text{processing\_complexity} = \frac{\text{image\_width} * \text{image\_height} * \text{number\_passes}}{\text{number\_processors}} \quad (2)$$

The results demonstrate that the GPU is a liable option for the processing of hardware intensive operations and algorithms. We do not discourage the use of the CPU as the main processing unit, but rather encourage the use of this incredible processor as a helping hand in the conquest for real-time image processing and scientific calculations.

Segmentation of processing problems and algorithm still pose a big threat, since proper understanding of the imbedded GPU architecture is required to efficiently take full advantage of the available hardware. It is clear that certain classes of computer vision and image processing techniques are well suited to this parallel processing architecture.

6. Conclusion

In this paper we present a detailed Graphics Processing Unit (GPU)-based implementation of the well known Canny edge detection algorithm. We provided code snippets of our implementation together with explanations of important constructs and issues. We believe that the lack of implementation detail associated with GPU development is a shortcoming in the existing body of knowledge which hampers further development. We presented graphs depicting the execution speed achieved by our implementation, which shows that the complex Canny algorithm may be used in real-time video applications.

References

[1] Dokken T, Hagen TR. and Hjelmervik JM, "The GPU as a high performance computational resource", *Proceedings of SCCG'05*, May 12-15, 2005, Budmerice, Slovakia.

[2] Houston M, "General-purpose computation on graphics hardware", *Proceedings of SIGGRAPH 2007*

[3] "ARB vertex shader specification", located at http://oss.sgi.com/projects/oglsample/registry/ARB/vertex_shader.txt

[4] "ARB fragment shader", located at http://oss.sgi.com/projects/oglsample/registry/ARB/fragment_shader.txt

[5] Fernando R (*Editor*), *GPU Gems – Programming Techniques, Tips, and Tricks for Real-time Graphics*, Addison-Wesley, 2004

[6] Harris M (*Editor*), "Mapping Computational Concepts to GPUs", *GPU Gems 2*, 2005

[7] Nguyen H (Editor), *GPU Gems 3*, Addison-Wesley, 2008.

[8] Rost, Randi J, *OpenGL Shading Language*, 2nd Edition, Addison-Wesley, 2005.

[9] Canny J, "A Computational Approach to Edge Detection", *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol.8, 1986, pp.679-714.

[10] GPGPU Forum, access home page at http://www.gpgpu.org

[11] The Inquirer, 12th September 2007, accessed at http://www.theinquirer.net/gb/inquirer/news/2007/09/12/gpgpu-drastically-accelerates-anti-virus-software

[12] Smith T, "Nvidia GPGPU line sparks into life with Tesla", The Register, 21 June 2006, accessed at http://www.reghardware.co.uk/2007/06/21/nvidia_launches_tesla/print.html.

[13] Sugita K, Naemura T, Harashima H, "Performance evaluation of programmable graphics hardware for image filtering and stereo matching", *Proceedings of the VRST'03*, 2003

[14] Govindaraju NK, Raghuvanshi N, Manocha D, "Fast and approximate stream mining of quantiles and frequency using graphics processors", *Proceedings of SIGMOD'05*, June 14-16, 2005, USA, pp.611-621.

[15] Chitty DM, "A data parallel approach to genetic programming using programmable graphics hardware", *Proceedings of GECCO'07*, July 7-11, 2007, London UK, pp.1566-1572.

[16] Xu Z, Bagrodia R, "GPU-accelerated evaluation platform for high fidelity network modelling", *Proceedings of the 21st Int. Workshop on PADS'07*, 2007.

[17] Govindaraju NK, Larsen S, Gray J, Manocha D, "A memory model for scientific algorithms on graphics processors", *Proceedings of SC2006*, November 2006, Tampa, Florida, USA, 2006.

[18] Moreland K, Angel E, "The FFT on a GPU", *Graphics Hardware 2003*, pp.112 – 136.

[19] Scheuermann T, Hensley J, "Efficient histogram generation using scattering on GPUs", *Proceedings of I3D 2007*, Seattle, Washington, April 30-May 02, 2007, pp.33-37.

[20] Kruger J, Westermann R, "Linear algebra operators for GPU implementation of numerical algorithms", ACM Transactions on Graphics, vol.22, no.2, July 2003, pp.908-916.

[21] Moravanszky A, "Dense matrix algebra on the GPU", 2003. accessed at http://www.shaderx2.com/shaderx.PDF.

[22] Manocha D, "General-purpose computations using graphics processors", IEEE Computer, August 2005, pp.85-88.

[23] Fung J, Mann S, Aimone C, "OpenVidia: Parallel GPU Computer Vision", *Proceedings of MM'05*, November 6-11, Singapore, pp.849 – 852.

[24] Micikevicius P, "General Parallel Computation on Commodity Graphics Hardware: Case Study with the All-Pairs Shortest Paths Problem", *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2004, pp. 1359-1365.