# GPGPU Q2E2004


Bent Bisballe - 20001467
deva@aasimon.org

10th January 2005

## Contents

# 1 Introduction

GPGPU stands for General-Purpose computation on GPUs. A GPU is a Graphics Processing Unit, which is the floating point pipeline processor that works on a modern graphicsadapter.
GPGPU is the possibility to use this processor for nongraphics purposes.
But why use the graphics adapter for something it was not designed to do?
The anser is strikingly clear: *Because we can!*
Most of the time this very fast processing unit is not doing anything, and why let all this good calculation power go to waste.
Furthermore, the GPU is specialized in calculating with floatingpoint values, vectors and matrices, so for these specific tasks it is many times as efficient as the CPU.

Ok, then what is all this lib thing about?

I together with Jesper Fruergaard decided quite a while back (during a raytrace introductory course) that we wanted to implement a raytracer that could run on the GPU, due to the fact that our CPU implementation [pentagon] was quite inefficient.
So it all began... and after a while ended!
We initially wrote our raytracer for the linux platform (since this was the platform we were best aquainted with), so the obvious choice was also to write the GPU based raytracer for linux.
Unfortunately our given framework, that wraps the [Cg] programming language [RenderTexture] (which is the lowlevel c language that we ant to use for the GPU programming task), didn't work too well on linux, in fact it didn't work at all, despite the authors claiming it to do. This lead us to write our own framework, namely libgpgpu.
The ideas for the library was plenty, and we slowly came to realize that this was far more than we could comprehend beside writing a raytracer, and so the project focus changed; libGPGPU was born.

## 1.1 The Problem

Cg among other things is a compiler called cgc, that can compile a c-like language into an assembly language that can be uploaded and run on the GPU. In order to use it, one has to do a lot of setup code in OpenGL, to initialize the hardware, setup modes, and lots of other tedeous things.
We wanted to create a library that was able to wrap all [OpenGL] code, and thereby hide it away, in order to leave the important code as the only thing for the eye to see.
Furthermore we wanted to make it work on every thinkable hardwarecombination, so the user not would have to think cross-platformish while coding.

Lastly we wanted to implement a series of generally used methods on the texture which is used for in- and output for the GPU.
Despite all these hideaway ideas, we still wanted Cg up front in order to preserve the power of the relatively low level access we were granted with Cg.
We decided to use an object oriented model, since the architectural entry-point for GPU programming fits nicely into this model.

# 2 Previous Work

## 2.1 Cg

Cg comes itself with a set of native c methods for uploading programs, setting parameters and what else is needed to create, load and run a program. These methods however are in c, which has a nasty habbit of being quite messy. In other words, this was not exactly matching our requrements.

## 2.2 RenderTexture

RenderTexture tries to give a clean interface by wrapping the OpenGL initialization code and the Cg handler code into an object which is initialized using a string of parameters (much like a normal program).
However it totally fails its own task, since it only offers a little corner of the functionality of Cg, which leads to lots and lots of Cg and OpenGL code in the mainprogram anyway!
As mentioned previously it also failed on another point; it didn't work on linux.

## 2.3 Brook

We have also been introduced to some highlevel languages. One of them is Brook, which try to get to a higher level of abstraction, by delivering a compiler for a standart c++ language, with kernel stream extensions. It actually compiles the Brook language into normal c++ and Cg code, which then is compiled normally into a real program.
This fits nicely into our requirements, except for the fact that the lowlevel Cg access is gone.

## 2.4 Sh

Sh is much like Brook a c++ language with extensions, that compiles into something else. As with Brook it lags the oportunity to go lowlevel, and thereby also disqualifies itself for the task.

# 3 Implementation

We wished to implement the library on the linux platform, using the g++ compiler, the OpenGL and nVidias Cg frameworks.
Also we wanted to make the library work on both ATI and nVidia graphics cards.
The latter proved to be quite hard to achieve!
At our disposition we had an ATI Radeon 9800 and an nVidia FX5200. These two cards are quite different, in both performance and precision. The Radeon card are working in 24 bits internally and the nVidia card are working in 32 bits internally. This amongst other things gives the ATI card a performance boost on calculations containing much data transfer and little calculation, compared to the nVidia card, as we will see later on.
Besides the hardware differences, the cards/drivers differed in their way of handling texture coordinates. The ATI card only supports the `Texture2D` (texture coordinates run from 0 to 1) type and the nVidia card only supports `TextureRECT` (texture coordinates run from 0 to the width/height of the texture).
We wanted to hide this difference for the user of the library, by creating macros and build in programs, for the user to call, that automatically uses the correct texture type.
Also we wanted to hide all the calls to the GL subsystem, in order to avoid setup code for the pbuffer etc., since this code often is messy looking and thereby desturbs the actual (important) code.

In order to be able to choose the correct texturetype (and other hardware specifik code) we decided to detect what hardware we were running on, on compiletime, and statically compile the algorithms into the library.

We came up with a library design that contains a single Gpu object, which is hidden from the user, that contains all pbuffer initialization code. This object is automatically generated on library loadtime (the library is a dynamic library). The Gpu object represents the graphics hardware, and therefore has to be unique. At the disposition to the user, we have 4 main classes: Kernel, FragmentProgram, VertexProgram and Texture. These classes are all the user needs to create a working gpgpu application. The Kernel class represents a calculation kernel. It is created using either a FragmentProgram or a FragmentProgram *and* a VertexProgram. When initialized, the outputtexture can be set, and a call to the run method will invoke the calculation.
The FragmentProgram and VertexProgram are both subclasses of the Program superclass, which handles setting of the parameters.
A call to getParam returns a Parameter object, which is aware of its own type and the mapping of it bewteen c++ and Cg (according to the informa-

tion it gets from the Cg compiler). It has a set method that saves the value on the CPU for now.
To create a Vertex or Fragmentprogram, plain Cg code is written, either directly into the code as a string, or read from a file, just like the Cg createProgram calls.
The program has been compiled and uploadet to the gpu when the constructor returns, i.e. if the object is successfully created, the program did not contain compile errors.
Once the program is fully loaded, the parameters can be set, and the program put into a kernel for execution.
The Texture class represents the texture, both on CPU and GPU. It has an upload and a download method, that works like a bridge between the CPU and the GPU. When used as a parameter for a program, it does not move data across this bridge, since there is no need to. In other words it is efficient to move around with the Texture object, but quite inefficient to upload to it, and download from it.
Apart from being a normal datatype (like float and float4) it has some buildin programs that can be run on it. We made these to make the life easier for the user, by handling a lot of trivial functionality, that are quite tedeous to program. First of all we have made a generel reduce method, that takes a fragmentprogram as its parameter, and runs it on the entire texture, reducing blocks of 4 pixels to a single pixel recursively until a texture of 1x1 is reached and thus returned as a float4 value. On top of this we created four reduce methods: max, min, avg and sum. Beside this we made a show method, that simply shows the texture streched out over the entire output window. This was thought of as a debug method, and are therefore implemented prioritizing accuracy over speed.
Here we had quite a few problems with the nVidia card, since it did not seem to be able to show a 32 bit float texture (It simply turned out white). We solved this by making a copy program, that first copies the 32 bit float texture into an 8 bit texture, and thereby shows *it* in the window. Unfortunately this is much lesser efficient than on the ATI cards, where we can render the texture directly into the output window.

Here is a simplistic application, that illustrates the connection between the classes, it is not a complete application, since the textures need initial data.

```
int main() {
   FragmentProgram fp("mycgprograms.cg", "fagment_program");
   VertexProgram vp("mycgprograms.cg", "vertex_program");

   Kernel kernel(&fp, &vp);

   Texture tex_in(512, 512);
```

```
    Texture tex_out(512, 512);
    fp.getParam("texture")->set(&tex_in);
    kernel.setOutput(&tex_out);

    kernel.run();
    tex_out.show();
}
```

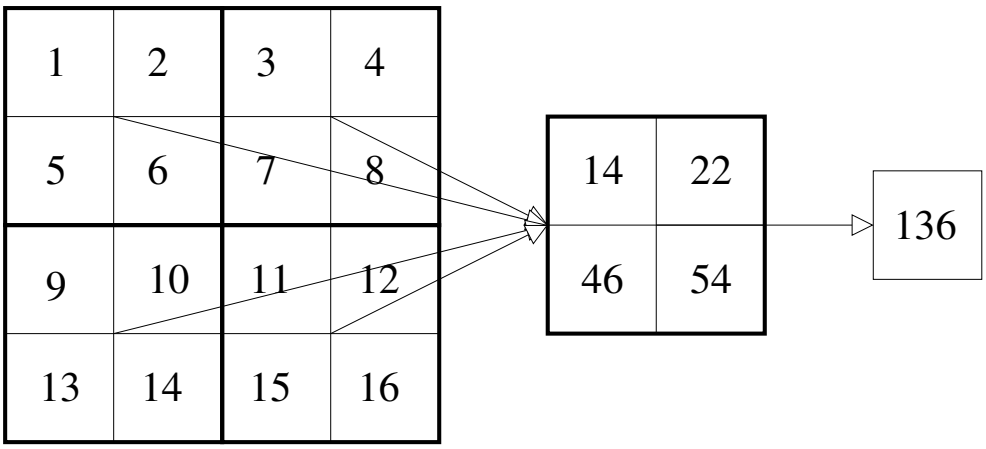
Can it be much simpler?

### 3.1 Problems and solutions

First of all we had to figure out how to initialize the pbuffer. At first we created one pbuffer pr. texture, but this seemed hopelessly inefficient. Then we came up with the idea to create just one, with a fixed size of ENOUGH (currently 1024x1024) and then just use a part of it using either scissor or by creating a quad sized not to cover the entire pbuffer, but only an area mathcing the output texture. We tested both, and found them equally efficient, and randomly chose the latter.
We have testet other sizes on the pbuffer and found out that the pbuffer is a little faster to render, if it matches the size of the outputtexture exactly (the GPU does not do nothing on a lot of pixels), but compared to the cost of having to switch context and use up more memory on the grachics card, we thought the two methods equally bad, and thereby chose the first one since it was the easiest to implement.
As mentioned previously we had a lot of trouble differentiating between the two texture types. The `Texture2D` maps the texture to coordinates running from 0 to 1, and `TextureRECT` mappes to 0 to the texture size. To make it work independent of this, we decided to use a predefined Vertex program that could create variables containig the correct coordinates, so the user wouldn't have to think about calculating them. This however we never got entirely to work.

### 3.2 Theory

In order to make the reduce method work, we had to make it parallellable, i.e. we had to make it work in independent sub calculations that did not have any influence on the other calculations.
To achieve this we used a recursive reduce method, illustrated by the figure below:




As shown the algorithm works by reducing four entries into one, until there is only one left.
This says itself that the initial input has to be a $n \times m$ matrix, where $n = m$ and $n$ is a power of 2.
If $n \neq m$ a different tactic must be used. Then several different algorithms must be run, in order to reduce different shapes into a single pixel, and thereby still shrinking the entire matrix.

To implement it on the GPU the obvious way is to use textures in smaller and smaller sizes. But since it never uses more than log$n$ textures and their size are divided by 4 in every step, it is not such a bottleneck after all.
we did not have to adress the problem with the nonsquare textures, since they are only supported on nVidia hardware, and due to the fact that we wanted to be platform independent, we simply decided not to go further into that.
However an alternative solution could have been found, using a square texture and only use part of it for data storage.

## 4 Results

We have run two test programs on three platforms. The platforms are decsripbed as follows:

| MACHINE: [deva] | MACHINE: [zzzz] | MACHINE: [glock] |
|---|---|---|
| Intel Celeron 2.6GHz | Intel Pentium 3 900MHz | Intel Pentium 4 3GHz |
| nVidia GeForce FX5200 | nVidia GeForce FX5200 | ATI Radon 9800 |
| AGP rate: 4X | AGP rate: 2X | AGP rate: 4X |
| 512 MB RAM | 256 MB RAM | 1024 MB RAM |
| Gentoo Linux | Gentoo Linux | Fedora Core 2 |

The first of two programs are made in such a way, that it benchmarks on little calculation per fragment on much data input. The other one is created

to the opposite, much calculation on little input. The two program are as follows:

**reduce avg** calculates the avarage on each coordinate over a texture, using the buildin reduce function.

```
float4 main(float2 coord0 : TEXCOORD0,
            float2 coord1 : TEXCOORD1,
            float2 coord2 : TEXCOORD2,
            float2 coord3 : TEXCOORD3,
            uniform sampler texture) : COLOR {
  float4 result;

  float4 data00 = f4tex(texture, coord0);
  float4 data10 = f4tex(texture, coord1);
  float4 data01 = f4tex(texture, coord2);
  float4 data11 = f4tex(texture, coord3);

  result = (data00 + data10 + data01 + data11) / 4.0f;
  return result;
}
```

The nVidia version is slightly changed in order to make it fit the `TextureRECT` coordinates.
Both programs have a preceding vertex program, that set coordinates `coord0`, `coord1`, `coord2` and `coord3`, according to the hardware.

**crush** does nothing useful. It just applies a lot of math to some random numbers in a way we find hard for the hardware to optimize away.

```
float4 main(float2 coords : TEXCOORD0,
            uniform sampler texture) : COLOR {
  float4 result = float4(0,0,0,0);

  float4 d = f4tex(texture, coords);

  result += d * d.x; result -= d * d.y;
  result *= d * d.z; result /= d * d.w; result *= d * d;

  result += d * d.x; result -= d * d.y;
  result *= d * d.z; result /= d * d.w; result *= d * d;

  result += d * d.x; result -= d * d.y;
  result *= d * d.z; result /= d * d.w; result *= d * d;
```

```
  result += d * d.x; result -= d * d.y;
  result *= d * d.z; result /= d * d.w; result *= d * d;

  result += d * d.x; result -= d * d.y;
  result *= d * d.z; result /= d * d.w; result *= d * d;

  return result;
}
```

We got the following results running *reduce avg* on all platforms:

```
[deva]
Using GPU 0.49947798, 0.50036418, 0.50022524, 0.49992153, time 134.1162ms
Using CPU 0.49947670, 0.50036245, 0.50022602, 0.49992740, time  38.0926ms

[zzzz]
Using GPU 0.49947798, 0.50036418, 0.50022524, 0.49992153, time 233.6390ms
Using CPU 0.49947670, 0.50036245, 0.50022602, 0.49992740, time 328.7903ms

[glock]
Using GPU 0.49944305, 0.50032043, 0.50018310, 0.49988174, time  52.3538ms
Using CPU 0.49948033, 0.50035828, 0.50021737, 0.49992510, time  16.6301ms
```

The difference in the CPU results are due to different versions og glibc, and thereby different random values in the initial buffer.

When running the *crush* program on all platforms we got the following results:

```
[deva]
Using GPU time    0.20680ms
Using CPU time  122.82924ms
Upload           26.98222ms
Download         71.53818ms

[zzzz]
Using GPU time    0.27475ms
Using CPU time  342.81571ms
Upload           79.16436ms
Download        349.50496ms

[glock]
Using GPU time    4.23691ms
Using CPU time   96.10490ms
Upload           24.59458ms
Download         14.31307ms
```

The Upload and Download times are the time used to upload and download a single texture.

# 5 Discussion

Before starting to analyse the test results, it is worth a look on the differences between the three architectures:

1. Programs are identical to a certain extend, they differ in texture types.
2. Hardware underlay are different. The CPU can have impact on GPU performance.
3. Bit precission is different. nVidia is better than ATI, and therefore possibly slower.
4. Show texture is implemented very different on the two architectures. The nVidia card uses an extra copy algorithm to convert from 32 bit to 8 bit.
5. nVidia has created Cg ! The CG_PROFILE_VP30 and CG_PROFILE_FP30 profile works only on nvidia cards, since it is an nVidia stadard. When compiling to other cards it uses the CG_PROFILE_ARBFP1 and CG_PROFILE_ARBVP1 profiles, whitch is possibly slower.
6. The AGP bus speed varies, which might impact on texture up/download.
7. The number of pipelines are not the same, Radeon 9800 has 8, nVidia FX5200 has 4.

Lets first have a look at the **reduce avg** algorithm.
It would be expected for the ATI card to outperform the nVidia card, due to the number of pipelines, and the smaller data amount it has to move (the 24 bits instead of 32).
Furthermore it is expected for `[deva]` to perform a little bit better than `[zzzz]` due to faster bus speeds.
When it comes to the CPUs `[glock]` should be faster than `[deva]`, which again should outperform `[zzzz]`, and they should all be faster than their GPUs according to [Kruger et al] that points out that the GPU works bad on large amounts of data, which is used for small amounts of calculations.

As to the CPU predictions they hold, and also the predictions on the GPU timings. Notice the difference in the floatingpoint precision.
We see a strange reading on the CPU speed of `[zzzz]` compared to its GPU, which actually outperforms it. What causes this is unknown.

Now lets look at the **crush** algortihm.
It is a bit more tricky, since we don't know the limits between the data bottleneck mentioned in [Kruger et al], but as pointed out previously, we designed the program in an attempt to exceed that limit.
The expectations on the GPUs in relation to each other are the same as in **reduce avg**, and the same goes to the CPUs. When comparing the respective GPUs to their CPUs, the GPU is expected to outperform its CPU.
In this program we also measure on texture up/downloads. Which are expected to be faster up than down on all platforms, and generally faster on `[glock]` than on `[deva]`, which again is expected to be faster than `[zzzz]`, all due to differences in AGP bus speed (`[glock]`: AGPx8, `[deva]`: AGPx4 and `[zzzz]`: AGPx2)

Again the CPU predictions hold. The texture up/download predictions hold also, though the very high value of the texture download time on `[zzzz]`, is a bit odd.
But then something surprising happens; the GPUs. This is an example of the hardware being very different ideed, and very unpredictive, when it comes to timings. The two nVidia cards perform relatively to each other, as expected, but the ATI card seems very slow (or the nVidia cards seem very fast). Many theories has been formed up to this point, but due to lack of knowledge of the underlying hardware (both nVidia and ATI are very keen on keeping their hardware specifications a secret), it is impossible to say anything reasonable about this behavior.

### 5.1 Conclusion

The differences on the hardware made it much more difficult than we first expected, but despite that, we got most of it working much like we hoped.
A few unpredicted problems occurred, but were solved pretty easy (the problem with the nVidia card unable to show 32 bit float textures).
All in all I'll say we have succeeded, despite the fact that our ToDo list is still long, and continues to grow. But it is worth mentioning that this is a project in growth, and we never expected all our ideas to be in this version anyway.

When it comes to Cg we found a couple of things that were undocumented features/bugs (for instance constants in vertexprogram magically disappear, and the min/max functions are said to work with all float vector sizes, but we found them only working up to float2), but since it is a beta version (1.3 beta) we find this acceptable, and look forward to the next stable version.

A few words about GPGPU in general; The GPU is not nessecarily faster than the CPU, but in any case, it is an extra wheel, that can help offload

the CPU, so GPGPU does always pay off (unless ofcause it costs more to up/download the data than doing the calculation on the CPU alone).

We found quite a few problem due to lacking drivers, but this will probably be fixed over time (Render to texture would be nice).

I have absolutely no doubts that there is a future for GPGPU!

## 5.2 Future Work

As of now we have not yet implemented all the functionality of which we originally planned. This lag is completely due to lag of time (Say I think we heard that one before).
Here is a list of these features, that might be included in a future version of libgpgpu.

1. Texture operator overloads.
2. More texture reduce operations.
3. Support for non square textures in reduce.
4. Optimized pbuffer rendering.
5. A full type system.
6. Better control over Texture2D and TextureRECT.
7. Cg library functions.
8. More Cg macros.

### Texture operator overloads

We originally planned for at complete texture type, with +, -, *, / and = overloadet, in order to be able to write stuff like

```
Texture a, b, c;
a = b + c;
```

and the likes. This is made possible by the C++ operator overload functionality and should thereby be fairly simple to implement.

### More texture reduce operations

As of now we only support four reduce methods (excluding the general one): sum, average, minimum and maximum. These four were implemented in order to test our general reduce method. More reduce methods are therefore quite easy to implement, but due to lack of ideas it just came down to these four.

### Support for non square textures in reduce

Currently only square textures can be reduced (i.e. the extra feature of TextureRECT of being non square is not supported). This should be fixed, using more reduce methods to first making the input texture square, and then procede using the trivial algorithm, as described in section 3.2.

### Optimized pbuffer rendering

We currently use a fixed size pbuffer for rendering all sizes of textures. This is not in any way optimal, since a lot of time is spend by the GPU calculating on pixels that are never stored in the output texture. An alternative solution would be to use several pbuffers with sizes matching the output textures, but this requires to much of memory on the graphics adapter (Using multiple texture sizes requires memory for both the pbuffer and the texture). An alternative solution would be to ask the GPU to ignore the area not covered by the output texture, either using the stencilbuffer or the depthbuffer.

### A full type system

As of now we only support 4 types: Texture2D, TextureRECT, float1 (which is the same as float) and float2. The Parameter object is laid out in a way, so the implementation of the rest of the types should be trivial. The only reason for us not to do this, was that we would like to spend our time on other parts of the library rather than implement types that we would not use our selves in our testprograms.

### Better control over Texture2D and TextureRECT

The differentiation between Texture2D and TextureRECT is currently hardware determined (on compiletime) since the cards we were working with (nVidia GeForce FX 5200 and ATI Radeon 9800) are too old to work with other than their native texture type (ATI uses Tex2D and nVidia uses TexRECT). But it has come to our knowledge that existing newer hardware supports multiple texturetypes in all formats, which will force us not to make the texturetype choice on compiletime, but rather leave it to the user which texture type should be used.

### Cg library functions

We have spend a lot of time discussing how we could implement Cg library functions, by which we mean predefined FragmentPrograms and VertexPrograms that are globally declared in the gpu namespace. These should be lacy evaluated, so that they would not take up space (and time) on the graphics adapter prior to use. Examples of these library functions could be a VertexProgram that multiplies all texture coordinates by 2, a FragmentProgram

that copies from one texture to another, a VertexProgram that inverts the coordinates and so on.

### More Cg macros

We have prepended some macros to the read Cg programs before sending them to the compiler. Currently these are wrappers to `f4tex2D`, `f4texRECT` (macro: `f4tex`) and `sampler2D`, `samplerRECT` (macro: `sampler`), using the one matching the texture type. More macros should be added to ease cross adapter development.

## 6 How To Run

Get the source at [source code], compile it running `make` in the `src/lib` folder. Then copy the lib file to someplace for the dynamic linker to find it. Now compile the test programs by running `make` in the `src/test` folder.
In order to run and compile the software, a working installation of Cg, OpenGL, and GLX is required.
Further info on how to use the library can be found on our website [webpage].

## References


[Cg] The Cg 1.2.1 manual:

`ftp://download.nvidia.com/developer/cg/Cg_1.2.1/Docs/Cg_Toolkit.pdf`

[OpenGL] The OpenGL 1.5 Reference Manual:

`http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf`

[RenderTexture] RenderTexture:

`http://sourceforge.net/projects/gpgpu`

[pentagon] The Pentagon raytracer homepage:

`http://www.aasimon.org/pentagon`

[Kruger et al] Linear algebra Operators for GPU implementation of Numerical Algorithms, Kruger J. and Westermann R.

[webpage] libGPGPU webpage:

`http://www.aasimon.org/libgpgpu`

[documentation] libGPGPU API documentation (doxygen):

`http://www.aasimon.org/libgpgpu/api`

[source code] libGPGPU sourcecode (including examples):

`http://www.aasimon.org/libgpgpu/libgpgpu-0.0.3.tar.gz`

[paper] This paper:

`http://www.aasimon.org/libgpgpu/libgpgpu-paper.pdf`

[paper source] The LaTeX2e source to this paper:

`http://www.aasimon.org/libgpgpu/libgpgpu-paper.tar.gz`