INF-3981

DIPLOMA THESIS IN COMPUTER SCIENCE

Utilizing Ubiquitous Commodity Graphics Hardware for Scientific Computing

Bjørn-Hugo Nilsen

October 2007

FACULTY OF SCIENCE
Department of Computer Science
University of Tromsø

Abstract

Current GPUs have many times the memory bandwidth and computing power compared to CPUs. The difference in performance is getting bigger as the evolution speed of the GPUs is higher than of the CPUs. This make it interesting to use the GPU for general purpose computing (GPGPU). I begin by looking at the architecture of the GPU, and some different techniques for programming on a GPU, including some of the available high-level languages. I have implemented the Mandelbrot computation on a cluster of GPUs (the HPDC display wall), and compared it against two different CPU implementations on the cluster. I have also implemented the Mandelbrot computation in both Cg and Brook, and compared the performance of the two languages. My experimental study shows that the GPU implementation of the Mandelbrot application is up to twice as fast as the load-balanced CPU implementation on the cluster of 28 computers, and up to 6 times faster on one computer.

Contents

Chapter 1

Introduction

GPUs are specifically designed to be extremely fast at processing large graphics data sets. Current GPUs have many times the memory bandwidth and computing power as the current CPUs as figure 1.1 on the following page show. This , combined with the fact that modern GPUs are fully programmable, has lead to much research on utilizing the GPU for non-graphics computation, known as GPGPU (General Purpose computation on GPU). The research is mainly driven by to considerations:

Price/Performance Ratio: The latest GPUs are extremely fast compared to the CPUs. The Nvidia GeForce 8800 Ultra has 128 stream processors and a memory bandwidth of 103.7 GB/sec. Compare to the Intel Core 2 Extreme with 4 cores and a memory bandwidth of 21.34 GB/sec with FSB speed of 1333 MT/s using DDR3 memory. The high memory bandwidth of the GPU gives it an advantage over the CPU, which has a major bottleneck in memory bandwidth. The multi-billion dollar game industry makes sure that graphics cards are sold in high volumes, which keeps the prices low compared to specialty hardware.

Evolution Speed: The speed of the GPU is doubling every six months [15] since the mid-1990s, while the speed of CPUs are still following Moore's law, and doubling every 18 month, and this trend is expected to continue. This increase in speed is made possible by the highly parallel architecture of the GPU, which make the GPU able to utilize more transistors for computation than CPUs by increasing the numbers of pipelines.

It is done a lot of research on using GPUs for general computation on one computer, but there are hardly any work available on GPGPU on a cluster of GPUs. The main idea here is to study how we can use a cluster of GPUs for scientific computation, and how the performance of a GPU cluster is compared to a cluster of CPUs.

Main Results I have utilized the high-end graphic hardware on the display wall for general purpose computation. I have implemented a GPGPU version of the Mandelbrot computation, and compared it against two different CPU implementations on the display wall. I have also compared the performance of the high-end Quadro graphic adapter on the display wall with the newer GeForce 8800 card. The main results are:

- The performance of the GPGPU implementation of the Mandelbrot set versus the CPU implementation depends on the given parameters of the Mandelbrot set. For some parameters the GPU is faster, and for other the CPU is faster, as explained in subsection 4.4.1 on page 27. The GPU is much faster than the CPU when we can

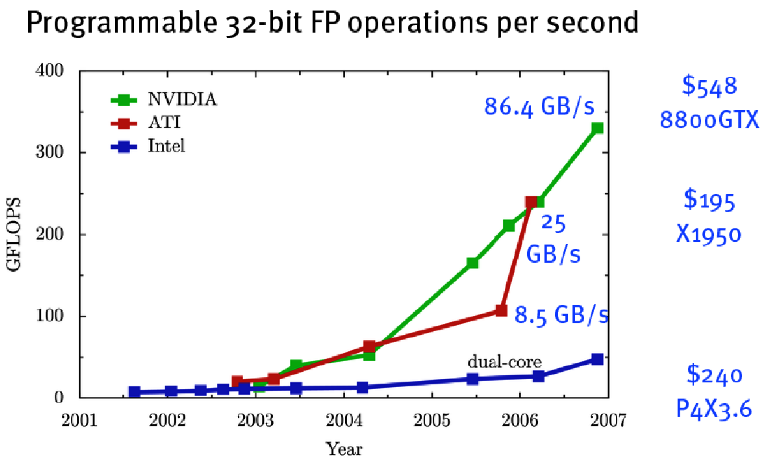


Figure 1.1: Increase of GPU performance over CPU. Graph from Owens [14]

utilize the parallelism of the GPU pipeline. The test show that the Mandelbrot on GPU cluster is up to twice as fast as the load-balanced CPU.

- The Mandelbrot set is not suited for load-balancing on the GPGPU implementation (section 4.3 on page 24).
- On one computer, the GeForce 8800 card is about 10 times faster than the 2 year older high-end Quadro 3400 on the Mandelbrot computation, and up to 6 times faster than the Pentium 4 3.2Ghz CPU.

I look at the architecture and give some examples on how to program on the GPU in chapter 2, And in chapter 3, I give an overview of some of the available languages for GPGPU programming, and take a closer look at the two languages I have used in my work; Cg and Brook.

Chapter 2

GPGPU Programming

2.1 Introduction

The GPU has become an extremely powerful processor. The speed of the GPUs are increasing faster than CPUs and is doubling every six months. The rapid increase of the speed of the GPUs , combined with the relative low cost, make them interesting to use for general computation. The GPUs give more computational power for the dollar than the CPUs.

GPUs are specialized processors, optimized for floating point operations, which make them very fast compared to CPUs:

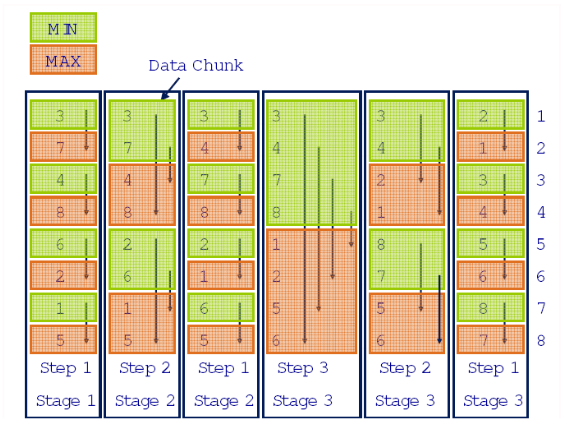
- 3.0 GHz dual-core Pentium 4: 24.6 GFLOPs
- NVIDIA GeForceFX 7800: 165 GFLOPs
- NVIDIA GeForceFX 8800: 518 GFLOPs

One of the reasons that the GPUs are increasing in speed so much faster than the CPUs are the different in the architecture of the processors. CPUs are general purpose processors and is optimized for high performance on sequential code, so they use a lot of transistors for non-computational tasks like branch prediction and caching. The GPUs are specialized on floating point operations, and have a parallel architecture. This allow the GPU to use more transistors for computation, which give the GPUs more computational power than the CPUs with the same number of transistors.

The GPU is very specialized and will never be a general purpose processor like the CPU. The GPU lack native support for integers and operands like bit-wise operations and shift operators. The GPU also lack precision for some tasks. The current GPUs support 32-bit floating points which will not be applicable to many problems. Despite these shortcomings of the GPU, there are a lot of computational tasks that can be solved successfully by the use of GPGPU.

The goal of GPGPU programming is to utilize the high power of the GPU to solve general problems faster for fewer dollars than specialized hardware.

Related Work There is done a lot of work on utilizing the GPU for general purpose computation. Govindaraju et al. [10] uses the GPU as a co-processor to sort billion record wide-key databases. Owens et al. [15] describes the techniques used in mapping general purpose computation to graphics hardware and survey the latest development in GPGPU programming. Other work on GPGPU include general linear equation solver [8], matrix



Bitonic sorting network on 8 data value. At each stage, the data is divided into sorted data chunk. The data is compared and swapped as indicated by the arrows. Note that at stage one, the data is not sorted, and the size of the sorted data chunk is only one data element.

Figure 2.1: A Bitonic sorting network. Figure from [10]

multiplication [11], parallel flow simulation using the lattice Boltzmann model [5], and cloud dynamics simulations [12].

2.1.1 Sorting

Sorting algorithms are important in computer science, and has been well studied. Purcell et al. [17] uses a bitonic sort where each stage of the algorithm is performed as one rendering pass. Govindaraju et al.[10] present the GPUTeraSort algorithm which uses the GPU as co-processor to sort databases with billion of records.

Many sorting algorithms require the ability to write to arbitrary locations. This requirement make them impossible to implement on GPUs. Recall that the GPU fragment processor is capable of doing gather but not scatter. The Bitonic merge sort has a known output and fits the GPU architecture.

The Bitonic merge sort is a parallel algorithm that sort bitonic sequences in multiple steps. A bitonic sequence is a monotonic ascending or descending sequence. Details on the Bitonic sort algorithm can be found in [3, Chapter 27]. The different sequences at each step is sorted in parallel on the GPU. In each step the values in one chunk of the input texture is compared to the values of another chunk in the input texture.

2.2 GPU Architecture

The GPU architecture is highly parallel. The Quadro FX3400 card has 32 fragment processors running in parallel, and the newest card from Nvidia, the GeForce 8800, has up to 128 fragment processors running in parallel. Compared to today's CPUs which can run up till four (quad core) tasks in parallel, the GPU has an huge advantage in performance.

The GPU pipeline (figure 2.2 on the next page) has two programmable processors; the vertex processor and the fragment processor. The fragment processor is the most interesting for GPGPU applications. There are usually more fragment operations than vertex operation in a graphics task, so the fragment processor have more computational power than the vertex processor (more pipelines). The fragment processor also has direct output to memory, and can read data from other fragments.

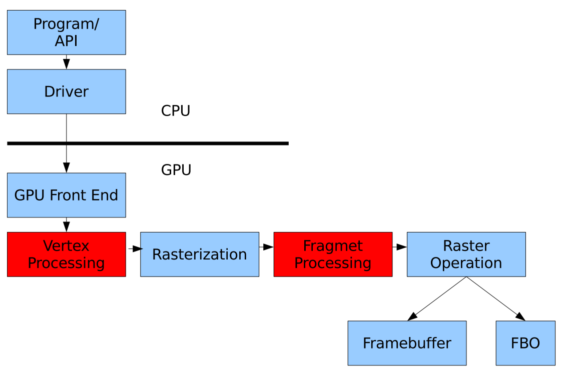


Figure 2.2: A simplified model of the GPU pipeline

API: OpenGL or DirectX

Driver: The graphic card driver. Usually a company secret.

GPU Front End: Receives commands and data from driver.

Vertex Processor: User programmable. Change position of input vertices

Rasterizer: Not user programmable. Generate fragment stream

Fragment Processor: User Programmable. Final processing on fragments. The most used processor in GPGPU programming.

Raster Operation: Depth checking and Blending.

Data on the GPU is stored in textures. The textures is similar to arrays on the CPU. The data in the textures is called fragments, which you can think of as pixels for now. A fragment is actually only a potential pixel, as not all fragments become pixels. The vertices coordinates can be changed in the vertex processor, but in GPGPU programs, they usually pass unchanged through the vertex processor. The rasterizer make fragments from the vertices position, and these fragments are processed by the fragment processor.

The last step of the pipeline is either the frame-buffer or a Frame Buffer Object (FBO). If the processed data is to be showed on the screen, the output is written to the frame-buffer. If the data is to be used as an input for the next step, it is written to on off-screen buffer (FBO).

There is no read-modify-write memory on the GPU. The memory is either read-only or write-only. The fragment processor can only write to a fixed output which is the current pixel being processed, but it can read from an arbitrary pixel (memory location). In other word, the fragment processor can do *gather*. The vertex processor on the other hand can only do *scatter*. It can only read from the given input, but it can write to an arbitrary memory location, as the purpose of the vertex processor is to move the vertex coordinates.

The data type supported by the GPU depends on which graphic card you have. Today's GPUs typically support 16- and 32-bit fixed and floating point numbers. High-level shader languages like Cg support integer types. The integers are however represented by floating point numbers in hardware, as the GPUs do not have native support for integer types.

2.3 Writing a program for GPU

2.3.1 GPU–CPU Analogies

Programming on a GPU is quite different from programming on a CPU. In this section I will draw some analogies between CPU and GPU programming. Think of the data on the GPU as a stream of data. The fragment program is executed on all elements of the stream. This is similar to SIMD parallel programming.

The textures on the GPU is similar to arrays on the CPU. As we usually use the fragment processor for GPGPU programming, we can use textures where we would use arrays on the CPU.

We can write to an off-screen texture instead of the frame-buffer. This is known as render-to-texture. The output written to an off-screen texture, also known as Frame Buffer Object (FBO), in one step of the computation, can be used as an input to the computation in the next step. So, render-to-texture can be used as a feedback in a “loop”.

To sum up the CPU–GPU analogies, here is a list of the analogies:

- Textures = Arrays
- Fragment Program = “Inner Loops”
- Render-To-Texture = Feedback
- Draw Geometry = Invoke Computation

2.3.2 Writing code

We can program the GPU by writing a shader program for the vertex processor or the fragment processor. The shader can be written in one of the many available languages like Cg, GLSL, etc. For more details on shader languages and high level GPGPU languages, see Chapter 3 on page 15. I have written the shaders in Cg and will also use Cg in the examples. In order to use Cg on the Display Wall, you will have to install the Cg Toolkit.

There are a lot of tutorials available on how to write a GPGPU program. The Basic Math Tutorial by Dominic Göddeke [9] is an excellent one, and explain in details how to write the program in both Cg and GLSL. I will go through a simple example on how to write a GPGPU program. The code examples are from my first version of the Mandelbrot program. This program just computed one image of the Mandelbrot set and read it back to the CPU.

To create a GPGPU program, we have to follow these steps:

1. Set up an OpenGL context to get access to the graphic driver
2. Preparing OpenGL for off-screen rendering
3. Setup initial data on the CPU
4. Create textures on the GPU
5. Set up textures as rendering target
6. Transfer data from CPU to GPU
7. Load the shader
8. Perform computation on the GPU
9. Read back data from the GPU to the CPU

Set up OpenGL In order to get access to the graphic hardware, we will need a valid OpenGL context. This will allow us access to the graphic hardware through the OpenGL API. In this case we can just use GLUT to set up the context. In the later versions of the Mandelbrot program, I was drawing the image to screen by letting the last output go directly to the framebuffer. For these versions I used SDL as the window handler, and the SDL context served the purpose of accessing the graphic hardware.

```
        glutInit ( &argc, argv );
        glutWindowHandle = glutCreateWindow("Mandelbrot");
```

In order to read back data written at the end of the GPU pipeline, we write it to an off-screen buffer. To change the framebuffer target between off-screen and on-screen buffer, we only need one line of code.

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
```

The first line of code will set the rendering target to an FBO. The second line restore the framebuffer as the rendering target, by binding the framebuffer extension to zero.

Initial input data The native data layout on the GPU is two-dimensional arrays. One- and three-dimensional arrays are supported, but I have not looked into the use of these. Two-dimensional arrays give the best performance as they are the native layout, and the indices can be used directly whitout any mapping. If possible, we should try to adjust our data to an two-dimensional array layout.

In order to transfer data to the GPU, we must first store it in the memory of the CPU. We have to allocate memory for the arrays on the CPU, and fill them with initial data, even if the computation is performed on the GPU.

Create And Set Up Textures Before creating textures, we have to decide which type of textures we want to use. I have used `GL_TEXTURE_RECTANGLE_ARB` which means the dimensions are not constrained to power of two, and the coordinates are not normalized.

The next decision to make is the format of the texture. We can have either a single floating point value per texel, or a four-tuple of floating points values in each texel. I have used the four-tuple floating point value (`GL_RGBA32F_ARB`) for the Mandelbrot computation, since I use the four floating point values to hold the different variables in the computation.

The next step is to map the data to the two-dimensional layout on the GPU texture. This is not a problem for the Mandelbrot set, since it is an two-dimensional image and map directly to the texture format. For other data set you will have to choose a way to map the data on the CPU to the textures. If you have an one-dimensional array of size N, and `RGBA` format on the texture, you could map it to a texture of size `sqrt(N/4)` by `sqrt(N/4)`. This will require N to be of a size that fits this mapping, but we can always pad out the array on the CPU to fit.

We also have to set up a mapping from array index to texture coordinates. We just set up an one-to-one mapping here:

```
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, width, 0.0, height);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glViewport(0, 0, width, height);
```

Transfer Data Between CPU and GPU In order to transfer data between the CPU and the GPU, we have to bind the texture to a texture target. This is done with only one line of code; `glBindTexture(texTarget, texid)`.

```
void transferToTexture (float* data, GLuint texID) {
        glBindTexture(textureParameters.texTarget, texID);
        glTexSubImage2D(textureParameters.texTarget,0,0,0,width,
                        height,textureParameters.texFormat,GL_FLOAT,data);
}
```

The three zeros in `glTexSubImage2D` are the offset and the mipmap level. We pass the whole array and do not use mipmap, so we can just leave them at zero. The width and height are the width and height of the texture. The data is a pointer to an array of float.

To read back the data from the GPU, we can use a function like this:

```
void transferFromTexture(float* data) {
        glReadBuffer(attachmentpoints[readTex]);
        glReadPixels(0, 0, width,
                    height,textureParameters.texFormat,GL_FLOAT,data);
}
```

Perform Computation on GPU Now we can load the shader program and start the computation on the GPU. The Cg shader is loaded by the call to `cgGLBindProgram-(shader_program)`. At this point we also have to set the input and output texture parameters, and other parameters passed to the function, if any. When the shader program is ready to run, we just use GL calls to draw a full screen quad in order to start the computation. The code to render a full screen quad is listed below.

```
        glBegin(GL_QUADS);
        glTexCoord2f(0.0, 0.0);
        glVertex2f(0.0, 0.0);
        glTexCoord2f(width, 0.0);
        glVertex2f(width, 0.0);
        glTexCoord2f(width, height);
        glVertex2f(width, height);
        glTexCoord2f(0.0, height);
        glVertex2f(0.0, height);
        glEnd();
```

For normalized textures, the width and height would be replaced with 1.

Note that the Cg code is read at runtime by the Cg-runtime. The code can be read from a separate file, or as a character sting within the C/C++ code. I will strongly suggest to use a separate file for the Cg code, as it make it easier to change the code. You can for instance comment out one line of code if you like, which would be hard to do if the whole code is in on character string.

The Ping Pong Technique The technique of using the output from one rendering pass as input to the next pass is often referred to as The Ping Pong technique. In this technique we use one FBO with multiple attachment points (two attachment points in the Mandelbrot implementation). We set up the management variables and swap the index variables for each pass as seen in the code below.

```
int writeTex = 0;
int readTex = 0;
GLenum attachmentpoints[] = {GL_COLOR_ATTACHMENT0_EXT,
                             GL_COLOR_ATTACHMENT1_EXT };
```

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, attachmentpoints[writeTex],
                          texTarget, TexID[writeTex], 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, attachmentpoints[readTex],
                          texTarget, TexID[readTex], 0);


for (int i=0; i<maxitr; i++) {
    glDrawBuffer (attachmentpoints[writeTex]);
    cgGLSetTextureParameter(mandelParamC, TexID[readTex]);
    cgGLEnableTextureParameter(mandelParamC);
    render_quad();
    swap();
}

void swap(void) {
    readTex = writeTex;
    writeTex = !readTex;
}
```

2.3.3 Advices and Pitfalls

Indices on the textures are not equal to the indices on a CPU array, even if the indices are not normalized. Normalized indices are in the range of 0-1. You might thing that the index on a not-normalized texture is in the range `0-(n-1)`, but that is not the case. In most cases this would not make any difference. In the case of the Mandelbrot computation however, it makes a different if the data is initialized on the CPU. The initial data in the Mandelbrot set is computed from the indices of the pixels. This means that the initial data must be computed on the GPU, when the computation is done on the GPU.

Changes to the code in order to improve performance might not always have the desired effect. The effect of the changes will depends on many things like the driver, texture type, functions used etc. You might not always get the expected result when trying to optimize the code. It is always a good idea to measure the performance before and after a change in the code.

There is a limit on the number of instructions in the shader programs. On the GeForce 6 series, the maximum number of instructions are 65,535. Loops will be unrolled, so if there is any loops in the shader program, the number of instructions will depend on the length of the loop. There is also a limitation on how long the operating system will wait on the shader to finish. This means that long shader programs working on large textures may time out.

If you want to time your GPGPU program, you should call `glFinish()` just before you start and stop the timer. If you want to time just one or some of the GL call, you must use `glFinish()`. The OpenGL calls are non-blocking, so you will never know which GL call the system is waiting for. `glFinish()` does not return until the effect of all previous GL commands are complete.

Move computation up the pipeline. There is no point of doing the same computation for every pixel in the fragment processor if you can do it only once on the CPU. Think of the pipeline as nested loops on the CPU, where the CPU is the outer loop, the vertex processor is an inner loop, and the fragment processor is the innermost loop. We would want to move computation to the outermost loop if possible.

2.4 Debugging

It is hard to debug the shader program. There is no `printf()` function or any other way to get the value of the variables during runtime. You can always read back the output at the end of the pipeline, but it might not be what you want.

There is not much help from the cgc compiler in debugging the code. If the shader code is listed as a character string in the C code, there is no error messages at all from the compiler. With the shader code in a separate file, the compiler will give some error messages. It is still not very helpful. A syntax error in the Cg code will give an error message like this.

CG ERROR : The compile returned an error. shaderComputeProgram

This error message is not very helpful, but at least you know that there is an error.

Conventional debugging tools like GDB is not useful for debugging GPGPU programs. These tools will only trace the code running on the CPU, and not the shader running on the GPU, which is what we are interested in. GDB will only report that `glEnd()` or some other gl-call take very long time.

There are some debugging tools for GPUs available. Duca et al. [4] present a relational debugging engine that seems interesting, but their program is not available. Other debugging tools that are available includes:

Shadesmith Fragment program debugger with graphical interface. Let you edit your fragment shader at runtime. Display values of registers at every pixel. Only available for Windows.

Imdebug Printf-style debugging. Read back memory and display in image window.

gDebugger OpenGL debugger. Support breakpoints and stepping

GLIntercept Another OpenGL debugger. Track all OpenGL state

Microsoft Shader Debugger Assembly and high level debugging. Only available for DirectX

ATI RenderMonkey ATI's shader development environment with debugging tools. Support DirectX and OpenGL. Windows only

Nvidia FXComposer Nvidia's shader development environment with debugging options. Support DirectX and OpenGL. Windows only

The available debugging tools are primarily for graphics applications. In most GPGPU applications it will be more useful to get the values in each pixels, rather that the image representations most debuggers present. Shadesmith give you a selected register value at each pixel, but is only available for Windows.

I did not find any suitable debugging tools that work under Linux and OpenGL. The only debugging I could do was to read back the output from the shader program, that is, read the content of the FBO back to a CPU array. This will help to verify that the shader works correctly and is helpful in many cases.

2.5 Conclusions

Writing a GPGPU program is quite different from writing a program for CPU. A GPGPU program is more like a SIMD parallel program for CPU where you run the same instructions on all the data. To write an efficient GPGPU program, you will have to choose the right data structures for your data, and possibly try different techniques to solve your problem.

Chapter 3

Programming Languages

3.1 Introduction

There are two kinds of programing languages that can be used to write GPGPU programs. Shading languages are, as the name suggest, used to write shaders for the GPU. Examples of shading languages is Cg, HLSL, and GLSL. The other type of languages is the high level GPGPU languages. These languages hide all the details of the OpenGL/DirectX API from the programmer. Examples of these languages is Brook, Sh, and Scout.

In my work I have tried one of each type of languages. Most of the code for the Mandelbrot is written in Cg. I have also implemented one Version in Brook and compared it to the Cg version (Figure 4.10 on page 26).

The choice of programming language will depend on the programming task, and the available graphics API. For my work, the shading languages turned out to be the most useful. HLSL only work with DirectX, so that is not an option for the display wall which runs Linux and OpenGL. The other options include Cg and GLSL. These two languages are not very different. GLSL has native supported in OpenGL, and does not require additional toolkit. Cg require the Cg toolkit.

It will not make any huge differences using one or the other of Cg and GLSL. I just had to decide for one of the languages, and I chose Cg. There is a lot of documentation and tutorials available on Cg, both on Nvidia's web pages and other. I also had The Cg Tutorial [7] book available, which was in favour of choosing Cg.

3.2 CG

CG is Nvidia's high-level shading language and is modeled on ANSI C (C for Graphics). Nvidia and Microsoft developed the language together and Nvidia called their implementation for CG, and Microsoft call it HLSL. CG and HLSL are almost syntactically identical. Cg can be used to write shaders for both OpenGL and Direct3D API.

Require Cg Toolkit.....

Cg support the same arithmetic and relational operators as C and C++. In addition, Cg has built-in support for arithmetic operations on vectors. Cg also provide a number of standard library functions. For a complete list of operators and library functions, check out the CG specification [13].

The standard library functions does not only make programming easier. In many cases these functions map to a single native GPU instruction, and can therefore be very efficient. I have done some testing of using library function instead of defining them myself, and the result I got show that the standard library functions are more efficient. See the result in subsection 4.4 on page 26.

3.3 Brook

Brook is an extension of standard ANSI C, and is a high-level GPGPU programming language. The data is represented as streams, which is a collection of data that can be operated on in parallel. Functions that operate on data streams are called kernels. kernels are special functions applied to every functions on the data stream. Full specifications and examples can be found in the Brook Spec v0.2 [1].

Brook is designed to hide the details of the graphics API from the programmer. You do not need any knowledge about OpenGL/DirectX to write a program for GPGPU using Brook.

A Brook program consist of some valid C/C++ code, and some Brook specific code in a separate file. The brook compiler will generate .cpp files from the .br files, and link against the runtime brook.lib. The platform is selected at runtime based on the runtime variable BRT_RUNTIME. The available platforms are CPU (cpu), OpenGL (ogl), and DirectX9 (dx9).

3.3.1 Installing Brook

The current release of Brook does not support Cg v.1.5, which is the current release of Cg. In order to get Brook up and running, you will need the Cg v.1.4.1. There is also problems with two of the GL header files, which will give some errors during compilation of Brook code. I found this on the gpgpu.org forum about the header files:

> The Nvidia header files are broken. Since they have moved to GL 2.0 and most of the ARB extensions have been promoted to core, they no longer declare them in the header files they install. If you reinstall mesa-dev after installing the Nvidia drivers, or, you get the latest gl.h/glext.h from oss.sgi.com, you should have better luck.

I got the latest gl.h and glext.h from the mesa-dev package and replaced them whit the one on my system, and then everything worked just fine.

3.4 Nvidia CUDA

CUDA (Compute Unified Device Architecture) is a computing architecture that make GPGPU development easier by using standard C for developing programs. CUDA also has a assembly language layer for advanced language development. The CUDA software development kit requires a CUDA-enabled GPU, which means a GeForce 8 series or a Quadro FX5600/4600.

CUDA programmers use C to write a kernel program that operate in parallel on large data set. The programmer do not have to think about graphics API like OpenGL, which make the program easier to write and debug. CUDA also offer the ability for threads on the GPU to communicate directly through an on-chip data cache.

CUDA also offer a hardware debugger and a performance profiler.

3.5 Conclusions

Brook is not the best choice for the Mandelbrot computation. Brook operate on data stream, and there is no way to access the framebuffer directly from the brook code. This means that the data must be read back to the CPU before output to screen. The data transfer between the CPU and the GPU will become a bottleneck in the Mandelbrot program when we start to zoom in on the image. The data would have to be transferred from the GPU to the CPU and back to the GPU for every time we zoom in on the Mandelbrot set. In Cg, we can write the data directly to the framebuffer.

When using one of the shading languages, you will have to have some understanding of the graphic API, and how the graphic adapter work. The high-level GPGPU languages hides all this details from you, but there is some drawbacks like the lack of direct access to the framebuffer. There is also a small performance penalty of using the high-level GPGPU languages, but the sacrifice in performance might be worth the much easier programming.

Chapter 4

Mandelbrot

4.1 Introduction

The Mandelbrot set is a set of points in a complex plane that are quasi-stable when computed by iterating the complex function

$$Z_{k+1} = {Z_k}^2 + C$$

where Z and C is complex numbers and Z_k is the kth iteration of Z. The initial value of Z is zero, and C is the position of the point in the complex plane. The next iteration values of Z can be produces by computing the real and imaginary part separately like this:

$$\begin{aligned} Z_{real} &= {Z_{real}}^2 - {Z_{imag}}^2 + C_{real} \\ Z_{imag} &= 2Z_{real}Z_{imag} + C_{imag} \end{aligned}$$

If the magnitude of Z gets greater than 2 it will eventually become infinite, and the point is not a part of the Mandelbrot set. The magnitude of Z will never become greater than 2 for points that are part of the Mandelbrot set. Obviously, we can not iterate the function infinite times to decide which points are part of the Mandelbrot set. We have to decide on a maximum number of iterations. If the magnitude of Z is not greater than 2 after max iterations, we consider the point to be a part of the Mandelbrot set. The picture will get more details with more iterations. The image will usually look fairly good with 100 iterations or more. The image in figure 4.1 on the next page is computed with 500 iterations and a resolution of 600x800 pixels.

I have implemented a version of the Mandelbrot algorithm that runs on the GPU. All the computation of the Mandelbrot set is done on the GPU and the CPU is only used to initiate computation on the GPU. I also have a CPU version of the same algorithm for reference. The Mandelbrot set can easily be divided into independent smaller part, and is therefore very suitably for parallel programming. The GPU architecture is parallel and the Mandelbrot set will be very suitable for implementation on a GPU.

The distributed version of Mandelbrot I have made is running on a cluster with high-end graphic cards. Each node on the cluster is responsible for calculating its own part of the image (static load balancing) and displaying it. Dynamic load balancing does not give the expected result on the GPU implementation, at least not for the Mandelbrot set. See section 4.3 on page 24

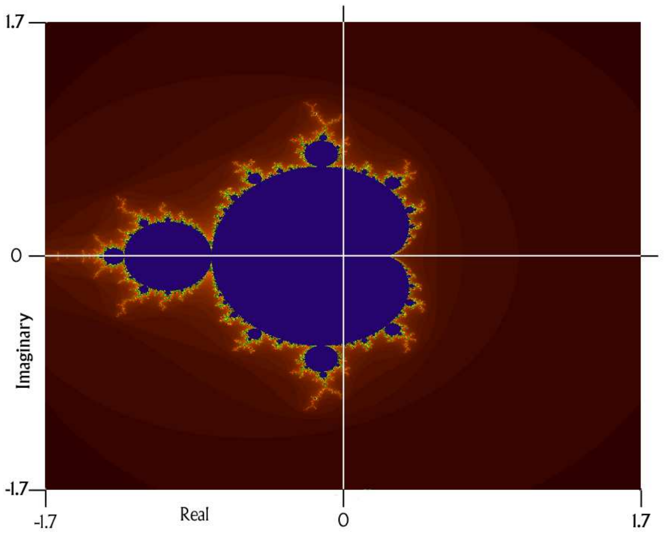


Figure 4.1: The Mandelbrot Set: Computed on the complex plane from -1.7 to 1.7 in both real and imaginary direction. The blue area are the pixels that are part of the Mandelbrot set

4.2 Implementation

The points on the complex plane must be scaled to match the display area. Color of each pixel is decided by the number of iteration computed on that pixel. A C-function for computing the Mandelbrot set on CPU, on a display of size width*height would be on the form:

```
scale_x = (realMax - realMin)/width;
scale_y = (imagMax - imagMin)/height;
for (y=0; y<height; y++){
    for (x=0; x<width; x++){
        Z_real = C_real = realMin + x * scale_x;
        Z_imag = C_imag = imagMin + y * scale_y;
        for (itr=0; itr<maxitr && (Z_real*Z_real + Z_imag*Z_imag) < 4; itr++){
            tmp = Z_real*Z_real - Z_imag*Z_imag + C_real;
            Z_imag = 2*Z_real*Z_imag + C_imag;
            Z_real = tmp;
        }
        display_pixel[y][x] = color(itr);
    }
}
```

`maxitr` is the maximum number of iteration computed on one pixel. The $Z_{real} * Z_{real} + Z_{imag} * Z_{imag}$ is compared against 4 rather than comparing $\sqrt{Z_{real} * Z_{real} + Z_{imag} * Z_{imag}}$ against 2 to avoid a square root operation. The color function maps the number of iterations to a given color. The pixels that belong to the Mandelbrot set is the pixels that is computed up to maximum iterations. In figure 4.1 the blue pixels are part of the Mandelbrot set.

The picture look best when there is a certain distance between the color for each iteration. If the color is to similar, some of the details get lost. I have used a color table which start with the color 0x0000002f for zero iterations. The color is a 32-bit number where the 3 least significant bytes represent respectively red, green, and blue. The color value is increased with the integer value 1032 for each iteration. A color table for 100 iterations is showed in table 4.1 on the facing page.

Iterations	32-bit color		8-bit Colors			
	Int	Hex	Alpha	Blue	Green	Red
0	47	`0x0000002f`	0	0	0	47
1	1079	`0x00000437`	0	0	4	55
2	2111	`0x0000083f`	0	0	8	63
⋮	⋮	⋮	⋮	⋮	⋮	⋮
98	101183	`0x00018b3f`	0	1	139	63
99	102215	`0x00018f47`	0	1	143	71
100	7144741	`0x006d0525`	0	109	5	37

Table 4.1: Color Table for 100 iterations. The 32–bit color value increases with 1032 for each iteration. This table show the different color values for red, green, and blue, extracted from the 32-bit color. The last entry (max iteration) is a special color for pixels that are part of the Mandelbrot set, and does not follow the 1032 increase

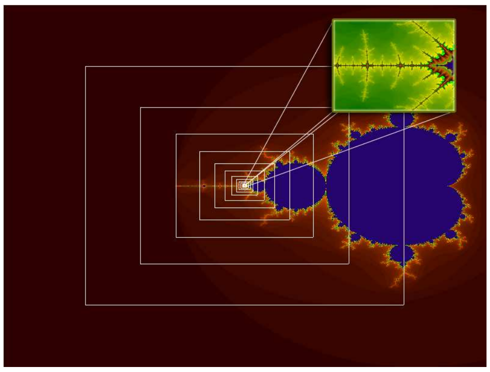


Figure 4.2: Zoom function 1. Zoom in on point(-1.42, 0). This function is used for the initial testing on one computer.

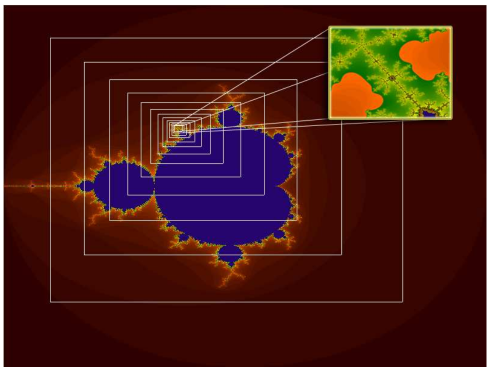


Figure 4.3: Zoom function 2. This is the function used on the display wall

4.2.1 Zooming

To zoom in on the Mandelbrot set we have to map a smaller part of the complex plane to the display area. My initial zoom algorithm zoom in on one point, meaning that the same chosen point is the center of the image all the time as in figure 4.2. When comparing performance of different implementations of Mandelbrot, it is important to zoom in on the same part of the image. The computation time is very different for different part of the image.

The parallel version on the display wall is compared against two existing parallel CPU implementations of Mandelbrot, one with static load balancing by *Jon M. Bjørndalen*, and one with dynamic load balancing by *Daniel Stødle*. For this test I changed my zoom function to zoom in on the same area as the static load balanced version. see figure 4.3.

4.2.2 GPU implementation

My Mandelbrot implementation on GPU is written in Cg, Nvidia's high level language for GPU programming. Cg stands for C for Graphics, and the syntax is similar to C.

To compute a Mandelbrot image we first need some initial data that is usually stored in an array on a CPU implementation. On a CPU, the data is iterated one element at the time, and the computation is finished for one pixel at the time. When the computation is finished for all pixels, the image can be displayed on a screen, and the next round of computation on a smaller area of the complex plane can be started.

On the GPU, the data is stored in a texture and we use a shader program to manipulate the data on the texture. There are two ways to get the initial data into the texture. One is to create the data on the CPU, and transfer it to the GPU. The other is to create an empty texture on the GPU, and let the GPU create the initial data. I have used the second approach in my algorithm. The advantage of this approach is that we do not have to allocate memory for the data on the CPU, fill it with data, and than transfer it to the GPU for each time we start to compute on a new image (every time we zoom in on the Mandelbrot set).

Pseudo code for the CPU side of my GPU implementation

```
create_textures ()
FOR number of zoom:
    set_rendering_target(frame_buffer_object)
    load (initialization_shader)
    run (initialization_shader)
    load (compute_shader)
    FOR maximum number of iterations:
        set_input_data (offscreen_texture_1)
        set_output_data (offscreen_texture_2)
        run (compute_shader)
        swap (offscreen_textures) //Input texture becomes output and vice versa
    set_rendering_target (frame_buffer)
    load (draw_shader)
    run (draw_shader)
    swap (frame-buffers)
    zoom ()
```

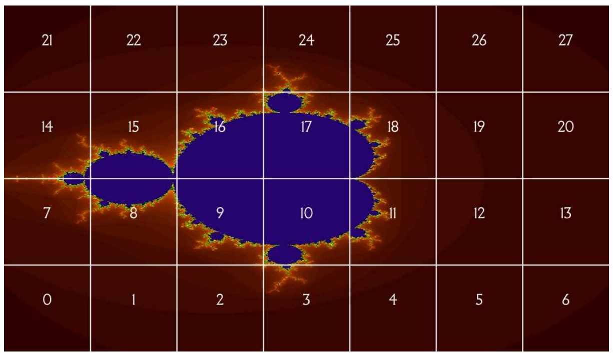
The pseudo code show the concept of the three different shader program, and how they are used to compute the Mandelbrot set. The offscreen textures are swapped between being input data and output data for every other iteration. All data is kept on the graphic card at all time. The draw-shader write the data to the frame buffer (double buffered), and all I have to do on the CPU side to display the image, is to call `swap` on the frame buffers.

This mean that i have two different shader program to compute the Mandelbrot set, and I actually have a third shader program to display the image on the screen. The shader program to compute the initial data looks like this:

```
float4 initM( in float2 coords : TEXCOORD0,
              ..... ,
              uniform float imagScaleI) : COLOR {
    float2 idx = texRECT (index, coords);
    return float4( realMinI + (idx.x * realScaleI),
                   imagMinI + (idx.y * imagScaleI),
                   0.0,
                   0.0);
}
```

This function is similar to setting the initial values of Z_{real} and Z_{imag} in the CPU version. `coords` is the actual coordinates of the pixel being computed. The `index` is a texture which contain coordinates created on the CPU side. This custom indices is useful when running the algorithm on the display wall, and we need to treat the entire wall as one coordinate system. The initial data computed in this function is stored in an output texture (return value), which will become the input texture in the first iteration of the actual computation.

The shader program for computing the Mandelbrot set look like this:



The Mandelbrot Set on the display wall tiles, computed on the complex plane from -1.7 to 1.7. The numbering is the order of tiles in the lamhost file. Each tile map the appropriate area from the complex plane to the display, and compute its own image

Figure 4.4: The display wall tiles

```
float4 computeM(in float2 coords : TEXCOORD0,
                ....,
                uniform float imagScaleC) : COLOR {
    float4 idx = texRECT (index, coords);
    float4 intx = texRECT (textureC, coords);
    float2 complex = float2(realMinC + (idx.x * realScaleC),
                            imagMinC + (idx.y * imagScaleC));
    if (dot(intx.xy, intx.xy) > 4)
        discard;
    return float4(  intx.x*intx.x - intx.y*intx.y + complex.x,
                    2.0 * intx.x * intx.y + complex.y,
                    intx.z + 1.0,
                    0.0);
}
```

If the magnitude of Z (intx.x, intx.y) is greater than 4, further processing on that pixel is stopped. If not, the computation is done on the CPU. The counter (intx.z) is incremented, and values are written to output texture.

The shader program that draw the image to the frame buffer

```
float4 drawM( in float2 coords : TEXCOORD0,
              uniform samplerRECT textureD,
              uniform float maxItr ) : COLOR{
    float4 intx = texRECT (textureD, coords);
    float4 outtx;
    if(intx.z == maxItr) // pixel is part of the Mandelbrot set
        outtx = float4(37.0/255.0, 8.0/255.0, 109.0/255.0, 0.0);
    else {
        float color = 47.0 + (intx.z * 1032.0);
        float r = fmod(color, 256.0);
        color /= 256.0;
        float g = fmod(color, 256.0);
        color /= 256.0;
        float b = fmod(color, 256.0);
        outtx = float4(r/255.0, g/255.0, b/255.0, 0.0);
    }
    return outtx;
}
```

The draw shader calculate the color of the pixel from the number of iterations, and write the result to the frame buffer.

4.2.3 Parallel GPU version

The parallel implementation uses the same three shader as described above. The only difference from the single computer version is the mapping from the complex plane to the

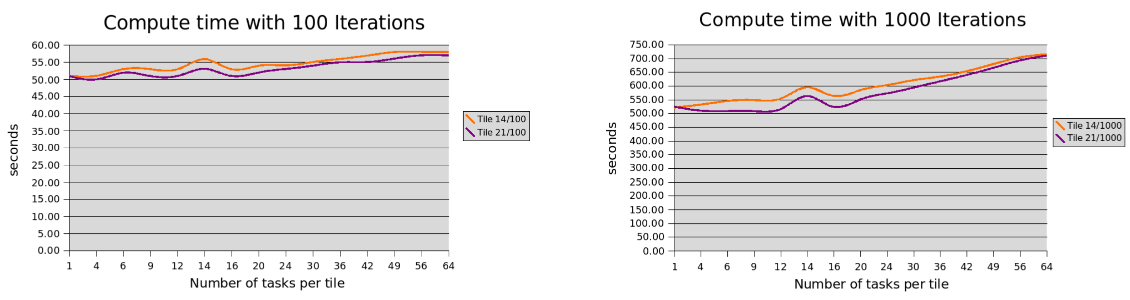


Figure 4.5: Performance with multiple tasks per tile. There is less computation with more tasks, but the time increase

display area. Each tile on the display wall compute its own part of the image. Every tile will map the appropriate part from the complex plane to its display area, compute the image, and display it on the screen. To synchronize the screen update, I have used MPI. Just before the screen update, every tile call MPI_Barrier, and wait until all tiles are ready to update the screen.

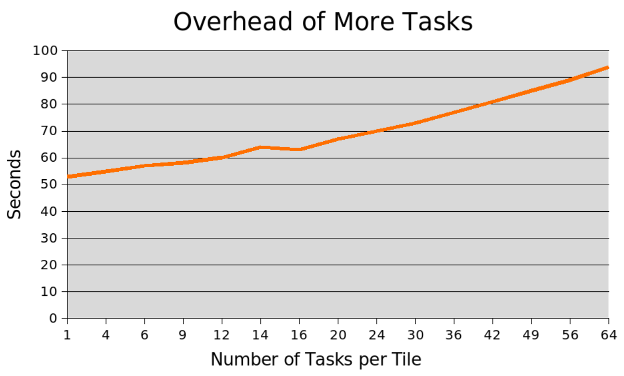
4.3 Load–Balancing

The Mandelbrot set i very suitable for dynamic load balancing. Each pixel will demand different amount of computation, and with static load balancing the load on each node on the cluster will be uneven. The nodes that finish first will be waiting for the other nodes, without doing any useful work. This is at least the case for Mandelbrot on CPU.

For the GPU implementation, it is a little different. The GPU itself perform computation in parallel. Conceptually, every pixel on the image is computed in parallel. The shader program that does the computation is called `max iteration` times (see subsection 4.2.2 on page 21, and one iteration is computed for every pixel each time. This is different from the CPU version, that compute all iteration for one pixel before continuing to the next. The shader must be called as long as there is some pixels left to compute, and it is the overhead of calling the shader that make it difficult to benefit from dynamic load balancing on the GPU. With static load balancing the GPU Version use about the same time to finish, independent of where we zoom in on the image.

Calling the compute–shader function is equivalent to draw a full screen quad. As long as there is one more pixel on the screen that is not finished computed, the shader function has to be called. Occlusion Queries can be used to count the number of pixels that are discarded. This count can be used by the CPU to decide when all pixels are discarded, and the fragment program can be terminated. The problem with this is that as long as there is one more pixel on the screen that is not finished computed, the shader function has to be called. When zooming in on the Mandelbrot set it is very unlikely that any tile will have none pixels from the Mandelbrot set. With occlusion query, the pseudo code from subsection 4.2.2 on page 21 looks like this

```
numberNotTerminated = imageSize
FOR (maximum number of iterations) && (numberNotTerminated > 0):
    set_input_data (offscreen_texture_1)
    set_output_data (offscreen_texture_2)
    gl_begin_query(query)
    run (compute_shader)
    gl_end_query(query)
    numberNotTerminated = glGetQueryResult(query)
    swap (offscreen_textures) //Input texture becomes output and vice versa
```



This figure show the overhead of the extra rendering when the computation is divided into more tasks. The test is performed on one computer and all pixels in the image are part of the Mandelbrot set, so there is no benefit from fewer computations.

Figure 4.6: Overhead with extra rendering

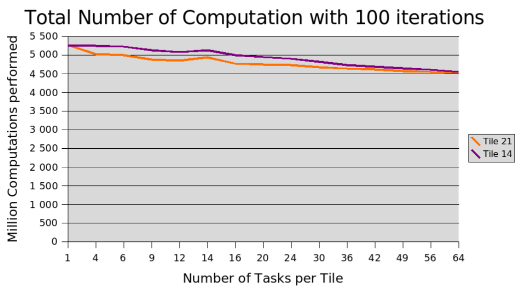


Figure 4.7: Decrease of computation with more tasks

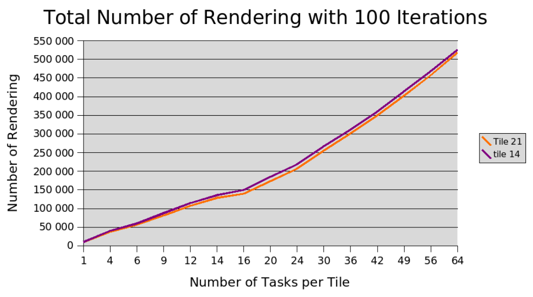


Figure 4.8: Increase of rendering with more tasks

To benefit from the occlusion query, I divided each tile into smaller tasks. The smaller the tasks are, the more likely that some of them will finish early. The division is done by rendering smaller quads. The data is still in two textures, and kept on the graphic card all the time. The only overhead of doing this on one tile is the extra rendering. Unfortunately, the overhead of the rendering is quite significant. Figure 4.6 show the overhead of the extra rendering without benefiting from fewer computations.

I ran some test with on two on the tiles that will benefit most from this technique. That is, the tiles with most pixels outside the Mandelbrot set. The tiles I used for testing is nr.14 and nr.21 (see figure 4.4 on page 23). The test is done with the tiles divided into different number of parts and with different number of maximum iterations. The result with 100 and 1000 iterations is in figure 4.5 on the preceding page. The time increase with more tasks, at the same time as the amount of computation decrease (figure 4.7). A similar experiment on the CPU would be to divide the inner loop into several smaller loop, which would have no significant influence on the computing time.

We get less computation by dividing the image into smaller tasks. The overhead of dividing the computation into smaller tasks, is the cost of more rendering. Unlike the CPU, it cost quite a lot to start a computation on the GPU. When want to do as much work as possible for each rendering operation (=each computation started). Because of this overhead with rendering, the Mandelbrot is not suitable for dynamic load balancing on GPU. Other applications might benefit more from this technique, if there is more work done for each rendering. The application would still have to have different workload for different parts of the computation.

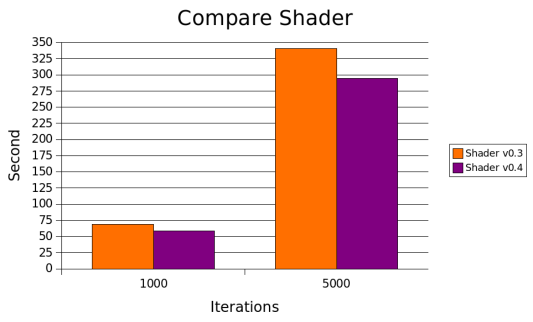


Figure 4.9: Performance of two different shaders. Zoomed 30 times with zoom function 1

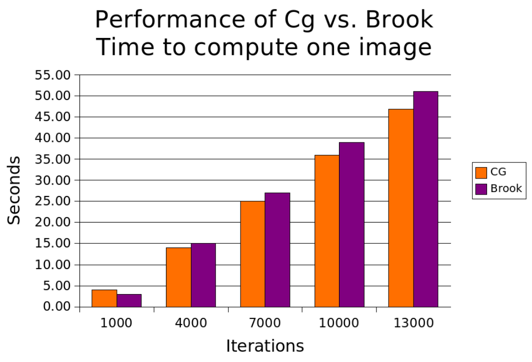


Figure 4.10: Cg vs Brook

4.4 Testing and improvements

Small changes in a shader program can make great difference in performance. Lets take a look at two different version of my compute shader. Compute Shader v0.3:

```
if((intx.x*intx.x + intx.y*intx.y) > 4){
    outtx = intx;
    discard;
} else {
    outtx.x = intx.x*intx.x - intx.y*intx.y + complex.x;
    outtx.y = 2.0 * intx.x * intx.y + complex.y;
    outtx.z = intx.z + 1.0;
}
return outtx;
```

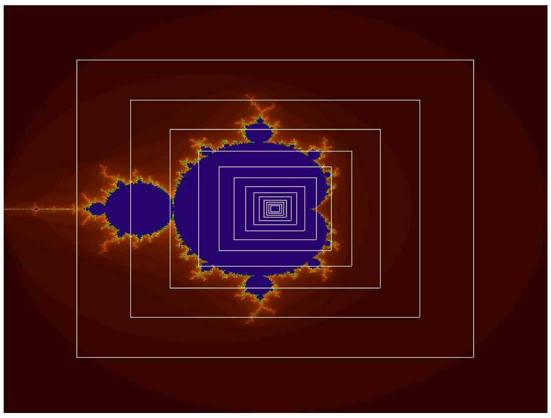
Compute Shader v0.4:

```
if (dot(intx.xy, intx.xy) > 4)
    discard;
return float4( intx.x*intx.x - intx.y*intx.y + complex.x,
               2.0 * intx.x * intx.y + complex.y,
               intx.z + 1.0,
               0.0);
```

These two shaders give the exact same output. In shader v0.4 i have removed the `else` statement from the branch, and replaced the calculation in the if-test with the `dot` function. These small changes gave an improvement of 14% on the computation time, as seen in figure 4.9.

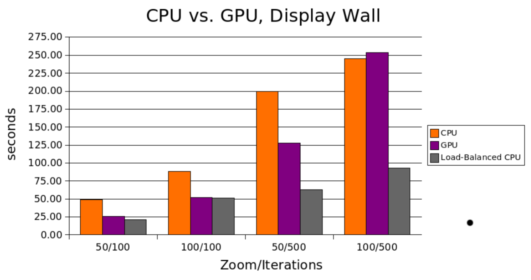
It is very important to test the performance for each optimization done. Optimizations that seem obvious, might turn out to slow down the algorithm. When I tried to replace `if((intx.x*intx.x + intx.y*intx.y) > 4)` with `if (dot(intx.xy, intx.xy) > 4)` in v0.3 without doing the other changes, the performance actually get worse. With the other changes done in v0.4 however, the usage of **dot** function had a small positive effect on performance. The **dot** function is part of the Cg Standard Library and should be more efficient, as it map to special GPU instructions. See *The Cg Tutorial* [7, Chapter 3].

According to [16, Chapter 30] the cost of a `if/else/endif` instruction is 6 clock cycles. A `if/endif` has a cost of 4 clock cycles. By rewriting the code to get rid of the `else` instruction, we will save a lot of clock cycles. A lot of these pixels are computed in parallel so it is hard to say exactly how many clock cycles are saved.



The third zoom function, to compare CPU and GPU implementation with more computing. Every square represents three round of zooming. After 20 round of zooming, the Mandelbrot set will cover all of the image. At this point the dynamic load balanced CPU implementation do not have any advantage of the load balancing.

Figure 4.11: Zoom function 3



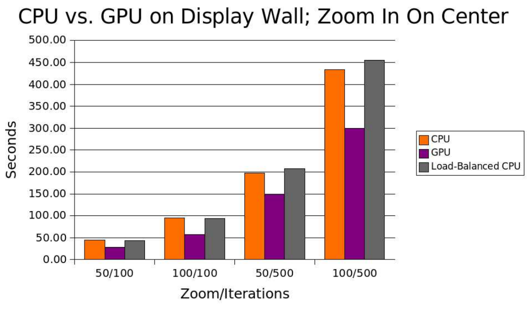


Figure 4.12: CPU vs. GPU on the display wall. Zoom function 2

Figure 4.13: CPU vs. GPU on the display wall. Zoom function 3

4.4.1 Performance of GPU vs. CPU

I have compared the GPU version of the Mandelbrot with two different CPU version on the display wall. The test is performed with two different zoom function, zoom function 2 (figure 4.3 on page 21), and zoom function 3 (figure 4.11). The purpose of using these two zoom function is to see how the different implementations performs with different work load. Zoom function 2 zoom in on the edge of the Mandelbrot set, and zoom function 3 zoom in on the middle of the set (center of the complex plane). Zooming on the edge will benefit the dynamic load balanced implementation. There will be a lot of areas with low computational cost. The third zoom function will eventually compute only on pixel that are part of the Mandelbrot set. The dynamic load balanced version will not be able to benefit from the load balancing when the amount of computation is the same for all parts of the image. The interesting here is to see if the GPU version is faster when they get the same amount of work to do.

The result of this test is shown in figure 4.12 and figure 4.13. The GPU implementation is faster than the CPU implementation with static load balancing with zoom function 2, except for the last test with 500 iterations and 100 zoom. The CPU version has an advantage with 100 rounds of zoom, as the amount of computation get smaller when zooming in on the edge. The amount of pixels belonging to the Mandelbrot set is fewer, and the CPU implementation can benefit from early exit from the iteration loop as explained in section 4.3 on page 24. The CPU implementation with dynamic load balancing is as expected faster than the one with static load balancing and the GPU implementation.

The test with zooming in on center of the Mandelbrot set (figure 4.13, show very little difference between the two CPU implementations. The CPU implementation with dynamic load balancing is a little slower than the one with static load balancing. This comes from the extra overhead with the dynamic load balancing, without getting any advantage over the static load balanced implementation from uneven distributed load. When we zoom in on the middle of the Mandelbrot set, the work load will be distributed evenly between the tasks. The GPU implementation is significantly faster than both CPU implementations for this test. This test show that the GPU compute the calculations in the Mandelbrot set faster than the CPU, but looses to the CPU in some cases because the GPU has to do some more work.

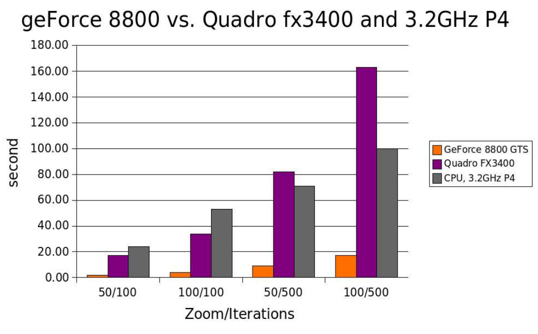
4.4.2 Performance of Quadro FX3400 vs. GeForce 8800 GTS

	Core Clock (MHz)	**Memory Clock (MHz)**	**Memory Size (MB)**	**Memory Type**	**Memory Bandwidth (GB/s)**
Quadro FX3400	350	450	256	256-bit GDDR3	28.8
GeForce 8800GTS	500	800	640	320-bit GDDR3	64

Table 4.2: Technical specification for GeForce 8800GTS and Quadro FX3400

The comparing of these two graphic cards are performed on one computer only. I have only one computer available with the GeForce 8800GTS card, so I have not been able to do any test on a parallel version on these cards. The CPU implementation I have compared with is run on the computer with the GeForce card, but the performance on the CPU is similar on both computers. The computers are Dell 370 Prescott 64 EMT, 3.2GHz, 2GB RAM. The performance on the CPU is not dependent of the graphic card on the computer as most of the work is done on the CPU.

As figure 4.14 show, the Quadro card is faster than the CPU for the test with 100 iterations. When the number of iterations is increased to 500, the CPU is faster. The GeForce card is faster than both the Quadro card and the CPU. This give some impression of the development on graphic cards the last years. It is about 3 years between these two graphics cards (2004-2007).



Performance of Nvidia Quadro FX3400 vs. Nvidia GeForce 8800GTS. Both test of GPU is performed on a similar computer; Dell 370 Prescott 64 EMT, 3.2GHz, 2GB RAM.

Figure 4.14: GeForce 8800GTS vs. Quadro FX3400

4.4.3 Brook

Brook for GPU [1] is a high level language for GPGPU applications. The Brook compiler generates c++ and Cg files from the brook file. The Brook language make it possible to write GPGPU application without knowledge about OpenGL or the graphic pipeline. A comparison between a Brook implementation and my Cg implementation would give an idea of how fast my Cg implementation is.

I have written a Mandelbrot application in Brook that compute one Mandelbrot image and write it to file. The Brook kernel function does not have direct access to the frame buffer, so the image has to be read back to the CPU before displayed on screen. The Brook language is therefor not suitable for a Mandelbrot application with zooming. I have compared the Brook implementation with my Cg implementation that also compute just one image and write it to file. I have compared the two implementation with different number of iterations, and the result is showed in figure 4.10 on page 26.

Brook is a little bit slower than the code written in Cg. This is as expected and similar to other experiments with Brook [2]. The result for 1000 iterations show that Brook is a bit faster than Cg. I am not sure why, but I suspect that the time is so low that the measurement is not accurate.

4.5 Conclusions

The Mandelbrot set is suitable for implementation on GPU. The data for the Mandelbrot computation is a natural grid, and maps very easy to the data structure of the GPU (textures). Each pixel in the Mandelbrot computation is independent of the other pixels and fits very well for the parallel architecture of the GPU.

It is hard to get some performance gain from dynamic load balancing for the Mandelbrot computation on the GPU. Dividing the computation into smaller tasks give a lot of overhead with the extra rendering. The time saved with less computation is not enough to make up for the extra time spent on rendering. However, the performance of the GPU is still better than the CPU for some cases. There is no point in doing dynamic load balancing if the time computing time is the same for all areas of the task. The performance on the Mandelbrot computation on newer graphic cards, like the Nvidia GeForce 8800GTS in subsection 4.4.2 on the facing page, is so good that it will be little to gain with dynamic load balancing.

The GPU has only single precision floating point (32-bit), and this will limit how much it is possible to zoom in. With the zoom function 2, that I have used for the tests on the display wall, it will limit the numbers of zooms til 100.

Chapter 5

Conclusions and Future Work

I have implemented the Mandelbrot set on a cluster of GPUs. The GPU implementation have better performance in some cases, compared to the CPU versions on the cluster. The Mandelbrot set is not ideal to measure the performance of the GPU as it Mandelbrot does not fully utilize the parallel architecture of the GPU. I would expect other application which have the same amount of computation for every rendering pass to Perform even better compared to the CPU.

The rapid increase in performance of the GPU is expected to continue in the future, and that the price will be low compared to specialty hardware. We have a lot of high-level languages and some vendor specific tools available for developing GPGPU code. These tools will develop and become better and easier to use in the future. All this things will probably make GPGPU programming more popular.

Nvidia has announced the release of the Nvidia Tesla system in October 2007. This is the first line of graphics chips designed for GPGPU. The product line consist of PCI Express add-on cards, desktop supercomputers, and computing servers. The Tesla system is used together with the Nvidia CUDA system which make GPGPU programming easier. This system will make GPGPU programming more available, and it is likely to believe that we will see more systems like this in the future.

Future Work: The Mandelbrot set is only useful to show the performance of the GPU cluster. We would like to use the cluster for some useful applications, and sorting stands out as a very interesting candidate for future work on a cluster of GPUs.

Bovindaraju et al. [10] show some very good result using a bitonic merge-sort algorithm to utilize the GPU as a co-processor to sort large databases. The bitonic sort algorithm seem to be the algorithm that fits the GPU architecture best. Their algorithm run on one computer only, but it should be possible to do something similar on a cluster of GPUs. In a future work we could use their work as a starting point for implementing a sorting algorithm on the cluster of 28 computers behind the HPDC display wall.

Bibliography

[1] BUCK, I. *Brook Spesification v0.2*. Stanford University, http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf.

[2] BUCK, I. Brook for GPUs. MDA Workshop, February 11th 2004. http://graphics.stanford.edu/projects/brookgpu/talks.html.

[3] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[4] DUCA, N., NISKI, K., BILODEAU, J., BOLITHO, M., CHEN, Y., AND COHEN, J. A relational debugging engine for the graphics pipeline. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), ACM Press, pp. 453–463. http://doi.acm.org/10.1145/1186822.1073213.

[5] FAN, Z., QUI, F., KAUFMAN, A., AND YOAKUM-STOVER, S. GPU Cluster for High Performance Computing. Proceedings of ACM/IEEE Supercomputing Conference, November 2004. http://portal.acm.org/citation.cfm?id=1048933.1049991.

[6] FATAHALIAN, D., SUGERMAN, J., AND HANRAHAN, P. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. Graphics Hardware, 2004. http://graphics.stanford.edu/papers/gpumatrixmult/.

[7] FERNANDO, R., AND KILGARD, M. J. *The Cg Tutorial*. Addison–Wesley, 2003.

[8] GALOPPO, N., GOVINDARAJU, N. K., HENSON, M., AND MANOCHA, D. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. ACM/IEEE SC/05 Conference, November 2005.

[9] GÖDDEKE, D. GPGPU::Basic Math Tutorial. http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html.

[10] GOVINDARAJU, N. K., GRAY, J., KUMAR, R., AND MANOCHA, D. GPUTeraSort: High Performance Graphics Co-Processor Sorting for Large Database Management. ACM SIGMOD, 2006. http://gamma.cs.unc.edu/GPUTERASORT/.

[11] GOVINDARAJU, N. K., LARSEN, S., GRAY, J., AND MANOCHA, D. A Memory Model for Scientific Algorithms on Graphics Processors. ACM/IEEE conference on Supercomputing, 2006.

[12] HARRIS, M. J., III, W. V. B., SCHEUERMANN, T., AND LASTRA, A. Simulation on Cloud Dynamics on Graphics Hardware. Proceedings of Graphics Hardware, 2003.

[13] NVIDIA. *CG Language Spesification*. http://developer.nvidia.com/object/cg_specification.html.

[14] OWENS, J. Experiences with GPU Computing, April 2007. `www.ece.ucdavis.edu/~jowens/talks/intel-santaclara-070420.pdf`.

[15] OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E., AND PURELL, T. J. A Survey of General-Purpose Computation on Graphics Hardware. Eurographics, August 2005. `http://www.cs.virginia.edu/papers/ASurveyofGeneralPurposeComputationonGraphicsHardware.pdf`.

[16] PHARR, M., Ed. *GPU Gems 2*. Addison–Wesley, 2005.

[17] PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2003), Eurographics Association, pp. 41–50.

[18] WILKINSON, B., AND ALLEN, M. *Parallel Programming*, second ed. Pearson Education International, 2005.