Playing Ping Pong with Render-To-Texture

Version 0.99

Dominik Göddeke, University of Dortmund, Germany
dominik.goeddeke@mathematik.uni-dortmund.de
http://www.mathematik.uni-dortmund.de/~goeddeke

Version history:
[29-11-2004] first draft, only Windows
[20-12-2004] minor updates and corrections
[24-02-2005] added some more details, corrected nasty performance promise
[25-03-2005] added warning about ATI and some hint for proper compilation
[02-04-2005] gave the tutorial a full rewrite to match the implementation that has changed quite a bit in the last couple of months
[05-04-2005] added GLSL version to sources
[15-04-2005] removed the pointless window, now really "offscreen"

This tutorial explains how to perform a simple vector operation completely on the GPU using the so-called ping pong approach and the open-source RenderTexture class written by Mark Harris (http://www.markmark.net). The code samples have only been tested on NV3X and NV4X hardware because of the unavailability of ATI hardware at my place. Additionally, Linux is unsupported because I didn't have the time yet.

The example is chosen because it is as simple as possible while not containing any graphics at all. However, it is not completely arbitrary. Full sources for this tutorial are provided on my web page.

Please feel free to complain about any bugs etc., but keep in mind that this tutorial is provided „as is“, without any guarantee or liability. If you find this short tutorial useful, a short e-mail is always welcome.

Preface

I assume you have basic knowledge of OpenGL and C/C++. The following packages need to be installed to start developing under MS Windows:

- NVIDIA SDK,: http://developer.nvidia.com, SDK 8.0 or better (or some other source of the proper headers and libraries for OpenGL)
- at least Cg 1.2 for NV3x, Cg 1.3 for NV4x: http://delevoper.nvidia.com (if you use Cg, for the GLSL version, you just need a driver that exposes the neccessary extensions)
- Microsoft Platform SDK: http://msdn.microsoft.com
- Microsoft Visual C++ 2003 or 2005 (any other C++ compiler should suffice, but has not been tested yet)
- The OpenGL Extension Wrapper Library (GLEW) http://glew.sourceforge.net

Compilation can be tricky at first, to get this bit of code compiled using VS, make sure you have the INCLUDE and LIB directories of all the packages mentioned above listed in the corresponding sections in the "project properties" dialog of VS. You might have to add "cg.lib cgGL.lib glew32.lib" to the "additional dependencies" section of the linker as well.

Mathematical Problem

In this tutorial, we are going to implement the so called **SAXPY_V** operation, a modification of the **SAXPY** function of the BLAS library. This operation can be considered as a low-level building block for all kinds of linear algebra operators and high-level linear solvers. For example, banded matrix vector operations arising in Finite Element discretisations can be implemented as a series of **SAXPY** calls. Given three vectors **Y**,**X** and **A**, all of length N and of 32-bit floating point precision, a CPU implementation using three arrays for the vectors would look like this:

```
FOR J=0 TO somefixednumberofiterations
    FOR I=0 TO N
        Y[I] = Y[I] + A[I]*X[I]
```

Note that the vectors **X** and **A** remain unchanged during the iteration, and the vector **Y** gets updated in each step.

To get this bit of code over to the GPU, we proceed like this: First, we create a data structure that maps well onto the GPU for our vectors. Then we set up an OpenGL program using GLUT. After that, we create an offscreen buffer in the GPU memory using the RenderTexture class. The actual math is performed through a simple Cg fragment shader program, which we initialise next. We then upload our vectors into the offscreen buffer and perform the iteration using a ping pong approach which will be explained in detail later on. After the iteration has finished, we download the results back into CPU memory and compare them with a solution calculated on the CPU.

Step 1: Arranging the data

In GPGPU calculations, textures are the equivalent of arrays used in CPU programs. In our case, we need three 1D arrays, one for **X**, **Y** and **A** respectively. Since 2D textures are way more efficient on GPUs and because of the limitation of 1D textures (they can contain at most 4096 elements, while 2D textures can contain 4096^2 elements), we need to map our 1D data into a 2D texture. For the sake of simplicity, we assume that the size of our vectors is N=2^k by 2^m for some appropriate integers k and m so that N<4096^2: Only the latest NV40 cards provide so called non-power-of-two textures natively without emulation, but we want this implementation runnable on NV30 GPUs. We decide to use one texture for all three vectors, storing the **Y** values in the red channel, and the **X** and **A** values in the green and blue channels. The actual mapping from 1D to 2D is implemented in a row-major way. The following code snippet gives the neccessary declarations:

```
typedef float rgb[3];
int N = 256*256;
int numIterations = 100;
texWidth = sqrt((float)N);
texHeight = texWidth;
```

The **createTextureData()** function creates such a 2D array (in CPU memory) that contains the data in a format well-suited to be used as a texture later on:

```
rgb* createTextureData () {
	// allocate memory
	rgb* p = (rgb*)malloc(N*sizeof(rgb));
	// and fill it with some arbitrary nonsense values
	for (int i=0; i<texWidth; i++)
	for (int j=0; j<texHeight; j++) {
		// red channel is y-Vector
		p[i*texWidth+j][0] = 0.0;
		// green channel is x-Vector
		p[i*texWidth+j][1] = 2.0/3.0;
		// blue channel is a-Vector
		p[i*texWidth+j][2] = 3.0/2.0;
	}
	return p;
}
```

Please keep in mind that this data layout is chosen for simplicity's sake, if you want to score proper GFLOPS, you might have to come up with a better idea, eventually taking advantage of the intrinsic SIMD capabilities and the parallelism on the GPU which is able to perform calculations on a four-tupel of data (RGBA) in one cycle. Some ideas how to do that are outlined in my "Performance Tuning Tutorial", available on the web.

Step 2: Setting up OpenGL, initialising the neccessary extensions

I won't go into details here, I assume you are able to do this. Getting pointers for extension functions can be painful, but luckily the GLEW library wraps all that up nicely. In the accompanying implementation, the setup takes place in the subroutines **initGLUT()** and **initGLEW()**. Note that GLEW automatically checks if all neccessary extensions are supported.

Step 3: Creating an offscreen buffer

The pBuffer extension to OpenGL (vendor-specific, in case of NVIDIA, it's called NV_FLOAT_BUFFER) allows the use of offscreen floating point rendering targets (which we definitely need because we don't want results to be clamped to [0,1]). We need to keep in mind one important detail: These buffers are either read-only (when bound as an input texture) or write-only (when bound as render target). Unfortunately, we need to read and update the **Y**-vector in each step. Basically, there are two solutions to this problem: We could use two pBuffers (the usual way with Linux at the moment), or we could use a double-buffered one. Each pBuffer has its own OpenGL context, so we would have to ensure that our data is shared between these contexts. Additionally, we would have to switch between these contexts in each iteration, which is expensive since changing the GL context implies a flush of the graphics pipeline. But luckily, pBuffers can be double-buffered. This means we will use one single buffer, which has two “surfaces”, one we read from and the other we write to. We also want to use the fast “render to texture” approach: In the first step, we render the results into a buffer which we then use as an input texture for the next step without any actual copying of data or anything else that would inhibit performance. This process is known as the **ping pong approach**.

The RenderTexture class graciously takes care of all the dirty details, so let's take a look at some code next:

```
const char *modeString = "rgb=32f doublebuffer texRECT rtt";
```

This is the initialisation string for the RenderTexture class. It translates into the following wishlist passed to the RenderTexture class in the next step: We want

- a double-buffered offscreen rendering target,
- with three channels of 32 bit precision each,
- supporting the fast “render to texture” approach (limited to Windows so far),
- and access to its data with the texRECT extension instead of the tex2D texture lookup.

Rectangular textures (aka texRECT) differ from regular textures: Their coordinates are in the range [0,texWidth] x [0,texHeight] as compared to [0,1] x [0,1]. When using rectangular textures, texels should be accessed at the texel center by adding 0.5 to its texture coordinates. To see what I mean, just output the WPOS value from below in your shaders. For example, the first texel is located at (0.5,0.5).

With this in mind, we can put the whole creation process into a single subroutine:

```
RenderTexture* createOffscreenBuffer(void) {
        // create new instance of RenderTexture class
        RenderTexture *rt = new RenderTexture();
        // set it up to suit our needs
        rt->Reset(modeString);
        // and initialise it to preferred size
        if (!rt->Initialize(texWidth, texHeight)) {
                // throw some error
        }
        // setup RT
        rt->BeginCapture();
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, texWidth, 0.0, texHeight);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glViewport(0, 0, texWidth, texHeight);
        glBindTexture(rt->GetTextureTarget(), rt->GetTextureID());
        glTexParameteri(rt->GetTextureTarget(), GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(rt->GetTextureTarget(), GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(rt->GetTextureTarget(), GL_TEXTURE_WRAP_S, GL_CLAMP);
        glTexParameteri(rt->GetTextureTarget(), GL_TEXTURE_WRAP_T, GL_CLAMP);
        rt->EndCapture();
        return rt;
}
```

In the first half of this routine, we just create the buffer using the narrow RenderTexture interface and set it up to run in the mode and size we require. The second half of this code snippet requires some more explanation: By calling **BeginCapture()**, we turn our buffer (to be more precise, its active surface) into the current OpenGL render target. Later on, we want to use a one to one mapping of the values in the vectors, the viewport and the textures, so we set up the RenderTexture instance to use a simple one-to-one 2D orthographic projection. We then bind it as a texture and set the OpenGL texture parameters to suit our needs. Note that this pBuffer has, as explained before, it's own context, so everything we do between the enclosing **BeginCapture()** and **EndCapture()** calls affects exactly the context we want to change. Keep this in mind, it is a common error if you start to modify this basic example.

Step 4a: Setting up Cg

To be able to use Cg, we first need to write a little Cg program that performs the actual calculation. We use a neat little fragment program for this. For the sake of simplicity, we inline it directly into the source code instead of calling the Cg compiler manually or through the Makefile. Our program takes a rectangle texture (through a variable of type **samplerRECT**) and the proper fragment coordinates (through the **WPOS** binding semantics) for the fragment as parameters, and calculates the result which is returned as a "color" value. Note that we don't have to worry about texture coordinates at all, since we set the viewport to full screen size earlier and will render something that serves as a stream data generator later on. All you need to know now to understand this fragment program is that it ***will be called independently for each value in our vectors***, meaning for each entry in our texture we set up earlier.

```
static char *fragmentSource =
  "float3 saxpy_v (in half4 screen : WPOS,\n "
  "           uniform samplerRECT texture)  : COLOR {\n "
  "  float3 OUT;\n"
  "  float3 tex = texRECT (texture, screen.xy);  // get y_i, x_i and a_i \n"
  "  OUT.r = tex.r + tex.g*tex.b;                // calc new y_i aka left-hand-side \n"
  "  OUT.gb = tex.gb;                            // copy over x_i and a_i for next iteration \n "
  "  return OUT;\n
}\n";
```

The subroutine **initCG()** takes care of setting up the Cg context the usual way. Check back with the accompanying sources for details. For future use, we get the **texture** parameter from the Cg runtime by calling

```
CGparameter textureParam = cgGetNamedParameter (fragmentProgram, "texture");
```

Step 4b: Setting up GLSL alternatively

GLSL is set up just like in any of the numerous tutorials available, I recommend taking a look at http://www.lighthouse3d.com/opengl/glsl/ or directly in the accompanying sources, the subroutine is called **initGLSL()**. The syntax of the GL shading language is a little bit different to Cg, the shader for our saxpy implementation looks like this:

```
const char *fragmentSource = {
"uniform sampler2DRect texture;\n"
"void main(void) {\n"
"  vec2 texcoord = gl_TexCoord[0].st;\n" // get texcoord and texture data
"  vec4 data = texture2DRect(texture, texcoord);\n"
"  data.x = data.x + data.y*data.z;\n" // perform saxpy
"  gl_FragColor = data;\n"
"}\n "
};
```

Step 5: Uploading the data to the offscreen buffer

This is the last preprocessing step: In order to work on proper data, we need to upload the starting values we defined earlier in an array on CPU memory for the iteration to the GPU. This is achieved by the following subroutine:

```
void uploadTexture () {
        // upload texture to FRONT buffer of the texture
        rgb *vec = createTextureData();
        rt->BeginCapture();
        glDrawBuffer(GL_FRONT_LEFT);
        glRasterPos2i(0,0);
        glDrawPixels (texWidth,texHeight,GL_RGB,GL_FLOAT,vec);
        rt->EndCapture();
}
```

Note that it is essential to tell OpenGL to interpret the data (which we created in the **createTextureData()** subroutine) as floating point values by passing **GL_FLOAT** to the corresponding method. The internal format needs to be set to **GL_RGB** because we initialised the buffer to have three channels.

Step 6: Performing the actual computation by playing ping pong

After all initialisations are done, we enter the drawing process by calling **render()**. Inside this routine, all neccessary steps are performed. Before we dive into the implementation, I will sketch out the basic idea of the **ping pong** approach: Since the two surfaces of our offscreen buffer are either read-only or write-only, we will store the input data in the read-only buffer and we will write the results of the computations into the output buffer. We are working with a graphics programming paradigm, so the input data is going to be bound as a texture. Reading from this texture and writing into the output buffer is the first step to be performed.

So much for the ping step, before we start the next iteration (aka pong), we just swap the role of the two buffers: The output buffer we just “rendered” to (I use quotation marks because we didn't create an image in the traditional sense) becomes the new input buffer, again as a texture. This is why the whole process is called **render to texture**. The former input buffer can be overwritten with the results of this iteration step since we don't need it anymore. Then we swap buffers again and start over. Luckily, the RenderTexture class takes care of the ugly internals, so all we have to do is to keep track of which buffer is being used as input and which buffer is being rendered to. The easiest way to do this is by a couple of macros:

```
const GLenum glsurf[2] = {GL_FRONT_LEFT, GL_BACK_LEFT};
const GLenum wglsurf[2] = {WGL_FRONT_LEFT_ARB, WGL_BACK_LEFT_ARB};
int SOURCE_BUFFER = 0;
#define DESTINATION_BUFFER !SOURCE_BUFFER
#define SWAP()  SOURCE_BUFFER = DESTINATION_BUFFER;
```

With all that in mind, we are ready to write the display callback routine:

```
void displayCallback() {
        // get RT, make it "writable"
        rt->BeginCapture();
        // bind Cg program
        cgGLBindProgram(fragmentProgram);
        cgGLEnableProfile(cgProfile);
        // tell the shader to use the right texture
        cgGLSetTextureParameter(textureParam, rt->GetTextureID());
        cgGLEnableTextureParameter(textureParam);
        // iterate test couple times
        for (int i=0; i<numIterations; i++) {
            // use BACK as render target
            glDrawBuffer(glsurf[DESTINATION_BUFFER]);
            // and FRONT as texture
            rt->BindBuffer(wglsurf[SOURCE_BUFFER]);
            // render viewport-sized quad. With our viewport, this makes sure the fragment
            // program is executed once per value (once per pixel/texel in the offscreen buffer)
            glBegin(GL_QUADS);
              glTexCoord2f(-texWidth,-texHeight); glVertex2f(-texWidth, -texHeight);
              glTexCoord2f(texWidth, -texHeight); glVertex2f(texWidth, -texHeight);
              glTexCoord2f(texWidth, texHeight); glVertex2f(texWidth, texHeight);
              glTexCoord2f(-texWidth, texHeight); glVertex2f(-texWidth, texHeight);
            glEnd();
            SWAP();
        }
        glFinish();
        // perfom some timing [...]
        // clean up and jump to postprocessing
        rt->EndCapture();
        cgGLDisableProfile(cgProfile);
        doPostprocessing();
    }
```

This is all the magic behind the ping pong approach. Note that all we render is a viewport-sized quad. This causes the rasterizer (a fixed part of the graphics pipeline we can't influence (yet?)) to create a fragment for each pixel in our viewport. The quad basically serves as a **data stream generator** for our fragment program which gets executed independently for each of these fragments. We set the texture coordinates (which are linearly interpolated between the corners of the quad) of our data texture to a one-to-one mapping between pixels and texels. In this way we are able to access the right positions in both the input and the output buffer, in each iteration: The index "I" in the CPU implementation directly translates to the index "WPOS" on the GPU.

After the loop in this subroutine is finished, all our calculations are done. There might be better ways to do this, but in this simple example we just call another routine (described in the next section) instead of exiting the GLUT main loop properly.

The timing needs some further commenting: Some GL calls execute asynchroneously, some don't. In order to get proper timings for GPGPU applications, we perform the same task quite often, and take the average runtime. If you take a look at the accompagnying implementation, please do not take the CPU timings seriously, the CPU code is a very straight forward cache-unfriendly implementation, so **please don't use these timings to measure any CPU vs. GPU ratio!**

The shader setup using GLSL is even easier:

```
rt->BeginCapture();
// enable GLSL program
glUseProgramObjectARB(programObject);
// identify the bound texture unit as input to the shader
glUniform1iARB(textureParam, 0);
for (int i=0; i<numIterations; i++) {
            // ... no other changes
}
glFinish();
```

Step 7: Postprocessing

In this last step, we just download the results from GPU memory for future use. In the accompanying implementation, the results are simply compared with a reference solution calculated on the CPU. Even this is beyond the scope of this tutorial. Here, we just sketch the download subroutine:

```
rgb *downloadTexture (GLenum target) {
        rgb *data = (rgb*)malloc(sqrtn*sqrtn*sizeof(rgb));
        rt->BeginCapture();
        glReadBuffer(target);
        glReadPixels(0, 0, texWidth, texHeight,GL_RGB,GL_FLOAT,data);
        rt->EndCapture();
        return data;
}
```

One error-prone detail remains: In the last iteration of the loop inside the display callback routine, we called **SWAP()**. So in order to get the correct results, we have to download them from the source buffer and not from the destination buffer, somewhat contrary to common sense.

Closing remark

Ok, you have reached the end of this tutorial, I hope you enjoyed it and found it useful.

If you look at the output of the program closely, you will notice two strange WIN32 error messages about missing PROCs: One is caused inside the initGLEW subroutine, the other is thrown while setting optimal parameters for the Cg runtime. Both errors don't affect this program. If you know why they occur, please drop me a note.

To dive further into the topic of GPGPU programming, I recommend the community site at http://www.gpgpu.org which comes with a ton of information and a great, active forum. If you prefer reading hardcovers, the text book GPU GEMS 2 comes with a a whole chapter of articles on GPGPU programming. Additionally, you might find some more tutorials useful which I gradually make available on my web page.