GPUCV: A FRAMEWORK FOR IMAGE PROCESSING ACCELERATION WITH GRAPHICS PROCESSORS

Jean-Philippe Farrugia(+)(*), Patrick Horain(+), Erwan Guehenneux(+), Yannick Alusse(+)

(+) GET/INT, Département EPH, 9 avenue Charles Fourier, 91011 Evry Cedex, France

(*) LIRIS, Université Lyon 1, 42 boulevard du 11 novembre, 69622 Villeurbanne Cedex, France

jean-philippe.farrugia@iuta.univ-lyon1.fr, Patrick.Horain@int-evry.fr

ABSTRACT

This paper presents a state of the art report on using graphics hardware for image processing and computer vision. Then we describe GPUCV, an open library for easily developing GPU accelerated image processing and analysis operators and applications.

1. INTRODUCTION

Nowadays, graphics processing units (GPU) have made their way to home computers through video games and multimedia. Since these highly specialized processors offer high computing power, it is interesting to use this readily available resource for general purpose computing, especially for image processing and analysis.

In this paper, we show that, while porting to GPU algorithms that process all pixels independently is be fairly straightforward, global image computation can still be achieved with some *ad hoc* implementation. We also propose a simple framework to harness the GPU power while minimizing development effort.

In the next section, we introduce GPU programming basics. Then we review previous work in this field, and we present our contribution in the fourth and fifth section. The sixth section discusses integration issues presents an Open CV port with our library. The final section discusses perspectives.

2. IMAGE PROCESSING AND COMPUTER VISION ON GPU : BASICS

2.1. Computing with GPUs

The purpose of modern graphics processors is to generate high quality images from three-dimensional models. Their processing pipeline consists of the following steps :

- world to screen projection;
- lighting computation;
- primitives assembly;
- rasterization: transforming geometric data into image fragments;
- colour computing: fragments are turned into pixel values.

GPUs host all these computations in a single chip and leave the central processing unit (CPU) available for other tasks. They offer great abilities for algebraic operations and native handling of floating point numbers, vectors and matrices. Their host graphics cards hold dedicated fast access memory for storing texture and data. General purpose computing with GPU is possible at the cost of reformulating common algorithms to fit the processing pipeline, which may or may not be easy. In 2002, the third generation of GPUs introduced programmable pipelines, so turning GPUs into stream processors.

2.2. GPU programming

Two steps in the processing pipeline are programable:

- the world to screen projection and lighting computation, which are handled by the vertex processor or vertex shader;
- the colour computation, which is handled by the fragment processor or fragment shader.

Both shaders have limitations.

- They can only access and modify their own data and have a fixed format output.
- Parameters are passed to shaders in read-only mode.
- Their number of assembly instruction is limited to 1024 or 65536;

Common high level languages for GPU programming include Cg (C for Graphics, nVidia) [1], GLSL (Open GL Shading Language, 3dLabs) [2] and HLSL (High Level Shading Language, Microsoft and nVidia) [3]. While these languages impose to compile and link separately the shader and CPU programs, the metaprogramming approach used by Sh [4] and Brook GPU [5] allows to hide the difference between CPU and GPU. Although, these languages are not image specialized and remain hard to use for image processing and computer vision scientists.

3. PREVIOUS WORK

Since high level languages are available for GPU programming, algorithm transposition became less difficult. The graphics processor may be viewed as a "SIMD" (single instruction, multiple data) stream processor, where the fragment processor is the kernel computation applied to all the data. This allows to easily transpose parallel algorithms. In this section, we shall describe previous work based on this approach.

The OpenVidia library [6] developed by Fung *et al.* offers a collection of fragment shaders dedicated to computer vision and image processing and a framework to easily apply them on an image or a video.

In [7], Moreland and Angel used a fragment shader to compute a fast Fourier transform on GPU. Here, the main difficulty is to implement the transform as several steps where pixel are processed independently. The fast Fourier transform on GPU is four times faster than on CPU.

Strzodka and Telea presented a generalized bi-dimensional distance transform in [8]: each pixel received, in addition of its distance, a label corresponding to the closest object to this pixel. This labelling is obtained via a fragment shader which alters the depth buffer. An adaptive version is also presented. Performances are impressive and this method is up to 34 times depending on the complexity of the image.

Strzodka and Galbe [9] introduced a motion estimation method based on eigen vectors analysis in a spatio-temporal tensor. The sequence is transmitted frame by frame in video memory and eigen vectors, tensor and visualization are computed with fragment shaders. This speeds up the process up to 2.8 times with reference to a Pentium 4 processor.

4. A GLOBAL APPROACH OF GPUS

Fragment shaders may speed up image processing and computer vision algorithms but they are subject to a serious limitation: algorithms that can be implemented must be parallel at the pixel level. It is not possible to access the value of nearby pixels that have just been computed for neighborhood recursive processing.

Therefore, some image processing operations require further GPU resources. Approaches to GPU processing purely based on Opengl rather than fragment shaders have been proposed by Cabral *et al.* [10] for tomographic reconstruction, Rumpf and Strzodka for image segmentation [11] and non linear diffusion filtering [12]. Hybrid methods, mixing fragment shaders with geometry methods were proposed by Kelly and Kokaram [13] for motion estimation and Hoff [14] for distance transform. The methods used in these works may not be reproduced with the fragment shader kernel / pixel flow method. Therefore we think there is a need for a global library in which these algorithms may be implemented.

5. A NEW PROGRAMMING INTERFACE FOR IMAGE PROCESSING WITH GPU

We developed GPUCV, an open library [1] meant for easy development of GPU accelerated image processing and computer vision applications.

First, it allows easy shaders handling. A "Filter Manager" class provides for shaders creation and application on images. Since OpenGL and the GLSL shading language are used for shaders, any system supporting OpenGL 2.0 will support our interface. It supports not only the fragment shader kernel scheme, but also using vertex shaders, which are rarely used for image processing and vision algorithms on GPU (unlike the fragment shaders, they allow to modify the coordinates transmitted through the graphics pipeline). Shaders can be dynamically generated as meta-shaders which allows flexibility in setting up possibly many parameters. Last, our programming interface proposes to use any geometry for shader computing to allow geometric methods. Other libraries generally impose a screen aligned quad mapped with the processed image. All these characteristics are unique to our library and cannot be exploited with OpenVidia.

Second, GPUCV is an extension of Intel's Open CV [15] library that is fairly popular for developing interactive computer vision applications. It is meant to support GPU acceleration for applications written with Open CV without modifying the application source code. The current version of GPUCV includes colour conversion, histogram and morphological operations. In this section, we detail the methods and difficulties of porting operators and present some results. Since image processing nor computer vision scientist are usually not familiar with (painful) graphics programming and hacking, this is an advantage on GPGPU languages like Sh or Brook GPU.

5.1. Issues for GPU integration into Open CV

We designed an interface meant to hide complexity of handling fragment and vertex shaders. GPUCV operators share Open CV data structure, have similar interface with their Open CV counterpart and can seamlessly be mixed with native Open CV operators.

[1] Available for download from *http://picolibre.int-evry.fr/projects/gpucv*.

This raises some difficulties that are discussed hereafter. First, shaders can only process images in video memory while Open CV's IplImage are stored in central memory, so we need an transparent and efficient transfer mechanism. Second, a GPU version of an Open CV operator may not be faster than the original CPU version, so heuristics for smart switching between the Open CV and GPU versions. Finally, OpenGL should not be visible to the user.

5.1.1. Image handling

Transferring data between the video and central memories takes a lot of time, so we need efficient image synchronization. The image object manager is our interface between Open CV's IplImage structure in central memory and OpenGL textures in video memory. It holds a flag that controls whether result image data should be left in video memory or pulled back in central memory, which is the default behaviour for compatibility with native Open CV operators.

5.1.2. Switching mechanism

An Open CV operator may need more than one shader to be ported. For example, depending on the parameters, a shader may be more adapted than another. The Open CV version may even be faster than the GPU one, so we need a switching mechanism to choose the best option depending on the parameters values.

5.1.3. GL layer hiding

GPUCV initialization allows to choose between automatic or manual OpenGL configuration. The user may create and use his own OpenGL context, or let the library do it for him.

5.2. Results

5.2.1. Brute force tests

We compared the execution time of native Open CV operators with their GPUCV counterpart. GPUCV is up to 18 times faster than native Open CV, as shown in table 1. Theses tests were realized on a computer with a Pentium 4 CPU, 1 Gigabyte of RAM and a GeForce 7800 GTX GPU. Full benchmarks with various GPUs and image sizes can be found on the GPUCV web site.

	Open CV	GPU
3x3 erosion	24 ms	3.0 ms
5x5 erosion	31 ms	8.5 ms
RGB to XYZ	11 ms	0.6 ms
RGB to HSV	18 ms	1.7 ms
Binary threshold	1.2 ms	0.7 ms

Table 1. Example performance comparison between native Open CV and GPUCV for a 1024x1024 image.

5.2.2. Histogram computing

As a further example of what our library can do, we propose a novel method to compute histograms on GPU with vertex shaders. Our idea is to associate vertice with pixels and to move them to some position depending on their associated pixel value and then to count those vertices in the frame buffer, which can be done with 16 bits floating point blending available with new graphic cards. The process consists in two steps: the first one classifies vertice associated to pixels relatively to their value, so creating columns of vertice with the same luminance, and the second pass sums the columns to compute the histogram. For a histogram with N classes, the result is a line of N pixels in the framebuffer, where the value of the i^{th} pixel equals the number of pixels in class i of the source image. Computing the histogram of a 512x512 image with this technique requires approximately 5 milliseconds on a Geforce 7800 GTX GPU, which is 4 times faster than the state of the art method using occlusion query with fragment shaders. In spite this remains much slower than optimized software on CPU such as the 10 times faster Intel's Open CV library [15], this may still be valuable if histogram computing occurs inside a pipeline of GPU operations because this saves data transfer between the video and central memories. This method may not be adapted with other libraries like Open Vidia.

5.2.3. Hough transform

We ported the "HoughLines" OpenCV operator, intended to detect lines in a binary image. The principle is to sample the parameter space in a fragment program to find lines in a texture: for each point in the parameter space, the shader samples several pixels on the corresponding line in image space and counts those whose color is not null. The final color, for each point in the parameter space, is this count. We then find peaks in this image by thresholding it.

Similarly to generalized hough transform by [16], this method could be extended to detect any pattern by adding a certain number of rendering passes, one for each different pose of the pattern to detect. It is achievable with a single fragment shader, but we chose to speedup computing by deporting some trigonometric computations on the vertex shader, which is only possible with our system. At this time, this gpu-based hough transform is not really faster than the CPU one but it has the advantage of not breaking the gpu pipeline by tranferring image data back to the central memory.

5.2.4. Example application

We have partly ported to GPUCV a motion capture by computer vision application that is based on Open CV [17]. We

ported the image processing steps of the matching process just by replacing the names of the Open CV operators with their GPUCV equivalent. These very limited modifications achieved a 1.2 acceleration factor, which is an early result to be enhanced with further optimizations.

6. CONLUSION AND PERSPECTIVES

In this paper, we showed that commodity GPU can be used for image processing and analysis. We describe our open source library GPUCV that offers a powerful framework for integrating fragment shaders, vertex shaders and OpenGL processing while hiding graphics processing at application level. It brings GPU acceleration into native CPU Open CV applications. It is available at
http://picolibre.int-evry.fr/projects/gpucv.

7. REFERENCES

[1] R. Fernando and M.J. Kilgard, *Cg, the definitive guide to programmable real time graphics*, Addison-Wesley professionnal, 2004.

[2] R. Rost, *Open GL Shading Language*, Addison-Wesley professionnal, 2004.

[3] Microsoft corporation, "Directx developer center," http://msdn.microsoft.com/directx.

[4] M. Mac Cool and S. Du Toit, *Metaprogramming GPUs with Sh*, AK Peters, Inc, 2004.

[5] Stanford University Graphics Lab, "Brook gpu," http://graphics.stanford.edu/projects/brookgpu/.

[6] James Fung, Steve Mann, and Chris Aimone, "Openvidia: Parallel gpu computer vision," in *Proceedings of the ACM Multimedia 2005*, November 2005, pp. 849–852.

[7] K. Moreland and E. Angel, "The FFT on a GPU," in *SIGGRAPH/Eurographics Workshop on Graphics Hardware*. 2003, pp. 112–119, Eurographics Association.

[8] R. Strzodka and A. Telea, "Generalized Distance Transforms and skeletons in graphics hardware," in *Proceedings of EG/IEEE TCVG Symposium on Visualization (VisSym '04)*, 2004, pp. 221–230.

[9] R. Strzodka and C. Garbe, "Real-time motion estimation and visualization on graphics cards," in *Proceedings IEEE Visualization 2004*, 2004, pp. 545–552.

[10] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, New York, NY, USA, 1994, pp. 91–98, ACM Press.

[11] M. Rumpf and R. Strzodka, "Level set segmentation in graphics hardware," in *Proc. of IEEE International Conference on Image Processing (ICIP'01)*, 2001, vol. 3, pp. 1103–1106.

[12] M. Rumpf and R. Strzodka, "Nonlinear diffusion in graphics hardware," in *Proc. of EG/IEEE TCVG Symposium on Visualization (VisSym '01)*, 2001, pp. 75–84.

[13] F. Kelly and A. Kokaram, "Fast image interpolation for motion estimation using graphics hardware," in *IS&T SPIE Electronic Imaging - Real-Time Imaging VIII*, 2004.

[14] Kenneth E. Hoff III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver, "Fast computation of generalized Voronoi diagrams using graphics hardware," *j-COMP-GRAPHICS*, vol. 33, no. Annual Conference Series, pp. 277–286, 1999.

[15] G. IBradski, A. Kaehler, and Pisarevsky, "Learning-based computer vision with intel's open source computer vision library," *Intel Technology Journal*, 2005.

[16] R. Strzodka, I. Ihrke, and M. Magnor, "A graphics hardware implementation of the Generalized Hough Transform for fast object recognition, scale, and 3d pose detection," in *Proceedings of IEEE International Conference on Image Analysis and Processing (ICIAP'03)*, 2003, pp. 188–193.

[17] J. Marques Soares, P. Horain, A. Bideau, and Manh Hung Nguyen, "Acquisition 3d du geste par vision monoscopique en temps réel et téléprésence," in *actes de l'atelier "Acquisition du geste humain par vision artificielle et applications", Toulouse*, 2004.