Learning and Nonlinear Models (L&NLM) – Journal of the Brazilian Neural Network Society, Vol. 9, Iss. 2, pp. 130-137, 2011

AUTOMATIC GENERATION OF WRAPPER CODE FOR VIDEO PROCESSING FUNCTIONS

Daniel Oliveira Dantas, Junior Barrera
Instituto de Matemática e Estatística
Universidade de São Paulo
{ddantas,jb}@ime.usp.br

Abstract – Processing video with GPUs requires the use of an API such as OpenGL or CUDA API. Recent advances are libraries such as GPUCV, with fast operators that take advantage of the GPU processing power but hide from the user its programming complexities. However the implementation of new operators is not as simple as it can be, and in GPUCV it is limited by the built-in templates. Here we describe a code generator that, from two kinds of directives merged in a shader source code, generates a wrapper code with all the OpenGL or CUDA API calls needed before calling the shader, simplifying the creation and maintenance of a library of video processing operators. The proposed library performance is better than GPUCV for almost all the tested operators.

Keywords – Image processing, video processing, GPU, OpenGL, real time.

1. Introduction

The Graphics Processing Units, also known as GPU, are gaining more processing power very quickly. GPU's are built to render synthetic scenes faster than the CPU. Graphics processing requires calculations over big data-sets of vertices and fragments. Vertices are the fundamental elements of the polyhedra that compose a synthetic scene. Fragments are like pixels of the final image, but with various depths. In the rendering process, fragment colors are combined, or the fragment closer to the camera is chosen to compose the final image.

The great processing speed of the GPU is obtained through parallelization of the calculations. An example of currently available GPU is the NVIDIA GeForce GTX 280. It has 240 processors, or CUDA cores, and can process the same number of vertices or fragments at the same time. That GPU has a wide data bus, with 512 bits, that provides a bandwidth of 140 GBytes/s between the GPU and the video memory. The communication with the RAM (*Random Access Memory*) is done through a PCI Express 2.0 x16 bus, with a bandwidth of 8 GBytes/s.

It is natural to try to use such processing power to accelerate parallelizable calculations, like simulations based in differential equations, operations with matrices, image and signal processing [1]. In fact, computer vision researchers are creating libraries to accelerate calculations, like GPUCV [2] and OpenVidia [3].

The libraries GPUCV and OpenVidia were implemented using the OpenGL API (*Application Programming Interface*). But OpenGL is a computer graphics API, and using it in an application for which it was not designed is anti-natural and complex. GPUCV hides from the user the OpenGL complexities, but to add new functions, the user has to write his own wrapper or use the built-in templates. These templates allow only one output image, and input data must be stored in a data structure.

Shader is the name of the functions that run in the GPU. The two more common kinds of shaders are the *vertex shader*, that processes vertices, and the *fragment shader*, that processes fragments. The goal of this article is to describe how to create shaders easily, hiding the OpenGL complexities by using a wrapper code generator. Such complexities are not necessary to create video and image processing functions, induce to errors, make code implementation and maintenance difficult.

We will describe two languages that can be used to implement shaders, GLSL and CUDA. Both languages have advantages, and both would be useful in a video processing library. We will also describe how to chain code of many shaders easily, making possible the creation of a high performance video processing pipeline. Such chaining is possible through the use of a wrapper code, that prepares the data and sends it to the GPU so that the user does not worry about OpenGL. The wrapper code is generated, in compile time, from the shaders source code plus a few comments that follow some rules.

2. Languages for programming GPU

Here we present two languages used to program GPU. We chose GLSL for being part of the OpenGL specification, and for being very similar to other popular languages like Cg (*C for graphics*) and HLSL (*High Level Shading Language*). We also chose CUDA, for having a powerful feature absent in GLSL, which is the capacity of choosing which fragments are processed first.

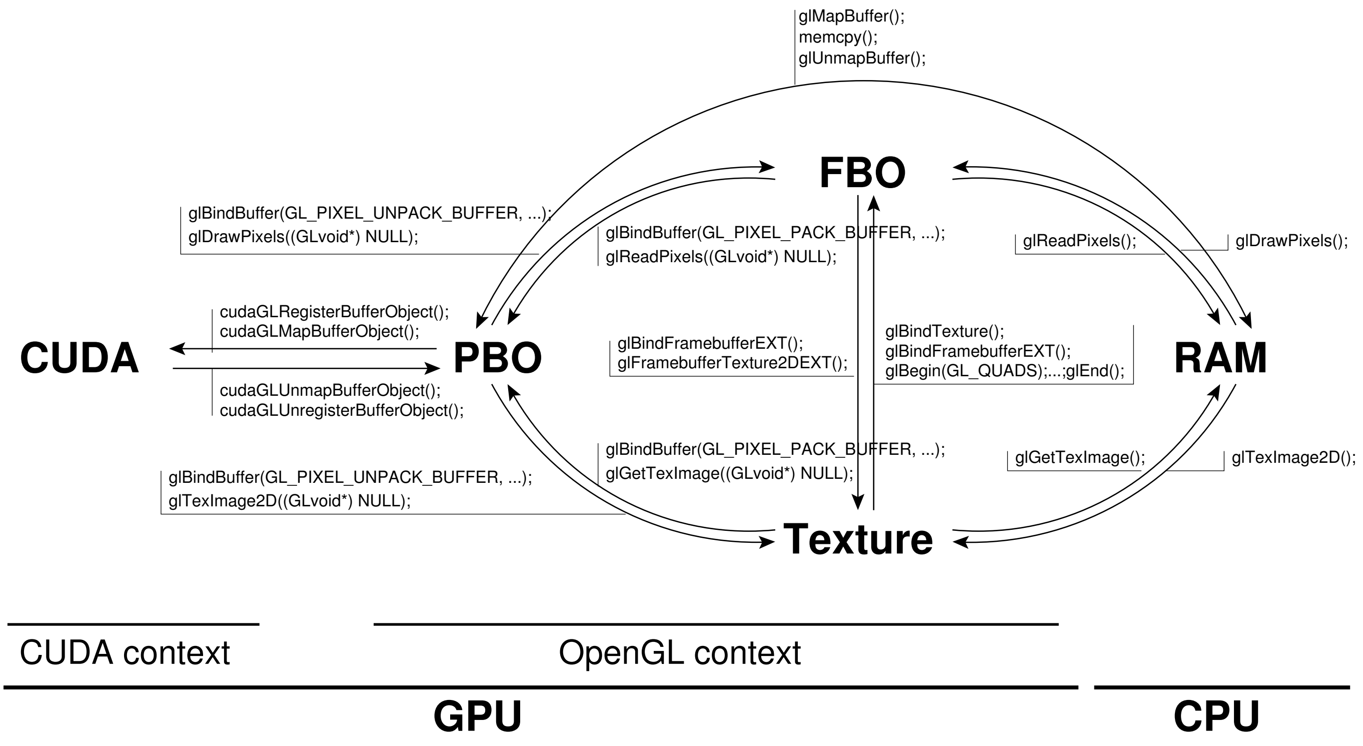
glMapBuffer();
memcpy();
glUnmapBuffer();
FBO
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, ...);
glDrawPixels((GLvoid*) NULL);
glBindBuffer(GL_PIXEL_PACK_BUFFER, ...);
glReadPixels((GLvoid*) NULL);
glReadPixels();
glDrawPixels();
cudaGLRegisterBufferObject();
cudaGLMapBufferObject();
CUDA
PBO
cudaGLUnmapBufferObject();
cudaGLUnregisterBufferObject();
glBindFramebufferEXT();
glFramebufferTexture2DEXT();
glBindTexture();
glBindFramebufferEXT();
glBegin(GL_QUADS);...;glEnd();
RAM
glBindBuffer(GL_PIXEL_PACK_BUFFER, ...);
glGetTexImage((GLvoid*) NULL);
glGetTexImage();
glTexImage2D();
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, ...);
glTexImage2D((GLvoid*) NULL);
Texture
CUDA context
OpenGL context
GPU
CPU

Figure 1: This figure shows the main function calls needed to transfer image data in RAM and in video memory. The proposed library calls those functions transparently to the user.

2.1 The GLSL language

The GLSL is a high level language, hardware independent, that is part of the OpenGL specification. It was designed following a series of directives and objectives described succinctly below [4].

The language should work with OpenGL, providing an alternative to its old fixed pipeline. Should expose hardware optimized resources, like data types and instructions present in the GPU. The OpenGL specification changes quickly to follow the technological developments of the GPU, so, GLSL should be flexible to expose hardware resources not yet invented.

The GLSL was created to be easy to use, and to avoid obsolescence. It was created with the same syntax of C, a popular and successful language for years, and with which many people who work with computer graphics are accustomed to use.

Supported data types are integer, float and boolean. The language natively supports operations with arrays of up to four positions, and square matrices. To access the texture memory, there are the data types `sampler1D`, `sampler2D` and `sampler3D`, for textures with one, two or three dimensions. Functions written in GLSL can have as input built-in variables and parameters, and can return values only through built-in variables. The names of the built-in variables start with `gl`.

The GLSL can be used in the implementation of image processing filters or operators, although it was not created with that objective. Each operator can be implemented as a fragment shader, and the input image must be stored as a texture.

An operation commonly done by fragment shaders is the texture mapping. Such shader can query the texture value from the correct position and change it in many ways. Can read texture values from a small window and combine them before writing the output. So, it is possible to implement operators like dilation, erosion, gradient, median filter and many others. The fragment shader can also combine values from many textures, so, it is possible to implement operators like minimum, maximum, sum and absolute difference between images.

2.2 The CUDA language

The acronym CUDA stands for *Compute Unified Device Architecture*. It is a new hardware and software architecture that allows the use of the GPU without mapping its instructions and data to a graphic API [5]. It is supported by the graphics cards GeForce 8 Series and newer.

Like GLSL, it is inspired by the C language to ease its learning, but code written in CUDA must be compiled with `nvcc`, compiler available in the CUDA SDK.

There are two kinds of functions that run in the GPU. Functions called by code that runs in CPU, which are preceded by the `__global__` qualifier, and functions called by code that runs in GPU, which are preceded by the `__device__` qualifier.

In the CUDA compatible GPU, there are many processors grouped together in multiprocessors. The programmer must divide the data in blocks, and the blocks in threads. Each block will run in a single multiprocessor. This division is done by defining

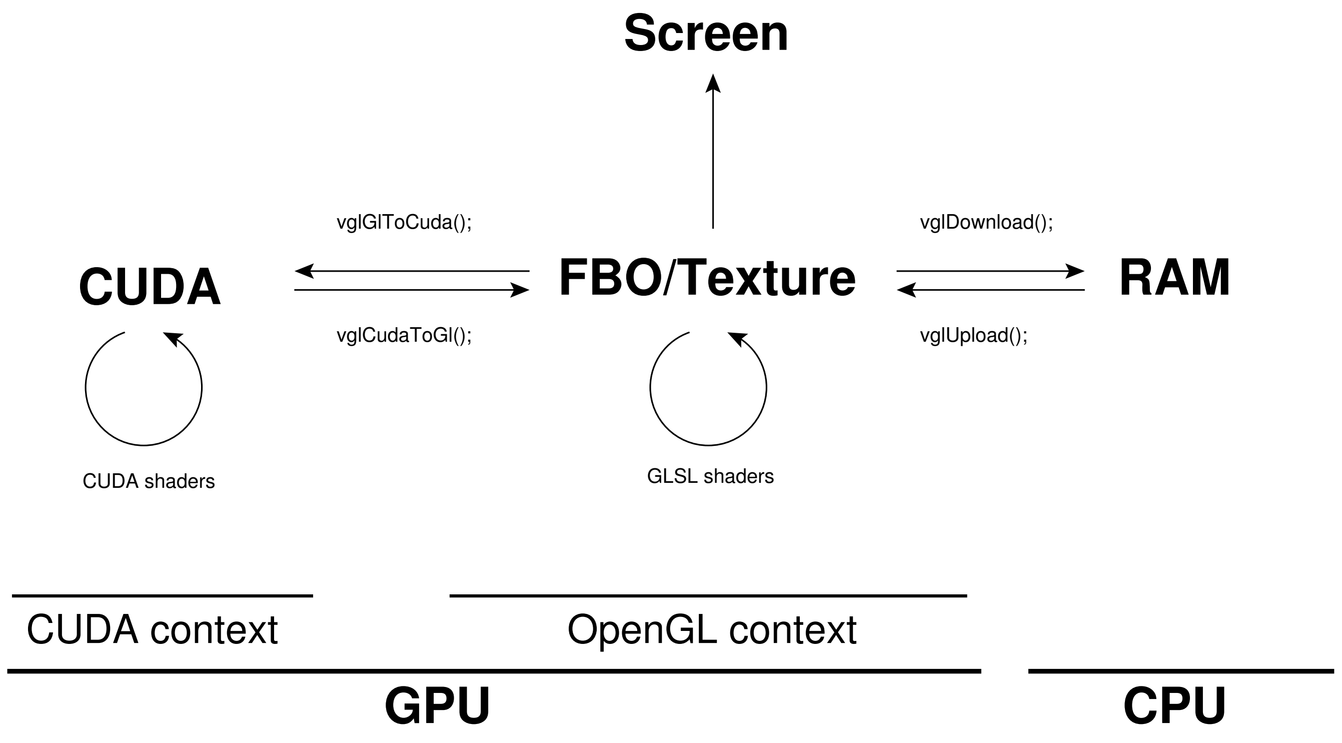

Figure 2: The commands of the proposed library hide from the user a series of API calls.

an execution configuration when calling a global function. An *execution configuration* contains the number of blocks and the number of threads per block.

There are four built-in constants visible in every `__global__` function: `gridDim` stores the number of blocks; `blockIdx` stores the block index; `blockDim` stores the number of threads per block; and `threadIdx` stores the thread index. The programmer must use these constants to define which index of the array or matrix each block and its threads will process. The CUDA language, differently from GLSL, requires a low level control by the user of which data will be used in function of the built-in constants. Such control make code written in CUDA more error prone than GLSL code, although it makes possible to define the order in which data is processed, feature required by dynamic programming algorithms for example.

3. Solutions available

There are at least three libraries that can be used in video processing: OpenCV, OpenVidia and GPUCV. OpenCV is a very popular library, quite comprehensive, with functions for image segmentation, object recognition, shape analysis, feature detection and tracking, and three-dimensional reconstruction [6]. Runs in the CPU, taking advantage of MMX/SSE instructions available in Intel® processors, but the parallelization is smaller than the reachable with GPU.

OpenVidia is a library with less image processing functions than OpenCV, but can take advantage of the processing power of the GPU [3]. It has functions implemented in CUDA and Cg. Among the functions available are a Canny edge detector, functions to register pairs of images, depth map from stereo images, feature tracking, Radon and Hough transforms. But OpenVidia does not hide from the user the complexities of OpenGL.

The GPUCV main goal is to allow the use of GPU in image processing with minimal effort [2, 7]. It has a subset of the OpenCV functions ported to GPU. To use GPUCV, it is not necessary to use OpenGL, but to create new functions the user has to write his own wrapper or use a small set of templates, and demands that the parameters are passed to the GPU through a data structure.

We will describe a wrapper code generator that hides from the user programming details that can be automatized. The user does not have to worry about creating wrapper functions that call the shaders, OpenGL API calls nor about how to convert data from the C++ language context to the context of GLSL and CUDA, focusing in the creation of operators and image filters.

4. Wrappers

Whenever an image is loaded from a file or captured from a camera, it is stored in a memory space belonging to the RAM context. To use the processing resources of the GPU, it is necessary to store the image in the video memory, in a logical space that belongs to the OpenGL context. That context is responsible not only for storing the image data, but also for running the shaders and for showing results in the screen. To be processed by a CUDA operator, it is necessary to transfer the image data from OpenGL to CUDA context.

To implement the wrappers, we created a special image container, of type `VglImage`, capable of storing the data in all the three contexts, RAM, OpenGL and CUDA. In RAM, the image is stored in an IplImage, compatible with OpenCV. In OpenGL, the image is stored in a texture, and the operators output needs a framebuffer, both stored as `GLuint` type handles. *Framebuffer* is the name given to the memory space where output from a shader is stored. In CUDA context, the image is stored in a GPU memory space defined by a pointer. The transfer between OpenGL and CUDA contexts needs an object called *Pixel Buffer Object*, stored as a `GLuint` type handle.

To control the transfers automatically, the `VglImage` structure stores the current context of each image. Operators run in a single context, CUDA or OpenGL. Input images must be in the same context as the operator when it is called; and output images will be in the same context as the operator when it finishes. So, the library copies automatically input images to the operator context when it is called, and changes the context of the output image to be the same as the operator when it finishes.

The Figure 1 shows, among other things, the functions used in the transfers between contexts. In the proposed library, the transfer is done automatically when necessary. It can also be forced by the user with a single function call, as shown in Figure 2.

The proposed library hides from the user the chain of calls to the OpenGL API. These calls are necessary for the pipeline of shaders to process the images correctly. In the video memory, each image is stored in a texture.

4.1 Wrapper GLSL

To generate the GLSL wrappers in C++ language, we use a Perl program that analyses each shader and generates two output files: a `h` file that contains the headers, and a `cpp` file, with the source code. Each shader is stored in a file with extension `frag`, and must contain a function `main`.

The files with the processed shaders are not changed, and must be available at runtime. The Perl program expects three arguments: the base name of the output files, the directory that contains the shaders, and the relative path of the shaders files so that the program can find them at runtime.

In the first time a shader is called by the application, the text file `frag` is loaded and compiled with `glCompileShader`. In case of success, the shader id is stored in a static variable, of type `GLuint`, inside the respective wrapper function in C++ so that the loading and compiling process is done only once.

The file `vglMipMap.frag` has, inserted in the GLSL compilable code, three kinds of comments that are interpreted by the preprocessor Perl to generate the wrapper function. The name of the function is the name of the `frag` file without extension.

The first comment, a multiple line comment between the lines `1` and `3`, is a *documentation comment*. It is copied exactly to the output files `h` and `cpp`. With these comments, it is possible to generate documentation by using, for example, `doxygen`.

The second comment, a single line comment occupying the whole line `4`, is a *declaration comment*. Its contents define the input parameters of the C++ function `vglMipMap`. It is a list, between parenthesis, of pairs <data type, variable name>, separated by comma like a C++ function parameter list. Valid types are `float`, `int` and `VglImage`. The `VglImage` type is used to store images, and must be preceded by one of the three semantic binding words: `IN_TEX`, `OUT_FBO` or `IN_OUT`, to indicate if the image is used as input, output, or both, respectively.

```
   Source code of vglMipMap.frag
1.  /** vglMipmap
2.  Get specified level of detail
3.  */
4.  // (IN_TEX: VglImage* src, OUT_FBO: VglImage* dst, float lod)
5.  uniform sampler2D sampler0;
6.  uniform float level; // lod
7.  void main(void){
8.      gl_FragColor = texture2DLod(sampler0, gl_TexCoord[0].xy, level);
9.  }
```

The third type of comment, the *attribution comment*, is the single line comment that appears after the definition of a `uniform` variable, like in line `6`. Defines which value the C++ code will attribute to the GLSL `uniform` variable. Each variable `uniform` of type `sampler` is related to a `VglImage` with input semantics in order of appearance in the code. In this case, the *attribution comments* are ignored.

It is necessary that each operator stores the output pixels in the exact position they should be, without translating the image after each call to an image operator. That is possible if we render a rectangle that occupies exactly the field of view of the camera, and mapping the texture that contains the output image to the rectangle using the desired fragment shader. As *modelview matrix* we use the identity, and the textures are mapped to a rectangle with extremities in (-1, -1) and (1, 1).

Each `VglImage` stores a handle to a texture and another to a FBO (*framebuffer object*). When the texture is attached to a FBO by calling `glBindFramebufferEXT()`, the shader output goes directly to the texture without the need of copying large regions of memory. In case of multiple output images, the output is also written to a single FBO, which must be temporarily attached to the multiple destination textures, each one corresponding to a different `GL_COLOR_ATTACHMENT` of this FBO. So it is possible to create functions that support multiple output images. These OpenGL features allow us to assemble a pipeline where both the input and output of each shader are stored in textures.

When there is a single output image, it is accessed from the GLSL code by writing on the built-in variable `gl_FragColor`. When there are two or more output images, they are accessed by writing on the `gl_FragData[]` array. The first output image has index 0 and so on.

Code generated from vglMipMap.frag

```
 1.  /** VglMipmap
 2.  Get specified level of detail.
 3.  */
 4.  void vglMipmap(VglImage* src, VglImage* dst, float lod){
 5.      vglCheckContext(src, VGL_GL_CONTEXT);
 6.      GLint _viewport[4];
 7.      glGetIntegerv(GL_VIEWPORT, _viewport);
 8.      static GLuint _f = 0;
 9.      if (_f == 0){
10.      fprintf(stdout, "FRAGMENT SHADER\n====================\n");
11.          _f = vglShaderLoad(GL_FRAGMENT_SHADER, (char*) "FS/vglMipmap.frag");
12.          if (!_f){
13.              fprintf(stderr, "%s:%s:  Error loading fragment shader.\n", __FILE__, __FUNCTION__);
14.              exit(1);
15.          }
16.      }
17.      glUseProgram(_f);
18.      glActiveTexture(GL_TEXTURE0);
19.      glBindTexture(GL_TEXTURE_2D, src->tex);
20.      glUniform1i(glGetUniformLocation(_f, "sampler0"), 0);
21.      glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, dst->fbo);
22.      glUniform1f(glGetUniformLocation(_f, "level"), lod);
23.      glViewport(0, 0, 2*dst->width, 2*dst->height);
24.      glBegin(GL_QUADS);
25.          glTexCoord2f( 0.0, 0.0);
26.          glVertex3f ( -1.0, -1.0, 0.0); //Left Up
27.          glTexCoord2f( 1.0, 0.0);
28.          glVertex3f ( 0.0, -1.0, 0.0); //Right Up
29.          glTexCoord2f( 1.0, 1.0);
30.          glVertex3f ( 0.0, 0.0, 0.0); //Right Bottom
31.          glTexCoord2f( 0.0, 1.0);
32.          glVertex3f ( -1.0, 0.0, 0.0); //Left Bottom
33.      glEnd();
34.      glUseProgram(0);
35.      glViewport(_viewport[0], _viewport[1], _viewport[2], _viewport[3]);
36.      if (dst->has_mipmap){
37.          glBindTexture(GL_TEXTURE_2D, dst->tex);
38.          glGenerateMipmapEXT(GL_TEXTURE_2D);
39.      }
40.      glActiveTexture(GL_TEXTURE0);
41.      vglSetContext(dst, VGL_GL_CONTEXT);
42.  }
```

The listing above shows the code generated from the source file `vglMipMap.frag`. The lines from `1` to `3` contain the multiple line *documentation comment*. Line `4` is generated from the *declaration comment* without the semantic bindings.

For each input image, that is, images with input semantics, which are `IN_TEX` or `IN_OUT`, a line like `5` is generated. It checks if the input images are in OpenGL context, that is, if the last time the image was changed, it was by an OpenGL function. If the image is in any other context, it is copied to the OpenGL context. Line `11` calls the function that loads, compiles and links the code from the file `vglMipMap.frag`.

Three lines, like the ones from `18` to `20`, are generated for each parameter of type `VglImage*` with input semantics. Each input image receives an index, starting from zero, and is appended to the constant name `GL_TEXTURE`, and is input parameter of the function `glUniform1i`. The input images obtained from the *declaration comment* are associated to the `sampler2D` variables contained in the shader source file in order of appearance.

The line `21` is generated for the first and only output image called `dst`. The output images are the ones with `OUT_FBO` or `IN_OUT` semantics. If there is more than one output image, unfortunately, the generated code is not very simple. Suppose that there is another output image called `dst1`. The code generated, placed after the line `21` would be like this:

```
21a.    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT1_EXT,
                                  GL_TEXTURE_2D, dst1->tex, 0);
21b.    glPushAttrib(GL_VIEWPORT_BIT | GL_COLOR_BUFFER_BIT);
21c.    GLenum buffers[] = {
21d.        GL_COLOR_ATTACHMENT0_EXT,
21e.        GL_COLOR_ATTACHMENT1_EXT, };
21f.    glDrawBuffers(2, buffers);
```

Besides images, other supported input types are `float` and `int`. Line `22` shows the function call that attributes the value of the variable `lod`, in the C context, to the variable `level`, in the GLSL context. When the variable is of type `int`, we call `glUniform1i` instead of calling `glUniform1f`.

For each output image, four lines like the ones from `36` to `39` are generated. They are responsible for filling the mipmap data if the texture was created with mipmaps.

Finally, for each output image, a line like `41` is generated. It saves the information that the correct image is in OpenGL context, and the information stored in RAM and CUDA context shall be ignored.

4.2 Wrapper CUDA

To generate the wrappers of CUDA in C++, we also use a Perl program, that analyses a list of files with extension `kernel`, and generates three output files: a file with extension `cu` containing all the wrappers of CUDA shaders; a file with extension `h` with all the prototypes of the wrapper functions; and a file with extension `kernel` with the CUDA shaders. The file `kernel` generated is nothing but the concatenation of every `kernel` file processed by the Perl program. It is included in the `cu` file by a C++ `#include` directive.

The Perl program expects two arguments: the base name of the output files; and the directory that contains the `kernel` files that define the shaders. Notice that in CUDA the operators are not loaded at runtime like in GLSL, so, the path to the files with shaders is not necessary. As can be noticed in the source code of `vglCudaInvert.kernel`, there are three kinds of comments like in the GLSL shaders. The name of the wrapper function is defined by the base name of the `kernel` file, so, our wrapper function will be called `vglCudaInvert`.

Source code of vglCudaInvert.kernel

```
 1.  /** vglCudaInvert
 2.  Inverts image in CUDA context.
 3.  */
 4.  // <<<in->h, 384>>>(IN_PBO: VglImage* In, OUT_PBO: VglImage* Out)
 5.  // (In->cudaPtr, Out->cudaPtr, In->width, In->height, In->nChannels)
 6.  template<typename T>
 7.  __global__ void global_Invert(T* in, T* out, int w, int h, int nCh){
 9.      T* arrIn = in + blockIdx.x * nCh * w;
10.      T* arrOut= out+ blockIdx.x * nCh * w;
11.      int minj = threadIdx.x;
12.      int maxj = nChan * w;
13.      int dj = blockDim.x;
14.      for (int j = minj; j < maxj; j += dj){
15.          arr_out[j] = -arr_in[j];
16.      }
17.  }
```

The first comment, a multiple line *documentation comment* in lines `1` to `3`, is copied to both `h` and `cpp` output files. The objective of the copy is to allow the use of `doxygen` to generate documentation.

In line `4` is a *declaration comment*. Its contents define the parameters of the C++ function `vglCudaInvert`. The first item in this comment, between the symbols "<<<" and ">>>", is an execution configuration, necessary in calls to CUDA functions. The shader defined in the `kernel` file will be called with that execution configuration. The second item is a list, between parenthesis, of pairs <data type, variable name> separated by comma, like in C++. Valid types are `float`, `int` and `VglImage`. The `VglImage` type is used to pass input and output images as parameters, and must be preceded by one of the three semantic binding words: `IN_PBO`, `OUT_PBO` or `IO_PBO`. The semantic binding indicates if the image is used as input, output, or both, respectively, and will be stored in a *Pixel Buffer Object*. In future implementations we may allow the use of textures, so, other semantics may be created.

The third type, the *attribution comment* in line `5`, defines the values that will be used to call the CUDA shader. It is a list, between parenthesis, of values separated by comma. These values will be used in the call of the shader, and the number of elements must be the same as the parameters the shader requires.

In CUDA the chanining of operators is easier than in GLSL, as there is no distinction between the memory regions used as input and output. The semantic binding is required only for updating the image context.

Code generated from vglCudaInvert.kernel

```
 1.  /** vglCudaInvert
 2.  Inverts image stored in cuda context.
 3.  */
 4.  void vglCudaInvert(VglImage* In, VglImage* Out){
 5.      vglCheckContext(In, VGL_CUDA_CONTEXT);
 6.      vglCheckContextForOutput(Out, VGL_CUDA_CONTEXT);
 7.      switch (In->depth){
 8.          case (IPL_DEPTH_8U):
 9.              global_Invert<<<In->height,384>>>((unsigned char*) In->cudaPtr,
                                    (unsigned char*) Out->cudaPtr,
                                    In->width, In->height, In->nChannels);
10.              break;
11.          default:
12.              fprintf(stderr, "vglCudaInvert:
                     Error:  unsupported img->depth = %d in file '%s' in line %i.\n",
                     input->depth, __FILE__, __LINE__);
13.              exit(1);
14.      }
15.      vglSetContext(Out, VGL_CUDA_CONTEXT);
16.  }
```

The listing above shows the code generated from the source file `vglCudaInvert.kernel`. The lines from `1` to `3` contain the multiple line *documentation comment*. Line `4` is generated from the *declaration comment* without the semantic bindings and the execution configuration.

For each input image, that is, images with input semantics, which are `IN_PBO` or `IO_PBO`, a line like `5` is generated. It checks if the input images are in CUDA context, that is, if the last time the image was changed, it was by a CUDA function. If the image is in any other context, it is copied to the CUDA context.

For each output image, that is, images with output semantics, which are `OUT_PBO` or `IO_PBO`, a line like `6` is generated. It checks if the output image was already allocated in CUDA context. When an image is created, the OpenGL texture is always allocated to allow the exhibition on screen. The CUDA space however is allocated only on demand.

The function call in line `9` contains the execution configuration extracted from the *declaration comment*, and the parameters are defined by the *attribution comment*. For each output image, a line like `15` is generated. It saves the information that the correct image is in CUDA context, and the information stored in RAM and OpenGL context shall be ignored.

5 Results

We compared the functions of the proposed library, which we will call VGL (or *VisionGL*), with equivalent functions of OpenCV and GPUCV. We used the command `runbench` from the application `GPUCVConsole`, available with the GPUCV library. To test our code, we added the calls to our functions to the `GPUCVConsole` source code and recompiled it.

The tests were done in an Athlon X2 with 2 GBytes of RAM and a GeForce 8800 GTS. Each operator was tested fifty times per library. The Table 1 show the average time in milliseconds for processing an RGB image with 1024×1024 pixels.

The VGL library gave results close to the GPUCV, some faster, and one, the GPU to CPU copy, slower as shown in Table 1, where we present all the operators of Farrugia [2] for comparison. The times do not include the compilation time of the shaders. The transfers between RAM (CPU) and video memory (GPU) are shown in the last two lines. The transfer from OpenGL to CUDA context is 8.8 ms, and the opposite is 0.3 ms. There are many functions implemented in the library but not shown in the table, for example parallel thinning, dense stereo calculation and background subtraction.

The tested shaders are written in GLSL in both GPUCV and VGL. The GPUCV library used was the revision 503 from its SVN repository. The OpenCV library used was the version 1.0.0.

6 Conclusion

We present in this article a library for processing video in real time, with better performance than GPUCV in all but one of the tested operators. The main advantage of our library is in the fact that it allows the creation of new operators easily in GLSL and in CUDA, and generates automatically their wrapper code. The generated wrapper code also transfers the image data between OpenGL, CUDA and RAM contexts automatically when necessary, or by calling a single function. It is also possible to create operators that support multiple output images.

A disadvantage of our library compared to GPUCV is the absence of a direct mapping between its functions and the ones in OpenCV. Future work may include to allow creation of shaders in CUDA that use texture fetch. We also plan to add support to OpenCL code.

The authors thank CNPq for the grant 142683/2006-0.

Table 1: Average time in milliseconds to process an image with 1024×1024 pixels

	OpenCV	GPUCV	VGL
Erosion 3×3	38.4	1.6	0.8
Erosion 5×5	63.4	3.3	2.1
RGB to XYZ	11.9	0.8	0.2
RGB to HSV	21.0	1.0	0.3
Threshold	2.2	0.7	0.2
Copy	3.1	1.3	0.2
Subtraction	7.8	0.9	0.3
CPU to GPU		10.3	5.1
GPU to CPU		5.3	8.5

References

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn and T. J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware". In *Eurographics 2005, State of the Art Reports*, pp. 21–51, August 2005.

[2] J. P. Farrugia, P. Horain, E. Guehenneux and Y. Alusse. "GPUCV: A Framework for Image Processing Acceleration with Graphics Processors". In *Multimedia and Expo, 2006 IEEE International Conference on*, pp. 585–588, 2006.

[3] J. Fung and S. Mann. "OpenVIDIA: parallel GPU computer vision". In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pp. 849–852, New York, NY, USA, 2005. ACM.

[4] R. J. Rost. *OpenGL(R) Shading Language*. Addison-Wesley, second edition, 2005.

[5] NVIDIA Corporation. *NVIDIA CUDA Architecture: introducion & overview*, April 2009. Version 1.1.

[6] G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly, Cambridge, MA, 2008.

[7] Y. Allusse, P. Horain, A. Agarwal and C. Saipriyadarshan. "Gpucv: an opensource gpu-accelerated framework forimage processing and computer vision." In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, pp. 1089–1092. ACM, 2008.