GPGPU – Basic Math Tutorial

Dipl. Inform. Dominik Göddeke

Lehrstuhl für Angewandte Mathematik und Numerik
Lehrstuhl für Computergrafik
Universität Dortmund

dominik.goeddeke@math.uni-dortmund.de
http://www.mathematik.uni-dortmund.de/∼goeddeke

Contents

1 Introduction

The goal of this tutorial is to explain the background and all necessary steps that are required to implement a simple linear algebra operator on the GPU: `saxpy()` as known from the BLAS library. For two vectors $\mathbf{x}$ and $\mathbf{y}$ of length N and a scalar value $alpha$, we want to compute a scaled vector-vector addition: $\mathbf{y} = \mathbf{y} + alpha * \mathbf{x}$. The `saxpy()` operation requires almost no background in linear algebra, and serves well to illustrate all entry-level GPGPU concepts. The techniques and implementation details introduced in this tutorial can easily be extended to more complex calculations on GPUs.

1.1 Prerequisites

This tutorial is based on **OpenGL**, simply because the target platform should not be limited to MS Windows. Most concepts explained here however translate directly to **DirectX**.
This tutorial is not intended to explain every single detail from scratch. It is written for programmers with a basic understanding of **OpenGL**, its **state machine concept** and the way OpenGL models the graphics pipeline.
For a good overview and pointers to reading material, please refer to the GPGPU community web page[1].
Updates of this tutorial are available on my homepage.

1.2 Hardware requirements

You will need at least a NVIDIA GeForce FX or an ATI RADEON 9500 graphics card. Older GPUs do not provide the features (most importantly, single precision floating point data storage and computation) which we require.
WARNING: Due to immature OpenGL driver support with respect to the relatively new techniques used in this tutorial, the accompagnying code will only work partially on ATI and GeForce FX hardware. This limitation will be removed in future driver releases.

1.3 Software requirements

First of all, a C/C++ compiler is required. Visual Studio .NET 2003, Eclipse 3.1 plus CDT/MinGW, the Intel C++ Compiler 9.0 and GCC 3.4+ have been successfully tested. Up-to-date drivers for the graphics card are essential. At the time of writing, using an ATI card only works with Windows, whereas NVIDIA drivers support both Windows and Linux.
The accompanying code uses two external libraries, **GLUT** and **GLEW** which need to be installed. Shader support for GLSL (see below) is built into the driver, and the Cg toolkit can be downloaded from NVIDIA's developer page.

1.4 Alternatives

For a similar example program done in DirectX, refer to Jens Krüger's *Implicit Water Surface*[2] demo (there is also a version based on OpenGL available). This is however well-commented example code and not a tutorial.
GPU metaprogramming languages abstract from the graphical context completely. Both *BrookGPU*[3] and *Sh*[4] are recommended.

2 Setting up OpenGL

2.1 GLUT

GLUT, the **OpenGL Utility Toolkit**, provides functions to handle window events, create simple menus etc. Here, we just use it to set up a valid OpenGL context (allowing us access to the graphics hardware through the GL API later on) with as few code lines as possible. Additionally, this approach is completely independent of the window system that is actually running on the computer (MS-Windows or Xfree/Xorg on Linux / Unix and Mac).

[1] `http://www.gpgpu.org`
[2] `http://wwwcg.in.tum.de/Research/Publications/LinAlg`
[3] `http://graphics.stanford.edu/projects/brookgpu/`
[4] `http://libsh.org/`

```
// include the GLUT header file
#include <GL/glut.h>
// call this and pass the command line arguments from main()
void initGLUT(int argc, char **argv) {
    glutInit ( &argc, argv );
    glutCreateWindow("SAXPY TESTS");
}
```

2.2 OpenGL extensions

Most of the features that are required to perform general floating-point computations on the GPU are not part of core OpenGL. OpenGL Extensions however provide a mechanism to access features of the hardware through extensions to the OpenGL API. They are typically not supported by every type of hardware and by every driver release because they are designed to expose new features of the hardware (such as those we need) to application programmers. In a *real* application, carefully checking if the necessary extensions are supported and implementing a fallback to software otherwise is required. In this tutorial, we skip this to prevent code obfuscation.
A list of (almost all) OpenGL extensions including specifications and examples is available at the OpenGL Extension Registry[5].
The extensions actually required for this implementation will be presented when we need the functionality they provide in our code. The small tool **glewinfo** that ships with GLEW, or any other OpenGL extension viewer, or even OpenGL itself (an example can be found when following the link above) can be used to check if the hardware and driver support a given extension.
Obtaining pointers to the functions the extensions define is an advanced issue, so in this example, we use GLEW as an extension loading library that wraps everything we need up nicely with a minimalistic interface:

```
void initGLEW (void) {
    // init GLEW, obtain function pointers
    int err = glewInit();
    // Warning: This does not check if all extensions used
    // in a given implementation are actually supported.
    // Function entry points created by glewInit() will be
    // NULL in that case!
    if (GLEW_OK != err) {
        printf((char*)glewGetErrorString(err));
        exit(ERROR_GLEW);
    }
}
```

2.3 Preparing OpenGL for offscreen rendering

In the GPU pipeline, the traditional end point of every rendering operation is the frame buffer, a special chunk of graphics memory from which the image that appears on the display is read. Depending on the display settings, the most we can get is 32 bits of color depth, shared among the red, green, blue and alpha channels of your display: Eight bits to represent the amount of ”red” in a image (same for green etc.: RGBA) is all we can expect and in fact need on the display. This already sums up to more than 16 million different colors. Since we want to work with floating point values, 8 bits is clearly insufficient with respect to precision. Another problem is that the data will always be **clamped** to the range of $[0/255; 255/255]$ once it reaches the framebuffer.
How can we work around this? We could invent a cumbersome arithmetics that maps the sign-mantissa-exponent data format of an IEEE 32-bit floating point value into the four 8-bit channels. But luckily, we don’t have to! First, 32-bit floating poit values on GPUs are provided through a set of OpenGL extensions (see section 2). Second, an OpenGL extension called **EXT_framebuffer_object** allows us to use an **offscreen buffer** as the target for rendering operations such as our vector calculations, providing full precision and removing all the unwanted clamping issues. The commonly used abbreviation is **FBO**, short for framebuffer object.
To use this extension and to turn off the traditional framebuffer and use an offscreen buffer (surface) for our calculations, a few lines of code suffice. Note that **binding** FBO number 0 will restore the window-system specific framebuffer at any time. This is useful for advanced applications but beyond the scope of this tutorial.

```
GLuint fb;

void initFBO(void) {
    // create FBO (off-screen framebuffer)
    glGenFramebuffersEXT(1, &fb);
    // bind offscreen buffer
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
}
```

[5] http://oss.sgi.com/projects/ogl-sample/registry/

3 GPGPU concept 1: Arrays = textures

One-dimensional **arrays** are the native CPU data layout. Higher-dimensional arrays are typically accessed (at least once the compiler is done with it) by offsetting coordinates in a large 1D array. An example for this is the row-wise mapping of a two-dimensional array `a[i][j]` of dimensions M and N into the one-dimensional array `a[i*M+j]`, assuming array indices start with zero (as in C, C++ and Java but not in Fortran).
For GPUs, the native data layout is a two-dimensional array. One- and three-dimensional arrays are also supported, but they either impose a performance penalty or cannot be used directly with the techniques we employ in this tutorial. Arrays in GPU memory are called **textures** or **texture samplers**. Texture dimensions are limited on GPUs, the maximum value in each dimension can be queried with a bit of code like this once a valid OpenGL context is available (that is, once GLUT is initialized):

```
int maxtexsize;
glGetIntegerv(GL_MAX_TEXTURE_SIZE,&maxtexsize);
printf("GL_MAX_TEXTURE_SIZE, %d \n",maxtexsize);
```

On today's cards, the resulting value is 2048 or 4096 per dimension. Be warned though that although a given card seems to support three-dimensional floating point textures of size 4096*4096*4096, the available graphics memory is still a hard limit!
On the CPU, we usually talk about **array indices**, on the GPU, we will need **texture coordinates** to access values stored in the textures. Texture coordinates need to access texel centers.
Traditionally, GPUs work on four-tupels of data simultaneously: There are four color channels called red, green, blue and alpha (RGBA). We will explain later on how we can exploit this to speed up our implementation on certain hardware.

3.1 Creating arrays on the CPU

Let us recall the calculation we want to perform: $\mathbf{y} = \mathbf{y} + alpha * \mathbf{x}$ for a given vector length N. We need two arrays containing floating point values and a single float value to do this:

```
float* dataY = new float[N];
float* dataX = new float[N];
float alpha;
```

Although the actual computation will be performed on the GPU, we still need to allocate these arrays on the CPU and fill them with initial values.

3.2 Creating floating point textures on the GPU

This topic requires quite a lot of explanation, so let us first recall that on the CPU, we simply need two arrays with floating point values. On the GPU, we use **floating point textures** to store the data.
The first important complication is that we have a variety of different so-called **texture targets** available. Even if we skip the non-native targets and only restrict ourselves to two-dimensional textures, we are left with two choices. **GL_TEXTURE_2D** is the traditional OpenGL two-dimensional texture target, referred to as **texture2D** throughout this tutorial. **ARB_texture_rectangle** is an OpenGL extension that provides so-called **texture rectangles**, sometimes easier to use for programmers without a graphics background. There are two conceptual differences between texture2Ds and texture rectangles, which we list here for reference. We will work through some examples later on.

texture2D **texture target** GL_TEXTURE_2D

- **texture coordinates** Coordinates have to be normalized to the range $[0;1]$ by $[0;1]$, independent of the dimension $[0;M]$ by $[0;N]$ of the texture.
- **texture dimensions** Dimensions are constrained to powers of two (e.g. 1024 by 512) unless the driver supports the extension **ARB_non_power_of_two or** unless the driver exposes OpenGL 2.0 which alleviates this restriction.

texture rectangle **texture target** GL_TEXTURE_2D

- **texture coordinates** Coordinates are not normalized.
- **texture dimensions** Dimensions can be arbitrary by definition, e.g. 513 by 1025.

The next important decision affects the **texture format**. GPUs allow for the simultaneous processing of scalars, tupels, tripels or four-tupels of data. In this tutorial, we focus on scalars and four-tupels exemplarily. The easier case is to allocate a texture that stores a single floating point value per texel. In OpenGL, **GL_LUMINANCE** is the texture format to be used for this. To use all four channels, the texture format is **GL_RGBA**. This means that we store four floating point values per texel, one in the red color channel, one in the green channel and so on. For single precision floating point values, a LUMINANCE texture will consume 32 bits (4 bytes) of memory per texel, and a RGBA texture requires 4*32=128 bits (16 bytes) per texel.
Now it gets really tricky: There are three extensions to OpenGL that expose true single precision floating point values as **internal format** for textures, `NV_float_buffer`, `ATI_texture_float` and `ARB_texture_float`. Each extension defines a set of enumerants (for example GL_FLOAT_R32_NV) and symbols (for example 0x8880) that can be used to define and allocate textures, as described later on. The **NV_float_buffer** extension should be considered legacy for NVIDIA GeForce FX (GeForce 5) class hardware although it is still supported in later generations of NVIDIA GPUs. This extension can only be used with texture rectangles. The enumerants for the two texture formats we are interested in here are **GL_FLOAT_R32_NV** and **GL_FLOAT_RGBA32_NV**. The first enumerant tells the GL that we want to store a single floating point value per texel, the latter stores a 4-tupel of floating point values in each texel. The two extensions **ATI_texture_float** and **ARB_texture_float** are identical from our point of view except that they define different enumerants for the same symbols. It is a matter of preference which one to use, because they are supported on both GeForce 6 (and better) and ATI hardware. The enumerants are **GL_LUMINANCE_FLOAT32_ATI**, **GL_RGBA_FLOAT32_ATI** and **GL_LUMINANCE32F_ARB**, **GL_RGBA32F_ARB** respectively. In this tutorial, we use the ARB extension.
The last problem we have to tackle is the question how to map a vector on the CPU into a texture on the GPU. We choose the easiest mapping that comes to mind: A vector of length N is mapped into a texture of size $\sqrt{N}$ by $\sqrt{N}$ for LUMINANCE formats (this means we assume N is a power of two), and into a texture of size $\sqrt{N/4}$ by $\sqrt{N/4}$ for RGBA formats, again assuming N is chosen so that the mapping "fits". For instance, $N = 1024^2$ yields a texture of size 512 by 512. We store the corresponding value in the variable **texSize**.
The following list summmarizes all the things we just discussed, sorted by the available GPU types: NVIDIA GeForce FX (NV3x), GeForce 6 and 7 (NV4x, G70) and ATI.

NV3x target texture rectangle only

format one to four channels

internal format NV_float_buffer

NV4x and better target texture2D and texture rectangle

format one to four channels

internal format NV_float_buffer (see below), ATI_texture_float, ARB_texture_float

ATI target texture2D and texture rectangle

format one to four channels

internal format ATI_texture_float

One additional remark: NVIDIA also supports the extension **ARB_color_buffer_float** on NV4x and better GPUs, which (among other things) effectively allows **NV_float_buffer** to be used in combination with texture2Ds.
After this large theory section, it is time to go back to some code. Luckily, allocating a texture is very easy once we know which **texture target**, **texture format** and **internal format** we want to use:

```
// create a new texture name
GLuint texID;
glGenTextures (1, &texID);
// bind the texture name to a texture target
glBindTexture(texture_target,texID);
// turn off filtering and set proper wrap mode
// (obligatory for float textures atm)
glTexParameteri(texture_target, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(texture_target, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(texture_target, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(texture_target, GL_TEXTURE_WRAP_T, GL_CLAMP);
// and allocate graphics memory
glTexImage2D(texture_target, 0, internal_format,
             texSize, texSize, 0, texture_format, GL_FLOAT, 0);
```

Let us digest this last OpenGL call one parameter at a time: We already know what `texture_target` should be. The next parameter (set to 0) tells the GL not to use any mipmap levels for this texture. The internal format

is clear, and so should be the `texSize` parameter. The next parameter (again set to 0) turns off borders for our texture because we don't need them. The texture format chooses the number of channels, as explained above. The parameter `GL_FLOAT` should not be misinterpreted: It has nothing to do with the precision of the values we want to store in the texture, it is only relevant on the CPU side because it tells the GL that the actual data which gets passed in later calls is floating point. The last parameter (set to 0) simply tells the GL that we do not want to specify any data for the texture right now. This call therefore results in a properly allocated texture corresponding to the settings we decided upon before.
One last remark: Choosing a proper data layout, that is, a mapping between texture formats, texture sizes and your CPU data, is a very problem-depending question. Experience shows that for some cases, defining such a mapping is obvious and in other cases, this takes up most of your time. Suboptimal mappings can seriously impact performance!

3.3 One-to-one mapping from array index to texture coordinates

Later on in this tutorial, we update our data stored in textures by a rendering operation. To be able to control exactly which data elements we compute or access from texture memory, we will need to choose a special projection that maps from the 3D world (world or model coordinate space) to the 2D screen (screen or display coordinate space), and additionally a 1:1 mapping between pixels (which we want to render to) and texels (which we access data from). The key to success here is to choose an orthogonal projection and a proper viewport that will enable a one to one mapping between geometry coordinates (used in rendering) and texture coordinates (used for data input) and pixel coordinates (used for data output). The mapping is based on the only value we have available so far, the size (in each dimension) we allocate textures with. One warning though: With texture2Ds, some scaling for the texture coordinates as explained above is still required. This is typically done in another part of the implementation and we explain it once we get there. With the projection and viewport set here, this is however trivial.
To set up the mapping, we essentially set the z coordinate in world space to zero and apply the $1:1$ mapping: The following lines can be added to the `initFBO()` routine:

```
// viewport for 1:1 pixel=texel=geometry mapping
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, texSize, 0.0, texSize);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, texSize, texSize);
```

3.4 Using textures as render targets

One key functionality to achieve good performance rates is the possibility to use textures not only for data input, but also for data output: With the framebuffer_object extension, we can **render directly to a texture**. The only drawback is: Textures are either **read-only** or **write-only**. GPU hardware design provides an explanation: Internally, GPUs schedule rendering tasks into several pipelines working in parallel, independent of each other. We will discuss this later on in more detail. Allowing for simultaneous reads from and writes into the same texture would require an awful lot of logic to prevent reading from a previously modified position (read-modify-write). Even if that chip logic was available, there would be no way to implement this (in hardware or software) without seriously inhibiting performance: GPUs are not instruction-stream based von Neumann architectures, but **data-stream based architectures**. In our implementation, we thus need three textures for the two data vectors: One texture (read-only) is used for the vector $\mathbf{x}$, another read-only texture for the input vector $\mathbf{y}$ and a third write-only texture that contains the result of the computation. This approach basically means that we rewrite our original calculation $\mathbf{y} = \mathbf{y} + alpha * \mathbf{x}$ to this: $\mathbf{y_{new}} = \mathbf{y_{old}} + alpha * \mathbf{x}$.
The framebuffer object extension provides a very narrow interface to **render to a texture**. To use a texture as render target, we have to **attach the texture to the FBO**:

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT,
                          texture_target, texID, 0);
```

The first parameter is obligatory. The second parameter defines the attachment point (up to four different texture attachments are supported per FBO, this depends on the hardware and can be queried using `GL_MAX_COLOR_ATTACHMENTS_EXT`). The third and fourth parameter should be clear; they identify the actual texture to attach. The last parameter selects the mipmap level of the texture, we do not use mipmapping so we simply set it to zero.

Unfortunately, the framebuffer_object specification only defines texture attachments with the format GL_RGB or GL_RGBA (the latter being important for us). LUMINANCE attachments will be defined in a follow-up extension. At the time of writing, NVIDIA hardware and the NVIDIA driver support them nontheless, but only in combination with the enumerants defined in **NV_float_buffer**. In other words, the distinction between formats that are allowed as floating point textures and formats that are allowed as floating point render targets or more precise, floating point color attachments, is essential. **Renderable texture formats** is an unofficial term to make this distinction.
Note that in order to successfully attach a texture, it just has to be allocated and defined by means of **glTexImage2D()**, it does not need to contain any useful data. It is a good analogy to think of FBOs as *structs of pointers*, in order to redirect rendering operations to a texture attachment, all we (conceptually) need to do is some pointer manipulation by means of OpenGL calls.

3.5 Transferring data from CPU arrays to GPU textures

To transfer data (like the two vectors dataX and dataY we created previously) to a texture, we have to bind the texture to a texture target and schedule the data for transfer with an OpenGL call. It is essential that the array passed to the function as a pointer parameter is properly dimensioned. In the case of our vectors, LUMINANCE format implies the array must contain texSize by texSize elements, and for RGBA formats, we need an additional factor of four more elements. Since we use GL_FLOAT, the data has to be a pointer to an array of floats as defined above. Note that we have absolutely no control when the data will actually be transferred to graphics memory, this is entirely left to the driver. We can however be sure that once the GL call returns, we can safely alter the data on the CPU side without affecting the texture. Additionally we are guaranteed that the data will be available when we access the texture for the next time. That being said, we can dive into the code. On NVIDIA hardware, the following code is hardware-accelerated:

```
glBindTexture(texture_target, texID);
glTexSubImage2D(texture_target,0,0,0,texSize,texSize,
                texture_format,GL_FLOAT,data);
```

The three zeros we pass as parameters define the offset and the mipmap level. We will ignore all of them because we do not use mipmaps and because we transfer a whole vector at once.
On ATI hardware, the preferred technique is to transfer data to a texture that is already attached to a framebuffer object by just redirecting the OpenGL render target to the attachment and by issueing a conventional OpenGL framebuffer image manipulation call:

```
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
glRasterPos2i(0,0);
glDrawPixels(texSize,texSize,texture_format,GL_FLOAT,data);
```

The first call redirects the output. In the the second call, we use the origin as the reference position because we download the whole chunk of data into the texture with the last call.
In both cases, the CPU array is mapped row-wise to the texture. More detailed: For RGBA formats, the first four array elements end up in the red to alpha components of the first texel and so on. For LUMINANCE textures, the first two texels in a row contain the first two components of the data vector.

3.6 Transferring data from GPU textures to CPU arrays

The other way round, there are again two alternative ways to implement transfers from GPU textures to CPU arrays. The traditional OpenGL texturing approach involves binding the texture to a texture target and calling **glGetTexImage()**. The parameters should be clear by now:

```
glBindTexture(texture_target,texID);
glGetTexImage(texture_target,0,texture_format,GL_FLOAT,data);
```

If the texture to be read back to the host is already attached to a FBO attachment point, we can again perform the pointer redirection technique:

```
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0,0,texSize,texSize,texture_format,GL_FLOAT,data);
```

Since we upload the whole texture from GPU memory to CPU memory, we pass the origin as the first two parameters. This technique is recommended.
One word of advice: Data transfers between main memory and GPU memory are expensive compared to computations on the GPU, so they should be used sparingly.

3.7 A short example program

Now it is time to lean back a little. I strongly suggest to start toying around before moving on to more advanced topics. Write a little example program and try to define textures with varying formats, targets and internal formats. Download data into them and read back the data again to a different CPU array to get acquainted with the techniques and implementation details. Try to put the different pieces of the puzzle together into a running program! Sections A.1 through A.4 about error checking in the appendix should be consulted to avoid problems.

For reference, this is the most minimalistic program I have come up with to achieve round trips, exemplarily using texture rectangles and **ARB_texture_float**:

```
#include <stdio.h>
#include <stdlib.h>
#include <GL/glew.h>
#include <GL/glut.h>

int main(int argc, char **argv) {
    // declare texture size, the actual data will be a vector
    // of size texSize*texSize*4
    int texSize = 2;
    // create test data
    float* data = new float[texSize*texSize*4];
    float* result = new float[texSize*texSize*4];
    for (int i=0; i<texSize*texSize*4; i++)
        data[i] = i+1.0;
    // set up glut to get valid GL context and
    // get extension entry points
    glutInit ( &argc, argv );
    glutCreateWindow("TEST1");
    glewInit();
    // viewport transform for 1:1 pixel=texel=data mapping
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, texSize, 0.0, texSize);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glViewport(0, 0, texSize, texSize);
    // create FBO and bind it (that is, use offscreen render target)
    GLuint fb;
    glGenFramebuffersEXT(1, &fb);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
    // create texture
    GLuint tex;
    glGenTextures (1, &tex);
    glBindTexture(GL_TEXTURE_RECTANGLE_ARB,tex);
    // set texture parameters
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                    GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                    GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                    GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                    GL_TEXTURE_WRAP_T, GL_CLAMP);
    // define texture with floating point format
    glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,0,GL_RGBA32F_ARB,
                 texSize,texSize,0,GL_RGBA,GL_FLOAT,0);
    // attach texture
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                              GL_COLOR_ATTACHMENT0_EXT,
                              GL_TEXTURE_RECTANGLE_ARB, tex, 0);
    // transfer data to texture
    glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,0,0,0,texSize,texSize,
                    GL_RGBA,GL_FLOAT,data);
    // and read back
    glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
    glReadPixels(0, 0, texSize, texSize,GL_RGBA,GL_FLOAT,result);
    // print out results
    printf("Data before round trip:\n");
    for (int i=0; i<texSize*texSize*4; i++)
        printf("%f\n",data[i]);
    printf("Data after round trip:\n");
    for (int i=0; i<texSize*texSize*4; i++)
        printf("%f\n",result[i]);
    // clean up
    delete [] data;
    delete [] result;
    glDeleteFramebuffersEXT (1, &fb);
    glDeleteTextures (1,&tex);
    return 0;
}
```

4 GPGPU concept 2: Kernels = shaders

In this chapter, we discuss the fundamental difference in the computing model between GPUs and CPUs, and the impact on our way of thinking algorithmically and methodically. Once we have a clear understanding of the **data-parallel** paradigm GPUs subject to, programming shaders is fairly easy.

4.1 Loop-oriented CPU implementation vs. kernel-oriented data-parallel implementation

Let us first recall the problem we want to solve: $\mathbf{y} = \mathbf{y} + alpha * \mathbf{x}$. On the CPU, we typically use a single loop over all array elements like this:

```
for (int i=0; i<N; i++)
    dataY[i] = dataY[i] + alpha*dataX[i];
```

Two levels of computation are active at the same time: Outside the loop, the **loop counter** is incremented and compared with the length of our vectors, and inside the loop, we access the arrays at a fixed position which is determined by the loop counter and perform the actual computation we are interested in: a multiplication and an addition on each data element. It is important to note that the calculations performed on each data element in the vectors are **independent** of each other, for a given output position, we access distinct input memory locations and there are no data dependencies between elements in the result vector. If we had a **vector processor** that is capable of performing operations on whole vectors of length N or even N CPUs, **we would not need the loop at all!** This paradigm is commonly called **SIMD (single instruction multiple data)**. On a side note: *partial loop unrolling* is a common technique in high performance computing to allow the compiler to make better usage of the extensions available in today's CPUs like SSE or SSE2.
The core idea of GPU computing for the problem we want to tackle in this tutorial should now be clear: We **separate the outer loop from the inner calculations**. The calculations we do inside the loop are extracted into a **computational kernel**: `y_new[i] = y_old[i] + alpha * x[i]`. Be aware that the kernel is no longer a **vector expression** but conceptually a **scalar template** of the underlying math that forms a single output value from a set of input values. **For a single output element, there are no data dependencies with other output elements, and all dependencies to input elements can be described relatively.**
In our example, the array index of a given output element is identical to the array indices of the input values, or more precisely, the input positions from all arrays are identical from the point of view of an output element. Another, less trivial dependency arises from the standard 1D Finite Difference scheme: `y[i] = - x[i-1] + 2*x[i] - x[i+1]`. Informally, the corresponding kernel would be: *"Compute each value in the vector* `y` *by multiplying the the value of* `x` *by two at that position, and subtract the the value to the left and the value to the right"*.
The programmable part of the GPU we want to use in our computations, the so-called **fragment pipeline**, consists of many parallel processing units, up to 24 in the GeForce 7800GTX. The hardware and driver logic however that schedules each data item into the different pipelines is not programmable! So from a conceptual point of view, all work on the data items is performed independently, without any influence among the various "fragments in flight through the pipeline". In the previous chapter we discussed that we use textures as render targets (the end point of the pipeline) and that we store our vectors in textures. Thus, another useful analogy that is valid for our kind of computations is: The fragment pipeline behaves like a vector processor of the size of our textures. Although internally the computation is split up among the available fragment processors, we cannot control the order in which fragments are processed. All we know however is the "address", the coordinates (pixel coordinates in screen space) in the target texture where an individual data item will end up. We can therefore assume all work is done in parallel without any data interdependence. This paradigm is commonly referred to as **data-parallel** computing.
Now that we have extracted the computational kernel from our problem, we can discuss the way the programmable fragment pipeline is actually programmed. Kernels translate to shaders on the GPU, so what we have to do is to write an actual shader and include it into our implementation. In this tutorial, we discuss how to achieve this with the Cg shading language and the OpenGL shading language (GLSL) The following two subsections are therefore partly redundant, the idea is that you pick one and skip the other because advantages and disadvantages of each language and runtime are beyond the scope of this tutorial.

4.2 Creating a shader with the Cg shading language

To use shaders with Cg, we have to distinguish between the **shading language** and the **Cg runtime** which we use to prepare the shader. There are two types of shaders available, corresponding to the two programmable stages of he graphics pipeline. In this tutorial, we rely on the **fixed function pipeline** in the **vertex stage** and only program the **fragment shader**: The fragment pipeline is much better suited for the kind of computations we pursue, using the vertex stage is an advanced topic beyond the scope of this tutorial. Additionally, the fragment pipeline traditionally provides more computational horsepower.

Let us start with writing the shader code itself. Recall that the kernel on the CPU contains some arithmetics, two lookups into the data arrays and a constant floating point value. We already know that textures are the equivalent of arrays on GPUs, so we use texture lookups instead of array lookups. In graphics terms, we **sample the textures at given texture coordinates**. We will postpone the question of how correct texture coordinates are calculated automatically by the hardware until the next chapter. To deal with the constant floating point value, we have two options: We can inline the value into the shader source code and dynamically recompile the shader whenever it changes, or, more efficiently, we can pass the value as a uniform parameter. The following bit of code contrasts a very elaborate version of the kernel and the shader source:

```
                                float saxpy (
                                    float2 coords : TEXCOORD0,
                                    uniform sampler2D textureY,
                                    uniform sampler2D textureX,
                                    uniform float alpha ) : COLOR
                                {
                                    float result;
float yval=y_old[i];                float y = tex2D(textureY, coords);
float xval=x[i];                    float x = tex2D(textureX, coords);
y_new[i]=yval+alpha*xval;           result = y + alpha*x;
                                    return result;
                                }
```

Conceptually, a fragment shader like the one above is a tiny program that is executed for each fragment. In our case, the program is called `saxpy`. It receives several input parameters and returns a float. The colon syntax is called **semantics binding**: Input and output parameters are identified with various state variables of the fragment. We called this "address" in the previous section. The output value of the shader has to be bound to the **COLOR** semantics, even though this is not too intuitive in our case since we do not use colors in the traditional sense. Binding a tuple of floats (float2 in Cg syntax) to the **TEXCOORD0** semantics will cause the runtime to assign the texture coordinates that were computed in a previous stage of the graphics pipeline to each fragment. More on that later, for now, we can safely assume that we sample the textures at precisely the coordinates we want. The way we declare the textures in the parameter list and the actual lookup/sampling function depends on the **texture target** we use: For **texture2D**, the keywords are `sampler2D` and `tex2D(name,coords)`, for **texture rectangles**, we need to use `samplerRECT` and `texRECT(name,coords)` accordingly.
To use four-channeled textures instead of luminance textures, we just replace the type of the variable that holds the result of a texture lookup (and of course the return value) with **float4**. Since GPUs are able to perform all calculations on four-tuples in parallel, the shader source for texture rectangles and RGBA textures reads:

```
float4 saxpy (
    float2 coords : TEXCOORD0,
    uniform samplerRECT textureY,
    uniform samplerRECT textureX,
    uniform float alpha ) : COLOR
{
    float4 result;
    float4 y = texRECT(textureY, coords);
    float4 x = texRECT(textureX, coords);
    result = y + alpha*x;
    // equivalent: result.rgba=y.rgba+alpha*x.rgba
    //         or: result.r=y.r+alpha*x.y; result.g=...
    return result;
}
```

We store the shader source in a char array or in a file to access it from our OpenGL program through the Cg runtime.

4.3 Setting up the Cg runtime

This subsection describes how to set up the Cg runtime in an OpenGL application. First, we need to include the Cg headers (it is sufficient to include `Cg/cgGL`) and add the Cg libraries to our compiler and linker options. Then, we declare some variables:

```
// Cg vars
CGcontext cgContext;
CGprofile fragmentProfile;
CGprogram fragmentProgram;
CGparameter yParam, xParam, alphaParam;
char* program_source = "float saxpy( [....] return result; } ";
```

The `CGcontext` is the entry point for the Cg runtime, since we want to program the fragment pipeline, we need a **fragment profile** (Cg is profile-based) and a **program container** for the program we just wrote. For the sake of simplicity, we also declare three handles to the parameters we use in the shader that are not bound to any semantics, and we use a global variable that contains the shader source we just wrote. We encapsulate all the Cg runtime initialization in one function:

```
void initCG(void) {
    // set up Cg
    cgContext = cgCreateContext();
    fragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
    cgGLSetOptimalOptions(fragmentProfile);
    // create fragment program
    fragmentProgram = cgCreateProgram (
                      cgContext, CG_SOURCE, program_source,
                      fragmentProfile, "saxpy", NULL);
    // load program
    cgGLLoadProgram (fragmentProgram);
    // and get parameter handles by name
    yParam = cgGetNamedParameter (fragmentProgram, "textureY");
    xParam = cgGetNamedParameter (fragmentProgram, "textureX");
    alphaParam = cgGetNamedParameter (fragmentProgram, "alpha");
}
```

4.4 Setting up a shader for computation with the OpenGL shading language

It is not neccessary to include any additional header files or libraries to use the OpenGL Shading Language, it is built into the driver. Three OpenGL extensions (**ARB_shader_objects**, **ARB_vertex_shader** and **ARB_fragment_shader**) define the API, and the GLSL specification defines the language itself. Both API and language are part of core OpenGL 2.0, but we use the older enumerants nontheless.
We define a series of global variables for the **program object**, the **shader object** and **handles** to the data variables we want to access in the shaders. The first two objects are simply data containers managed by OpenGL, one program can consist of exactly one vertex and fragment shader, both subtypes can consist of several shader sources, a shader can in turn be part of several programs etc.:

```
// GLSL vars
GLhandleARB programObject;
GLhandleARB shaderObject;
GLint yParam, xParam, alphaParam;
```

Writing the actual shaders is similar to using the Cg shading language, so we just present two example shaders. The two main differences are probably that all variables we assign explicitly through the runtime are declared globally, and that instead of parameter bindings to GL state variables, we use a set of reserved variable names that are bound implicitely to the current GL state.

```
// shader for luminance data and texture rectangles
uniform samplerRect textureY;
uniform samplerRect textureX;
uniform float alpha;

void main(void) {
    float y = textureRect(textureY,gl_TexCoord[0].st).x;
    float x = textureRect(textureX,gl_TexCoord[0].st).x;
    gl_FragColor.x =  y + alpha*x;
}
```

```
// shader for RGBA data and texture2D
uniform sampler2D textureY;
uniform sampler2D textureX;
uniform float alpha;

void main(void) {
    vec4 y = texture2D(textureY, gl_TexCoord[0].st);
    vec4 x = texture2D(textureX, gl_TexCoord[0].st);
    gl_FragColor = y + alpha*x;
}
```

We can encapsule the initialization of the GLSL runtime again into a single function. The GLSL API is designed to mimic the traditional compilation and linking process. For details please refer to the Orange Book or take a look at the various GLSL tutorials available on the internet.

```
void initGLSL(void) {
    // create program object
    programObject = glCreateProgramObjectARB();
    // create shader object (fragment shader) and attach to program
    shaderObject = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
    glAttachObjectARB (programObject, shaderObject);
    // set source to shader object
    glShaderSourceARB(shaderObject, 1, &program_source, NULL);
    // compile
    glCompileShaderARB(shaderObject);
    // link program object together
    glLinkProgramARB(programObject);
    // Get location of the texture samplers for future use
    yParam = glGetUniformLocationARB(programObject, "textureY");
    xParam = glGetUniformLocationARB(programObject, "textureX");
    alphaParam = glGetUniformLocationARB(programObject, "alpha");
}
```

5 GPGPU concept 3: Computing = drawing

In this chapter, we will discuss how everything we learned in the previous chapters of this tutorial is pieced together to perform the scaled vector-vector addition $\mathbf{y\_new} = \mathbf{y\_old} + alpha * \mathbf{x}$. In the accompanying implementation, this is encapsulated in the `performComputation()` routine. Four steps are required: The kernel is activated, input and output arrays are assigned using the shader runtime and finally, the computation is triggered by rendering a suitable geometry. This last step is embarrassingly simple after all the foundations we laid in the previous chapters.

5.1 Preparing the computational kernel

To activate the kernel using the Cg runtime, the **fragment profile** we created previously is **enabled** and the shader we wrote and already loaded is **bound**. Only one shader can be active at the same time. More correctly, only one vertex and one fragment shader can be active at a given time, but we rely on the fixed function pipeline at the vertex stage in this tutorial, as mentioned before. This requires just two lines of code:

```
// enable fragment profile
cgGLEnableProfile(fragmentProfile);
// bind saxpy program
cgGLBindProgram(fragmentProgram);
```

Using the GLSL runtime, this is even easier: If our program object was linked successfully, all we have to do is to install the program as part of the rendering pipeline:

```
glUseProgramObjectARB(programObject);
```

5.2 Setting input arrays / textures

Using the Cg runtime, linking and enabling the input arrays/textures `y_old` and `x` and the uniform value (the right hand side of the equation, texture identifiers `y_oldTexID` and `xTexID` respectively) is straight-forward use of the Cg runtime:

```
// enable texture y_old (read-only)
cgGLSetTextureParameter(yParam, y_oldTexID);
cgGLEnableTextureParameter(yParam);
// enable texture x (read-only)
cgGLSetTextureParameter(xParam, xTexID);
cgGLEnableTextureParameter(xParam);
// enable scalar alpha
cgSetParameter1f(alphaParam, alpha);
```

In GLSL, we have to bind our textures to different texture units (the Cg runtime does this automatically for us) and pass these units to our uniform parameters:

```
// enable texture y_old (read-only)
glActiveTexture(GL_TEXTURE0);
glBindTexture(textureParameters.texTarget,yTexID[readTex]);
glUniform1iARB(yParam,0); // texunit 0
// enable texture x (read-only)
glActiveTexture(GL_TEXTURE1);
glBindTexture(textureParameters.texTarget,xTexID);
glUniform1iARB(xParam, 1); // texunit 1
// enable scalar alpha
glUniform1fARB(alphaParam,alpha);
```

5.3 Setting output arrays / textures

Defining the output array (the left side of the equation) is essentially the same operation like the one we discussed to transfer data to a texture already attached to our FBO. Simple pointer manipulation by means of GL calls is all we need. In other words, we simply redirect the output: If we did not do so yet, we attach the target texture to our FBO and use standard GL calls to use it as the render target:

```
// attach target texture to first attachment point
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT,
                          texture_target, y_newTexID, 0);
// set the texture as render target
glDrawBuffer (GL_COLOR_ATTACHMENT0_EXT);
```

5.4 Performing the computation

Let us briefly recall what we did so far. We enabled a $1:1$ mapping between the target pixels, the texture coordinates and the geometry we are about to draw. We also prepared a fragment shader we want to execute for each fragment. All that remains to be done is: Render a *suitable geometry* that ensures that our **fragment shader is executed for each data element we stored in the target texture**. In other words, we **make sure that each data item is transformed uniquely into a fragment**. Given our projection and viewport settings, this is embarrassingly easy: All we need is a **filled quad** that covers the whole viewport! We define the quad with standard OpenGL (immediate mode) rendering calls. This means we directly specify the four corner vertices of

the quad. We also assign texture coordinates as vertex attributes to the four vertices. The four vertices will be transformed to screen space by the fixed function vertex stage which we did not program. The **rasterizer**, a fixed-function part of the graphics pipeline located between the vertex and fragment stage, will then perform a bilinear interpolation for each pixel that is covered by the quad, interpolating both the position (in screen space) of the pixels and the vertex attributes (texture coordinates) from the vertex data. It generates a fragment for each pixel covered by the quad. The interpolated values will be passed automatically to the fragment shader because we used the binding semantics when we wrote the shader. In other words, rendering a simple quad serves as a **data stream generator** for the fragment program. Because of the viewport and projection settings we implemented, the whole output array / texture) is covered by the quad, and is thus transformed into a fragment automatically, including interpolated attributes such as texture coordinates. Consequently, the fragment program execution is triggered for each output position in the array: **By rendering a simple textured quad, we achieve that the kernel is executed for each data item in the original vector / texture!** This is what we wanted to achieve throughout the whole tutorial.
Using **texture rectangles** (texture coordinates are identical to pixel coordinates), we use a few code lines like this:

```
// make quad filled to hit every pixel/texel
glPolygonMode(GL_FRONT,GL_FILL);
// and render quad
glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(0.0, 0.0);
    glTexCoord2f(texSize, 0.0);
    glVertex2f(texSize, 0.0);
    glTexCoord2f(texSize, texSize);
    glVertex2f(texSize, texSize);
    glTexCoord2f(0.0, texSize);
   glVertex2f(0.0, texSize);
glEnd();
```

Using texture2Ds (and therefore normalized texture coordinates), this is equivalent to:

```
// make quad filled to hit every pixel/texel
glPolygonMode(GL_FRONT,GL_FILL);
// and render quad
glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(0.0, 0.0);
    glTexCoord2f(1.0, 0.0);
    glVertex2f(texSize, 0.0);
    glTexCoord2f(1.0, 1.0);
    glVertex2f(texSize, texSize);
    glTexCoord2f(0.0, 1.0);
    glVertex2f(0.0, texSize);
glEnd();
```

One remark for advanced programmers: In our shaders, we only use one set of texture coordinates. It is possible to define several sets of (different) texture coordinates per vertex, refer to the documentation of `glMultiTexCoord()` for details.

6 GPGPU concept 4: Feedback

After the calculation is performed, the resulting values are stored in the target texture `y_new`.

6.1 Multiple rendering passes

In a proper application, the result is typically used as input for a subsequent computation. On the GPU, this means we perform another **rendering pass** and bind different input and output textures, eventually a different kernel etc. The most important ingredient for this kind of **multipass rendering** is the **ping pong** technique.

6.2 The ping pong technique

Ping pong is a technique to alternately use the output of a given rendering pass as input in the next one. In our case, this means that we swap the role of the two textures `y_new` and `y_old`, since we do not need the values in `y_old` any more once the new values have been computed. There are three possible ways to implement this kind of data reuse (take a look at Simon Green's FBO slides[6] for additional material on this):

- Use one FBO with one attachment per texture that is rendered to, and bind a different FBO in each rendering pass using `glBindFramebufferEXT()`.

[6] `http://download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf`

- Use one FBO, reattach the render target texture in each pass using `glFramebufferTexture2DEXT()`.
- Use one FBO and multiple attachment points, switch using `glDrawBuffer()`.

Since there are up to four attachment points available per FBO and since the last approach turns out to be fastest, we will explain how to ping pong between different attachments. To do this, we first need a set of management variables:

```
// two textures identifiers referencing y_old and y_new
GLuint yTexID[2];
// ping pong management vars
int writeTex = 0;
int readTex = 1;
GLenum attachmentpoints[] = { GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT };
```

During the computation, all we need to do now is to pass the correct value from these two tupels to the corresponding OpenGL calls, and to swap the two index variables after each pass:

```
// attach two textures to FBO
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          attachmentpoints[writeTex],
                          texture_Target, yTexID[writeTex], 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          attachmentpoints[readTex],
                          texture_Target, yTexID[readTex], 0);
// enable fragment profile, bind program [...]
// enable texture x (read-only) and uniform parameter [...]
// iterate computation several times
for (int i=0; i<numIterations; i++) {
    // set render destination
    glDrawBuffer (attachmentpoints[writeTex]);
    // enable texture y_old (read-only)
    cgGLSetTextureParameter(yParam, yTexID[readTex]);
    cgGLEnableTextureParameter(yParam);
    // and render multitextured viewport-sized quad
    // swap role of the two textures (read-only source becomes
    // write-only target and the other way round):
    swap();
}
```

7 Overview of the accompanying example implementation

The accompanying example program[7] uses all concepts explained in this tutorial to perform the following operations:

- Create one floating point texture per vector.
- Transfer initial data to these textures.
- Create a shader using Cg.
- Iterate the computation several times to demonstrate the ping pong technique.
- Transfer the results back to main memory.
- Compare the results with a CPU reference solution.

Variable parts of the implementation In the code, a series of structs are used to store the possible parameters to OpenGL calls such as the enumerants from the various floating point texture extensions, the texture format, the slightly different shaders they imply etc. This is an example of such a struct for the luminance format, texture rectangles and the **NV_float_buffer** extension:

```
rect_nv_r_32.name             = "TEXRECT - float_NV - R - 32";
rect_nv_r_32.texTarget        = GL_TEXTURE_RECTANGLE_ARB;
rect_nv_r_32.texInternalFormat = GL_FLOAT_R32_NV;
rect_nv_r_32.texFormat        = GL_LUMINANCE;
rect_nv_r_32.shader_source    = "float saxpy ("\
                                "in float2 coords : TEXCOORD0,"       \
                                "uniform samplerRECT textureY,"       \
                                "uniform samplerRECT textureX,"       \
                                "uniform float alpha )  : COLOR {"    \
                                "float y = texRECT (textureY, coords);"\
                                "float x = texRECT (textureX, coords);"\
                                "return y+alpha*x; }";
```

To extract a working version for a special case, just do a search and replace, or use the second command line parameter which is just a string literal like `rect_nv_r_32`. In the application, the global variable `textureParameters` points to the struct actually used.

[7] available on my homepage

Command line parameters The program uses command line parameters for configuration. If you run the program without any parameters, it will print out an explanation of the various parameters. Be warned though that the parsing code for the command line is rather unstable (writing proper parsing code is clearly beyond the scope of this tutorial) and will probably crash if calling conventions are ignored. Please refer to the batch files included for examples.

The test mode The program can be used to test, for a given GPU and driver combination, which internal formats and texture layouts can be used together with the framebuffer object extension. There is a batch file called `run_test.bat` that calls the program with different command line parameters and generates a report file. It can in fact be used as a shell script on Linux with just minor changes.

The benchmark mode This mode is included just for fun. It times a sequence of computations and prints out MFLOP/s rates for given problem sizes. Like every benchmark, it should be taken with a grain of salt. Please refer to the `run_bench.bat` script file for examples.

A Appendix

A.1 Error checking in OpenGL

It is highly recommended to add calls to this useful function in your own codes very often while debugging.

```
void checkGLErrors(const char *label) {
    GLenum errCode;
    const GLubyte *errStr;
    if ((errCode = glGetError()) != GL_NO_ERROR) {
        errStr = gluErrorString(errCode);
        printf("OpenGL ERROR: ");
        printf((char*)errStr);
        printf("(Label: ");
        printf(label);
        printf(")\n.");
    }
}
```

A.2 Error checking with FBOs

The **EXT_framebuffer_object** extension defines a neat debugging routine which I just list here for reference. To decode the error messages, reading the section about **framebuffer completeness** in the specification is recommended.

```
bool checkFramebufferStatus() {
    GLenum status;
    status=(GLenum)glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
    switch(status) {
        case GL_FRAMEBUFFER_COMPLETE_EXT:
            return true;
        case GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT_EXT:
            printf("Framebuffer incomplete,incomplete attachment\n");
            return false;
        case GL_FRAMEBUFFER_UNSUPPORTED_EXT:
            printf("Unsupported framebuffer format\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT_EXT:
            printf("Framebuffer incomplete,missing attachment\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS_EXT:
            printf("Framebuffer incomplete,attached images
                    must have same dimensions\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_FORMATS_EXT:
            printf("Framebuffer incomplete,attached images
                     must have same format\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER_EXT:
            printf("Framebuffer incomplete,missing draw buffer\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER_EXT:
            printf("Framebuffer incomplete,missing read buffer\n");
            return false;
    }
    return false;
}
```

A.3 Error checking with Cg

Error checking with Cg is done slightly different. A self-written error handler is passed to the Cg runtime as an **error callback function**.

```
// register the error callback once the context has been created
cgSetErrorCallback(cgErrorCallback);

// callback function
void cgErrorCallback(void) {
    CGerror lastError = cgGetError();
    if(lastError) {
        printf(cgGetErrorString(lastError));
        printf(cgGetLastListing(cgContext));
    }
}
```

A.4 Error checking with GLSL

To see the results of the compilation, use this function:

```
void printInfoLog(GLhandleARB obj) {
    int infologLength = 0;
    int charsWritten  = 0;
    char *infoLog;
    glGetObjectParameterivARB(obj,
                              GL_OBJECT_INFO_LOG_LENGTH_ARB,
                              &infologLength);
    if (infologLength > 1) {
        infoLog = (char *)malloc(infologLength);
        glGetInfoLogARB(obj, infologLength,
                        &charsWritten, infoLog);
        printf(infoLog);
        printf("\n");
        free(infoLog);
    }
}
```

You can use queries like this for most states, refer to the GLSL documentation and specification for more details. Another very important query is to check if the program could be linked:

```
GLint success;
glGetObjectParameterivARB(programObject,
                          GL_OBJECT_LINK_STATUS_ARB,
                          &success);
if (!success) {
    printf("Shader could not be linked!\n");
}
```

A.5 Acknowledgements

Writing this tutorial would have been impossible without all contributors at the GPGPU. org discussion forums. They answered all my questions patiently, and without them, starting to work in the GPGPU field (and consequently, writing this tutorial) would have been impossible.