



UNIVERSITEIT VAN AMSTERDAM

# Generic Computing on a Graphics Processing Unit

Peter Geldof  
Student no. 0023094

Master of Science Thesis, Section Computational Science  
Universiteit van Amsterdam, Amsterdam, The Netherlands  
Supervised by: Dr. Robert Belleman

May 2007



## **Abstract**

With the increasing demand for better, more realistic and more detailed graphics, more and more effort and money is put into improving the Graphics Processing Unit (GPU) in a computer. As a result of this, the GPU has become a very powerful computing unit with its own programmable processor and a unique architecture. Unlike CPUs, GPUs are special purpose hardware, designed to perform tasks often encountered in computer graphics. While CPU manufacturers only just start using multiple cores in a CPU, parallelism is already in long use in GPUs.

Though GPUs are designed for graphics computations, they can be used for more. This research project explores the application of GPUs both in graphics and in generic computing. The emphasis will be on the latter. We explain how the GPU can be used to build useful programs that can speed up simple cellular automata to over 50 times faster than on a conventional processor. We also show that complex computations like the gravitational  $N$ -body problem can be turned into a GPU based version to speed up computation.

# Acknowledgements

In my work at the graduation project presented in this thesis, I have received help from many. First of all, I thank my supervisor Robert Belleman for supplying me with an endless amount of advice and discussion. Many thanks go to Simon Portegies Zwart, with whom I had the privilege to work with on the gravitational  $N$ -body problem. Furthermore, I am grateful to Mark Harris and David Luebke of NVIDIA for supporting my work by supplying two NVIDIA GeForce 8800 GTX graphics cards. I would like to thank Alessia Gualandris for supplying me with data and information. Calculations were done at the Hewlett-Packard xw8200 workstation cluster and the MoDeStA computer, both hosted by SARA computing and networking services, Amsterdam.

# Contents

<b>1</b>	<b>The graphics processing unit</b>	<b>4</b>
1.1	Introduction . . . . .	4
1.2	History . . . . .	4
1.3	Architecture . . . . .	5
1.3.1	The Stream Programming Model . . . . .	6
1.3.2	Vertex and fragment shaders . . . . .	7
1.4	Related work . . . . .	7
1.5	Overview of this thesis . . . . .	8
<b>2</b>	<b>Programming GPUs</b>	<b>9</b>
2.1	Graphics oriented programming languages . . . . .	9
2.1.1	RenderMan Shading Language . . . . .	9
2.1.2	PixelFlow . . . . .	10
2.1.3	Real-Time Shading Language . . . . .	10
2.1.4	Cg: C for graphics . . . . .	11
2.1.5	High Level Shader Language . . . . .	11
2.1.6	OpenGL Shading Language . . . . .	12
2.2	Generic oriented programming languages . . . . .	12
2.2.1	Sh . . . . .	13
2.2.2	Brook . . . . .	13
2.3	Selection and discussion . . . . .	14
2.4	An example GPU program . . . . .	14
<b>3</b>	<b>Generic computing on a GPU</b>	<b>19</b>
3.1	Introduction . . . . .	19
3.2	Conway's Game of Life . . . . .	20
3.2.1	A CPU implementation . . . . .	20
3.2.2	A GPU implementation . . . . .	21
3.3	Performance . . . . .	23
3.4	Conclusion and discussion . . . . .	23

<b>4</b>	<b>High performance direct gravitational <math>N</math>-body simulations on Graphics Processing Units</b>	<b>26</b>
4.1	Introduction . . . . .	26
4.2	Direct summation . . . . .	27
4.3	Time step schemes . . . . .	27
4.3.1	Shared time step scheme . . . . .	27
4.3.2	Individual time step scheme . . . . .	27
4.3.3	Block time step scheme . . . . .	28
4.4	Prediction and Correction . . . . .	28
4.5	Accuracy of the implementation . . . . .	28
4.6	The GRAPE special-purpose computers . . . . .	28
4.7	Implementation on a GPU . . . . .	29
4.8	Performance . . . . .	30
4.8.1	Precision . . . . .	31
4.9	Conclusion and discussion . . . . .	32
<b>5</b>	<b>Conclusion and discussion</b>	<b>34</b>
5.1	Performance . . . . .	34
5.2	Precision . . . . .	34
5.3	Suitable applications . . . . .	35
<b>6</b>	<b>Future work</b>	<b>36</b>
6.1	CUDA . . . . .	36
6.2	More shaders for the $N$ -body problem . . . . .	36
6.3	ATI/AMD fushion . . . . .	37
6.4	A look in the future . . . . .	37
<b>A</b>	<b>The <math>N</math>-body problem fragment program</b>	<b>43</b>

# Chapter 1

## The graphics processing unit

### 1.1 Introduction

Graphics Processing Units (GPUs) used to be fixed function processors that helped the CPU with displaying images. This changed early 2001, when the first programmable GPU was launched. With the GPU being programmable, developers had much more opportunities to control the graphics operations themselves.

Programming a GPU though, is very different from programming a conventional CPU. There are programming languages designed for programming a GPU that are easy to learn, but taking full advantage of the computational power of the GPU, requires knowledge about the GPU. To fully exploit the power of the GPU, it is advisable to understand its architecture and its capabilities. Also, knowing how the data exactly passes through the GPU is required in order to build efficient programs.

GPUs as they are now, have gone through an evolution where functionality, speed and capability improved over the years. Section 1.2 will go through the most important breakthroughs and explains how GPUs have become what they are now. Communication to, from and inside the GPU determines for a large part the performance that can be achieved. Over time, GPUs have used many different buses and slots with increasing maximum bandwidths. Also, the architecture has changed a lot. This is shown in section 1.3. Section 1.4 gives an overview of related work in both graphical and non-graphical use of the GPU. Finally, in section 1.5 gives an overview of this thesis.

### 1.2 History

The origin of the GPU as it is known today dates back to the late 1970s. The earliest graphic chips simply fetched the required data from system memory. The video controller then used this data to send images directly to the video output device. Not long after, video controllers would get memory of their own to use for graphics output. Some of these controllers could also do simple rasterizing operations. A good example of such a rasterizing operation is ‘bit Block Transfer’ (bitBLT). BitBLT is a technique where a bitmap image is

combined with another image, to avoid repeated rendering of ‘fixed’ attributes in an image, like a background. In the years after that, the rasterizer was augmented with hardware to support additional functionality. This concept of using a graphics card to speed up graphical operations became known as 2D hardware acceleration and became more and more common in computer systems. In the 1990s, there was an increasing demand for 3D hardware acceleration, as games became more and more complex. While 3D hardware acceleration could first only be achieved with CPU assistance, later on, separate 3D graphics became possible using add-on boards. These boards could not do 2D acceleration, and had to be used in combination with another video controller. In 1996, the first chipset was released which combined both 2D and 3D acceleration on a single graphics card. As the use of GPUs continued to grow, more functionality was added as well. Two major additions were the so called ‘Transform and Lighting’ (T&L). ‘Transform’ refers to an operation which converts 3D objects to a 2D view. While ‘Lighting’ refers to light sources that can be placed in a 3D environment while the 3D environment responds to it realistically. In 2001, programmable shaders were added to the GPU, making it much more controllable in its use for the developer. This also made it possible to use the GPU for more purposes than simply graphics.

Many companies have contributed to the current position of the GPU. Among the first was Evans & Sutherland (E&S), a company that produces hardware for visualization and simulation purposes. E&S specializes in building flight training simulators. Today, approximately 80% of the world’s commercial airline pilots are trained using E&S visual systems. One of E&S’s employees was Jim Clark, who founded Silicon Graphics, Inc. (SGI) in 1982. SGI created several high-performance based on Clark’s Geometry Engine, specialized hardware that accelerated the ‘inner-loop’ geometric computations needed to display. Later, SGI also developed a proprietary API known as IRIS Graphics Language (IRIS GL). As IRIS GL improved and more features were added, it became harder to maintain. In 1992, IRIS GL was renamed to OpenGL to be cheaply licensed by SGI’s competitors [27]. To this day, OpenGL remains the only real-time 3D graphics standard to be portable across a variety of operating systems. Its main competitor, Direct3D from Microsoft, runs only on Microsoft Windows-based machines [23]. Founded in 1985, ATI started with producing integrated graphics chips for large PC manufacturers like IBM [1]. Since 1987, it has developed independent graphics cards and graphics chips for game consoles. Currently, ATI is the second largest manufacturer of GPUs. Over the last decade, ATI’s main competitor is NVIDIA [25]. NVIDIA was founded in 1993 and dominates the GPU market together with ATI since 1997.

### 1.3 Architecture

The data a GPU receives, always comes from the CPU. The data a GPU needs as input is typically very large, and therefore it is important to know a little about the architecture in which the GPU resides and the speed of the connections between CPU and GPU.

A GPU is nowadays usually connected to the CPU via the northbridge<sup>1</sup> to either a PCI or an AGP bus. Since 1993, the standard bus for attaching peripheral devices to the mainboard

---

<sup>1</sup>The northbridge is used for both AGP and PCIe slots. For PCI and PCI-X, the southbridge is used.

is the Peripheral Component Interconnect, or PCI Standard. As can be seen in Table 1.1, PCI was improved several times, to PCI 2.2 and PCI-X, when AGP dominated the graphics bus market for a while. Now, a new PCI update called PCI Express (or PCIe) rules the market again with maximum bandwidths of up to 250 MB/sec per lane in both directions. With 16 lanes, this means a bandwidth of 8000 MB/sec. Especially on AGP, upload and download speeds are very different. The upload speed to the GPU is usually much higher than the download speed. On PCIe this difference is much less. There, the download speed is even a little higher than the upload speed. In generic computing, this difference is more important as output data is often required by the CPU.

Protocol	Size	Max bandwidth
PCI	32-bit	133 MB/sec
PCI	32-bit	266 MB/sec
PCI	64-bit	266 MB/sec
PCI	64-bit	532 MB/sec
PCI-X	64-bit	800 MB/sec
PCI-X	64-bit	1066 MB/sec
AGP 1x	32-bit	266 MB/sec
AGP 8x	32-bit	2133 MB/sec
PCIe 1x	32-bit	500 MB/sec
PCIe 16x	32-bit	8000 MB/sec

Table 1.1: Communication busses and bandwidths between CPUs and GPUs.

### 1.3.1 The Stream Programming Model

While the previous section discussed the GPU as a part of the PC architecture, this section will discuss more on the architecture of the GPU itself. In order to build efficient programs, it is very important to know how the data passes through the GPU. The Stream Programming Model, as opposed to the serial programming model in CPUs, uses stream processing to expose parallelism and communication patterns in the application. Therefore, it allows both efficiency in computation and in communication.

A ‘stream’ may be defined as an ordered set of data of the same data type. Such a stream passes through a series of ‘kernels’. These kernels operate on the entire stream and produce one or more streams as output, typically performing one or more operations on every element in the stream. The kernels work in order on a single element on which the same set of instructions is applied. The advantage of this model is that all elements in the stream may be operated on simultaneously. The SIMD (single instruction, multiple data) nature of this model makes it very suited for parallelism. As the number of elements in a stream is usually very high, the amount of parallelism depends largely on the number of processors that can compute on the stream simultaneously. In Figure 1.1, an illustration of a typical graphics pipeline is shown.

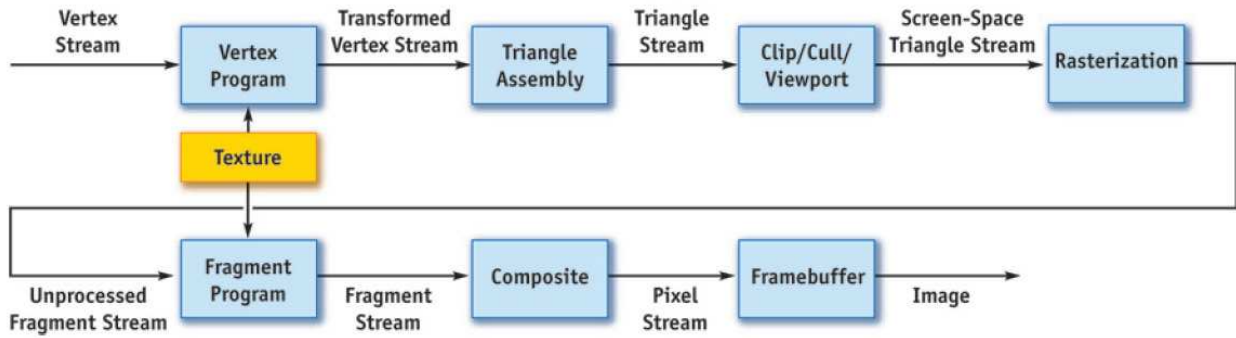


Figure 1.1: The Graphics Pipeline represented by the Stream Model. The stream formulation of the graphics pipeline expresses all data as streams (indicated by arrows) and all computation as kernels (indicated by blue boxes). Both user-programmable and nonprogrammable stages in the graphics pipeline can be expressed as kernels. [28]

### 1.3.2 Vertex and fragment shaders

In the graphics pipeline used by a GPU, there are two kernels that are user-programmable: the vertex and the fragment shader [28]. Any object is defined by a number of 3D points. These points make triangles which are used to describe a surface which, when looked at from a distance, represents a 3D object. The points of these triangles are also referred to as vertices (plural of vertex). The vertex shader allows manipulations to the positions of these vertices. Fragment shaders, receive the pixels in their final positions and may only modify the colour values. This makes fragment shaders highly suitable to, for example, map a simple 2D figure onto a complex 3D geometry, a technique that is known as ‘texture mapping’.

## 1.4 Related work

An increasing amount of research is being conducted on the use of graphics hardware. The large number of programming environments for a GPU suggest that there has been much research in this field. Some key researches are pointed out in this section. The GPU Gems series [5][30] contain a collection of articles on programming the GPU. Among these are detailed suggestions on how to simulate natural effects as water and fire, lighting and shadowing effects, material simulation, image processing and much more.

The GPU has been used for more than just graphics-related calculations. One of the first publications of such applications came from Rumpf and Strzodka. In 2001 they published their findings on performing finite element method computations [37], non-linear diffusion [36] and level set segmentation [35] all using programmable graphics hardware. Other publications have been in the areas of cryptography [4], audio signal processing [40], fluid simulations [16], linear algebra [15] and medical image processing [2].

## 1.5 Overview of this thesis

This thesis will explain how a GPU works and what is needed to run a program on it. This chapter introduced the GPU and its specific architecture. In chapter 2, some of the many GPU programming languages will be explained as well as the historical evolution of those languages. Chapter 3 shows how a simple cellular automaton like the ‘Game of Life’ can be converted to a GPU program. Chapter 4 discusses a more serious algorithm, called the gravitational  $N$ -body problem. A small, but computationally expensive part, of  $O(N^2)$ , will be transferred to the GPU. The entire program may speed up when decreasing the computation time of this small part. Finally, the thesis ends with conclusions and discussion on the achievements, followed by suggestions for future work.

# Chapter 2

## Programming GPUs

As is the case with conventional CPUs, a GPU program is often written in a high level language that abstracts as much from the underlying architecture as possible. This chapter shows a selection of the most important programming environments in chronological order of their introduction. With each language, a sample of a program is shown. There is no direct relation of these sample programs to each other, because the languages differ very much. We distinguish between languages designed to perform graphics related computations and languages that are also designed to do general purpose computations.

### 2.1 Graphics oriented programming languages

The first shading languages were not built to run on a GPU at all. They originated from the need to render scenes for movies and animations. Later, when the programmable GPU allowed real-time rendering, many shading languages erupted with all the same goal: to make the life of the graphics programmer easier.

#### 2.1.1 RenderMan Shading Language

One of the first shading languages was created by Robert Cook from Pixar Animation Studios to work in the RenderMan rendering software [31]. The RenderMan Shading Language was created long before GPUs became programmable and is a so called offline shading language. It is designed to do time consuming rendering of 3D graphics and is still widely in use in the rendering of animations in movies.

The RenderMan Shading Language defines five separate shaders that may be used in rendering a scene. There are surface shaders, light source shaders, volume shaders, displacement shaders and imager shaders. An example of a light source shader is shown here. This shader implements a simple point light.

```
light
pointlight(
    float intensity = 1;
```

```

    color lightcolor = 1;
    point from = point "shader" (0,0,0); )
{
    illuminate(from)
        Cl = intensity * lightcolor / L.L;
}

```

## 2.1.2 PixelFlow

The first attempt to make programmable graphics hardware was done by researchers from the University of North Carolina. They developed an experimental graphics system called PixelFlow to demonstrate the advantages of image-composition architectures and to provide a research platform for real time 3D graphics algorithms and applications [26]. This architecture included a renderer board, a shader board and a framebuffer board each with its own core. The PixelFlow architecture comes with its own shading language and compiler pfman. Pfman is based on the RenderMan Shading Language with a few minor modifications.

## 2.1.3 Real-Time Shading Language

In 2001, Stanford University released its own shading language. Their Real-Time Shading Language (RTSL) is again based on the RenderMan Shading Language described before, but executes, like the name suggests, in real-time [33]. This means that the language is suited for much wider purposes than just offline rendering. The system runs on OpenGL, a specification that defines an interface for writing graphics applications. RTSL can map vertex programs to either programmable vertex hardware, if available, or to the host CPU. Similarly, fragment programs are either mapped to programmable fragment hardware, or to multipass OpenGL. This is an example of a program written in RTSL. It is a surface shader that uses multiple different textures to create, decorate and light a bowling pin.

```

#include "lightmodels.h"
surface shader float4 bowling_ping (
    texref base, texref coated, texref marks, float4 uv)
{
    // Compute per-vertex texture coordinates
    float4 uv_wrap = {uv[0], 10 * Pobj[1], 0, 1 };
    float4 uv_label = {10 * Pobje[0], 10 * Pobj[1], 0, 1};
    // Compute constant texture transformation matrices
    matrix4 m_base = invert(translate(0, -7.5, 0) * scale(0.667, 15, 1));
    matrix4 m_coated = invert(translate(2.6, -2.8, 0) * scale(-5.2, 5.2, 1));
    matrix4 m_marks = invert(translate(2.0, 7.5, 0) * scale(4, -15, 1));
    // Compute per-vertex mask value to isolate front half of pin
    float front = select(Pobj[2] >= 0, 1, 0);
    // Transform texture coordinates, perform texture lookups and apply mask
    float4 Base = texture(base, m_base * uv_wrap);
}

```

```

float4 Coated = (1-front)*texture(coated, m_coated*uv_label);
float4 Marks = texture(marks, m_marks * uv_wrap);
// Invoke lighting models from lightmodels.h
float4 Cd = lightmodel_diffuse({0.4, 0.4, 0.4, 1}, {0.5, 0.5, 0.5, 1});
float4 Cs = lightmodel_specular({0.35, 0.35, 0.35, 1}, {0, 0, 0, 0}, 20);
// Composite textures, apply lighting and return final color
return (Coated over Base) * Marks * Cd + Cs;
}

```

## 2.1.4 Cg: C for graphics

Inspired by the work at Stanford, NVIDIA decided to develop a language of commercial-quality. Led by one of Stanford's RTSL researchers, NVIDIA collaborated with Microsoft on building a language with common syntax and features. C for graphics (Cg) intends to make programming the graphics hardware as simple and intuitive as programming in the general purpose language C [21]. Cg is optimized for working on NVIDIA GPUs and will therefore work less effective on GPUs from other manufacturers like ATI.

## 2.1.5 High Level Shader Language

The High Level Shader Language (HLSL) has the same language syntax and constructs as Cg [29]. The implementation of the two languages started separately, but thanks to Microsoft and NVIDIA working together, the languages moved towards each other, finally resulting in being fully interchangeable. Below, an example of a program written in Cg or HLSL is shown [6]. This fragment shader shows how a 3-dimensional brick wall responds to different positions of a light source.

```

float3 expand(float3 v)
{
    return (v - 0.5) * 2; // Expand a range-compressed vector
}

void C8E2f_bumpSurf(float2 normalMapTexCoord : TEXCOORD0,
                  float3 lightDir          : TEXCOORD1,

                  out float4 color : COLOR,

                  uniform sampler2D normalMap,
                  uniform samplerCUBE normalizeCube)
{
    // Normalizes light vector with normalization cube map
    float3 lightTex = texCUBE(normalizeCube, lightDir).xyz;
    float3 light = expand(lightTex);
    // Sample and expand the normal map texture
    float3 normalTex = tex2D(normalMap, normalMapTexCoord).xyz;
}

```

```

    float3 normal = expand(normalTex);
    // Diffuse lighting
    color = dot(normal, light);
}

```

### 2.1.6 OpenGL Shading Language

The OpenGL Shading Language (GLSL or GLslang) was created by the OpenGL Architecture Review Board to give developers more direct control of the graphics pipeline [34]. The language itself is very similar to Cg or HLSL, the difference lies in compatibility. GLSL runs on most languages that support OpenGL, but is not compatible with the other large graphics application interface, DirectX. Here is an example of a program that compute the light intensity written in GLSL.

```

varying vec3 normal;

void main()
{
    float intensity;
    vec4 color;
    vec3 n = normalize(normal);

    intensity = dot(vec3(gl_LightSource[0].position),n);

    if (intensity > 0.95)
        color = vec4(1.0,0.5,0.5,1.0);
    else if (intensity > 0.5)
        color = vec4(0.6,0.3,0.3,1.0);
    else if (intensity > 0.25)
        color = vec4(0.4,0.2,0.2,1.0);
    else
        color = vec4(0.2,0.1,0.1,1.0);

    gl_FragColor = color;
}

```

## 2.2 Generic oriented programming languages

Very quickly, researchers who wanted to exploit the computational power inherent in GPU, realized that GPUs are capable of more than just graphics oriented computations, as will be elaborated more on in chapter 3. Therefore, programmers without any experience in graphics programming wanted to use the GPU for their applications. In the before mentioned languages, the programmer is assumed to have basic graphics programming knowledge. By

using the stream processing capabilities of the GPU, these languages aim to provide a more intuitive approach for non-graphics programmers.

### 2.2.1 Sh

Sh is a free and open source metaprogramming language build on top of C++ [22]. The Sh program can be compiled by a C++ compiler with the Sh library to create an executable. When this executable is run, the Sh compiler compiles the shader to GPU assembly code. This code may be run immediately or stored for later use. The advantage of this approach is that shaders written in Sh may share parameters with its host program. Sh is suitable for both graphical and general purpose computations. This is a diffuse light vertex shader.

```
vsh = SH_BEGIN_VERTEX_PROGRAM {
    ShInputNormal3f normal;
    ShInputPosition4f p;

    ShOutputPoint4f ov;
    ShOutputNormal3f on;
    ShOutputVector3f lvv;
    ShOutputPosition4f opd;

    opd = Globals::mvp | p;
    on = normalize(Globals::mv | normal);
    ov = -normalize(Globals::mv | p);
    lvv = normalize(Globals::lightPos -
        (Globals::mv | p)(0,1,2));
} SH_END_PROGRAM;
```

### 2.2.2 Brook

Brook for GPUs is a system designed to perform general-purpose computing on programmable graphics hardware [3]. The goal of Brook is to expose the Stream Programming Model, so that programming the GPU becomes even more like programming a CPU. The Brook compiler, transforms the Brook code to Cg/HLSL code which is in turn compiled to execute on the GPU. This is an example of how a kernel is used in Brook to calculate on vectors.

```
kernel void saxpy (float a, float4 x<>, float4 y<>,
                  out float4 result<>) {
    result = a*x + y;
}

void main (void) {
    float4 X[100], Y[100], Result[100];
    float4 x<100>, y<100>, result<100>;
    ... initialize a, X, Y ...
}
```

```

streamRead(x, X);           // copy data from mem to stream
streamRead(y, Y);
saxpy(a, x, y, result);    // execute kernal on all elements
streamWrite(result, Result); // copy data from stream to mem
}

```

## 2.3 Selection and discussion

We have shown only a selection of the many programming environments and shading languages available. As stated before, the remainder of this thesis will focus on using the GPU for general purpose computing. It would therefore make sense to choose Sh or Brook as the language to work in. Yet, we choose to write our programs in Cg. Together with HLSL and GLSL, Cg is the most common used language for writing shaders. Though Cg was designed as a graphics language, it may just as well be used for generic computing. Brook even translates its program to Cg, and will therefore end up slower. Similarly, Sh hides much overhead in creating a proper GPU program, but this will result in a closed program, where it is hard to see what is done. Although Cg requires more preparation in the host program, it provides a more transparent program. Another reason to choose Cg above the other graphics oriented languages is for the large amount of documentation available. All of the boards used in the tests are manufactured by NVIDIA, and as Cg is build by NVIDIA, it is optimized to run on these boards. Finally, Cg is, unlike HLSL, platform independent and will therefore allow programs to be run on other hardware as well.

## 2.4 An example GPU program

Here, an example program is presented that implements both a vertex and a fragment shader. The goal of this simple program is to visualize a given set of particles. The idea is that, when looked at from a distance, a cluster of particles looks like a sphere, but when zooming in, the particles become separate spheres themselves. The CPU part is not shown here, but in short, it does the following:

- Read the data containing temperature, 3D position and 3D velocity vectors.
- Initialize the environment for both shaders.
- Provide an interface to view the points in space and time.
- Send a sphere-like image (texture) to the GPU.

The first step towards a smooth looking visualization happens in the CPU part of the program. Simple OpenGL statements are used to draw the particles. To visualize the temperature as well, the particles are given colours ranging from blue to red for cooler (outer) and warmer (inner) particles. Furthermore included in this visualization is a time

variable. Without a separate GPU program, this should be done by modifying the positions at each step and then sending the new positions back to the GPU. This program eliminates this communication by performing it on the GPU. The program uses a parameter which is normally reserved to define the normal of a vertex to put this information in. A normal is usually used to deal with lighting and shadows, but is nothing more than a 3-dimensional vector. As there is no need to deal with lighting and shadows in this program, this parameter can be used for storing velocity in the vertex program. The colour (temperature) is already defined in the host program, so it may simply be passed through. Then the position is modified using the old position, the velocity and the time step using Equation 2.1.

$$\mathbf{x}_t = \mathbf{x}_0 + \mathbf{v}t \quad (2.1)$$

Variable ‘t’ is a time value that the user may change using a graphical user interface. Undefined vertex programs pass through colours and normals automatically. They also multiply the position vector with the model view projection matrix, to set the viewpoint correctly. The next programmable stage in the graphics pipeline is the fragment program. The host program draws square points with an increased point size. This results in big squares overlapping each other. The fragment program, shown in Figure 2.2 multiplies each square with a given image of a filled circle. Still, the points overlap, because the squares are drawn front to back. In this case, only the part containing the circle to be drawn should be drawn. Therefore, an alpha value is assigned to all pixels in all particles. This alpha value may be used to indicate the amount of transparency of an object. In this case the ‘circle’ part must be completely opaque, which corresponds to an alpha value of 1, while the rest must be completely transparent, which corresponds to an alpha value of 0. Screenshots of resulting images are shown in Figures 2.3, 2.4, 2.5 and 2.6.

```

struct appin
{
    float4 myPosition : POSITION;
    float3 myNormal   : NORMAL;
    float4 myColor    : COLOR0;
    float2 coords     : TEXCOORD0;
};
struct vertout
{
    float4 HPOS       : POSITION;
    float4 COLO       : COLOR0;
    float2 coords     : TEXCOORD0;
};
vertout main(appin IN,
             uniform float step,
             uniform float4x4 ModelViewProj)
{
    vertout OUT;
    float4 position;
    float3 velocity;
    // apply the colour given in the host program
    OUT.COLO = IN.myColor;
    // apply the position given in the host program
    position = IN.myPosition;

    // use the velocity (stored as a normal) to modify the position
    // using step as a time step parameter.
    velocity = IN.myNormal;
    position.xyz = position.xyz + step*velocity;

    // multiply with the modelview projection matrix to get the proper
    // camera view and copy the coordinates
    OUT.HPOS = mul(ModelViewProj, position);
    OUT.coords = IN.coords;
    return OUT;
}

```

Figure 2.1: Example vertex shader for the calculation of the new position of a particle based on its velocity vector and integration time step (see Equation 2.1). This shader is implemented in Cg.

```

struct fragment
{
    float4 color0    : COLOR0;
    float2 coords    : TEXCOORD0;
};

struct pixel
{
    float4 color : COLOR;
};

pixel main(fragment IN, uniform sampler2D texture)
{
    pixel OUT;
    float4 c;

    // retrieve the required value from the external texture
    c = tex2D(texture, IN.coords);
    if(c.r>0.5)
    {
        // use the texture value to modify to colour (bright on
        // the inside, pale on the outside) and make the fragment
        // completely opaque
        OUT.color = c*IN.color0;
        OUT.color.a = 1.0;
    }
    else
    {
        // make the fragment completely transparent
        OUT.color.a = 0.0;
    }

    return OUT;
}

```

Figure 2.2: Example fragment shader that maps an external texture on each particle. Any value in the 2D sphere texture lower than 0.5 is made transparent.

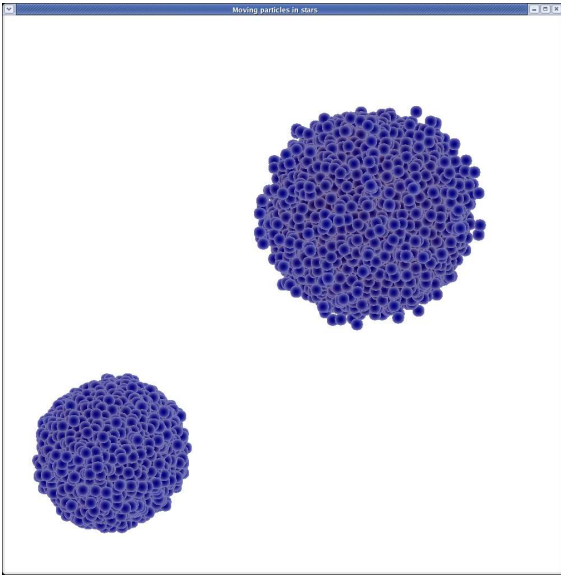


Figure 2.3: The initial position of the particles. The two clusters look like spheres from here.

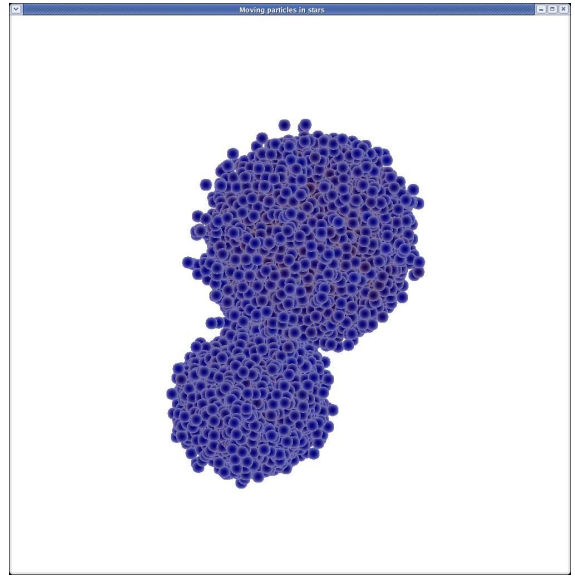


Figure 2.4: Using the velocity of the particles, a prediction of a future step can be made.

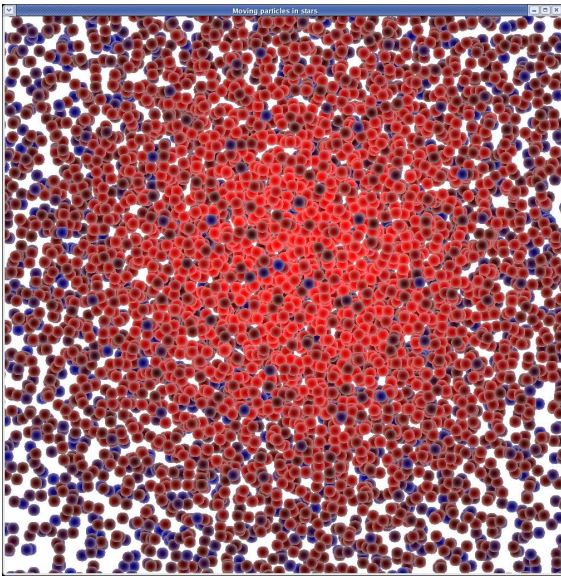


Figure 2.5: When zoomed in on a cluster, the separate particles are recognized. Also, the warmer inner particles show up.

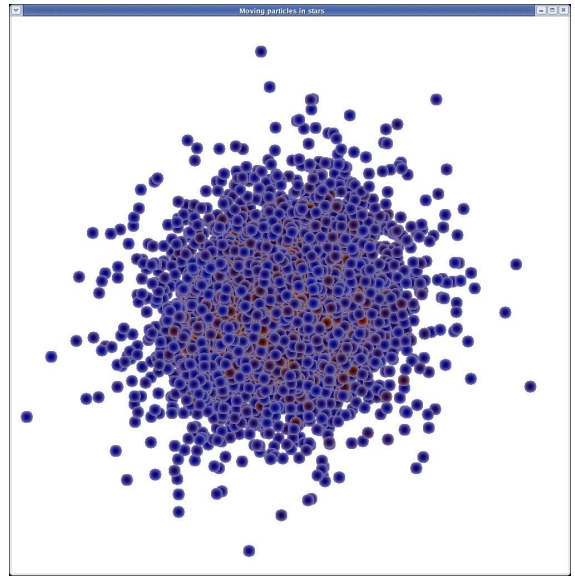


Figure 2.6: This shows the results of viewing too far in time: the clusters 'explode' and no longer give a serious representation.

# Chapter 3

## Generic computing on a GPU

### 3.1 Introduction

The inherent computational power in modern GPUs greatly exceeds that of modern conventional CPUs. As explained in the previous chapters, this was made possible by using efficient pipelines and, later, multiple processors. While CPUs only recently started using dual and quad cores processing units, the latest GPU contains a total of 128 processors. This gives a state of the art GPU like the NVIDIA GeForce 8800 GTX a much higher peak performance than modern day CPUs. Because GPUs became programmable, users could specify what kind of operations they wanted to do on the input streams. They could customize lighting, shading, and simulate complicated objects like water or fire.

It did not take long for GPU programmers to realize that there were more possibilities in using a GPU than just graphics computations. The input data that was used now, were simply floating point numbers. And the operations were nothing more than manipulations on these numbers. These numbers usually represent colours, positions and textures, but this is no restriction. These numbers could just as well stand for anything else. Using the GPU for computations other than for merely graphical purposes, is called general purpose computing on the GPU, or GPGPU. Of course not just any general purpose operation is suitable to be executed on a GPU. While a simple addition of two integers can be done perfectly well on the GPU, the time to prepare the GPU and upload the numbers would take much longer than the entire computation would take on a CPU. Also, GPUs are built to perform operations parallel on streams, which means they are more efficiently used when there are more elements in a stream. An addition of two large vectors can be more efficiently on a GPU. As computation is done faster on the GPU, it performs relatively better compared to a CPU when the vector size increases.

In this chapter, a practical example is given that explains how to convert a simple program to a GPU based program. This same example also demonstrates how generic computing may be implemented on a GPU. The Game of Life is used as the example program [7]. The Game of Life is a very simple cellular automaton with only a few rules. This enables a very efficient implementation on a GPU. In section 3.2, the Game of Life and its rules are introduced.

Then this section will present an implementation on a CPU and a detailed description on how to convert this to a GPU based implementation. In section 3.3, the performances on multiple different GPUs are compared to the CPU based version. Finally the results are discussed in section 3.4.

## 3.2 Conway's Game of Life

In 1970, mathematician John Conway thought of a recreational, zero player game where one can watch the evolution of living cells on an infinite 2-dimensional plane. Each cell has two states: off (dead) or on (alive). With a few simple rules, cells are determined either to be born, die, survive or remain dead. Because of these simple analogies with living organisms, the 'game' was called 'life'.

The cells in the 2-dimensional lattice interact with their eight neighbours, four adjacent and four diagonal. Interaction is limited by simply 'knowing' whether they are alive or not. The cells are updated according to the following rules:

1. Any living cell with fewer than two neighbours dies of loneliness.
2. Any living cell with more than three neighbours dies of crowding.
3. Any dead cell with exactly three neighbours comes to life.
4. Any living cell with two or three neighbours lives, unchanged, to the next generation.
5. Any other cell remains dead.

At each discrete time step, these rules are simultaneously applied to all cells in the lattice.

### 3.2.1 A CPU implementation

Before implementing this on a GPU, a straightforward CPU version of this cellular automaton is presented. This version will be used to validate the results of the GPU version and to compare perform. As it is impossible to simulate an infinite lattice, a square lattice of different sizes is used here. At each border of these squares, there is an extra row or column of cells to keep the design as simple as possible. These rows and columns will be read from, but not written to in the calculation of the new time step, and thus remain dead. Below, the entire algorithm is shown in one simple loop. To make sure that all cells are updated synchronously, the new results are stored in a temporary lattice until the time step completes.

```

for(n=0;n<steps;n++)
{
  for(i=1;i<xSize+1;i++)
    for(j=1;j<ySize+1;j++)
    {
      // count the number of living neighbours
      neighbours =
        lattice[i-1][j-1] + lattice[i-1][j] +
        lattice[i-1][j+1] + lattice[i][j+1] +
        lattice[i+1][j+1] + lattice[i+1][j] +
        lattice[i+1][j-1] + lattice[i][j-1];

      if(neighbours > 3 || neighbours < 2)
        tempLattice[i][j] = 0; // Game of life rule 1 and 2
      else if(!field[i][j] && neighbours == 3)
        tempLattice[i][j] = 1; // rule 3
      else if(field[i][j] && neighbours >= 2 && neighbours <= 3)
        tempLattice[i][j] = 1; // rule 4
      else
        tempLattice[i][j] = 0; // rule 5
    }

  for(i=0;i<xSize;i++)
    for(j=0;j<ySize;j++)
      lattice[i][j] = tempLattice[i][j];
}

```

### 3.2.2 A GPU implementation

The conversion of this simple algorithm to a version that runs on a GPU, requires a substantial amount of additional code. First, the program consists of two parts. One executes on the CPU which functions as a host for the GPU. The CPU program creates a graphics context that provides access to the GPU hardware. The second part runs on the GPU and is implemented in Cg. For building the environment, we used GPGPU tutorials [8] and [12] to help us get started. These tutorials shows step by step what needs to be done to set up a simple GPU program. The Cg implementation of the Game of Life is straightforward and can be compared directly to the CPU version:

```

float4 life(in float2 coords : TEX0,
           uniform samplerRECT texture,
           uniform float offsetX,
           uniform float offsetY) : COLOR
{
    float4 neighbours;

    // retrieve the own status and that of all 8 neighbours
    float4 c = texRECT(texture, coords);
    float4 bl = texRECT(texture, coords + float2(-offsetX, -offsetY));
    float4 l = texRECT(texture, coords + float2(-offsetX, 0));
    float4 tl = texRECT(texture, coords + float2(-offsetX, offsetY));
    float4 t = texRECT(texture, coords + float2(0, offsetY));
    float4 ur = texRECT(texture, coords + float2(offsetX, offsetY));
    float4 r = texRECT(texture, coords + float2(offsetX, 0));
    float4 br = texRECT(texture, coords + float2(offsetX, -offsetY));
    float4 b = texRECT(texture, coords + float2(0, -offsetY));

    // count the number of living neighbours
    neighbours = bl + l + tl + t + ur + r + br + b;

    // Game of life rule 1 and 2
    if(neighbours.r > 3 || neighbours.r < 2) c.r = 0;
    // rule 3
    else if(!c.r && neighbours.r == 3) c.r = 1;
    // rule 4
    else if(c.r && neighbours.r >= 2 && neighbours.r <= 3) c.r = 1;
    // rule 5
    else c.r = 0;

    return c;
}

```

The Cg program contains only the direct instructions for the GPU. Communication to the GPU and execution is done from the CPU. There are two application programming interfaces (APIs) that can be used for launching a Cg program: OpenGL and DirectX. We have chosen for OpenGL as DirectX does not execute on Linux systems. Building the environment starts with initializing the OpenGL Utility Toolkit (GLUT) which creates a graphics ‘window’ [38]. This window is normally used to display the resulting image calculated within the GPU. Depending on the computer settings, the colours in this window are limited to a maximum of 32 bits per pixel. This implies 8 bits per red, green, blue and alpha channel in the RGBA colour mode. This yields a maximum of  $2^{24}$  ( $\approx 16$  million) different colours<sup>1</sup>, which is more than enough to make two neighbouring colours indiscernible to the human

---

<sup>1</sup>The alpha channel does not contribute to the colour itself, but to its transparency.

eye. Unfortunately, this 8-bit precision is by far insufficient for most scientific production calculations. The GPU itself is capable of performing operations on data of 32-bit precision. This data will be clamped when it is shown on screen. Since it is not necessary to show the data as output, the framebuffer is avoided entirely. Instead, an off-screen buffer is used called a framebuffer object (FBO). This off-screen framebuffer is not clamped and can thus contain numbers in 32-bit notation. These 32-bit floating point numbers are stored in the IEEE 754 single-precision standard format [11]. Current graphics hardware arithmetic is also very similar to the arithmetic specified by the IEEE 754 standard<sup>2</sup>.

The strength of a GPU lies in its ability to perform many calculations very fast. Its weakness, compared to a CPU, is communication. The Game of Life is therefore perfectly suited since all calculation happens within the lattice. The only communication needed, is sending an initial lattice where the program starts with. To calculate the next time step, the host program now needs to draw a fully filled square. As the border rows and columns must remain dead, these must be excluded in the rendering of this square.

The 2-dimensional initial lattice of 0's and 1's is converted to a 2-dimensional texture. This texture is then presented as input to the GPU. The output of a time step is always used as input for the next time step, but unfortunately, the GPU is unable to read and write texture at the same step. Therefore, a second texture is defined and both textures are attached to separate points in the FBO. These attachment points are alternating read/write and write/read, so the textures may remain in the same piece of memory. This technique is actually the same as used in computer graphics double buffering. Finally, to get the output, the data in the texture that was last written to is retrieved.

### 3.3 Performance

We compare the performance of the Game of Life on a CPU with several GPUs. The CPU is an Intel Xeon 3.40 GHz, and the GPUs are NVIDIA's Quadro FX 1400, Quadro FX 4000, GeForce 6800 and the current flagship, GeForce 8800 GTX. The measurements were done on a  $N * N$  square lattice where  $N$  increases from 128 to 1024 in powers of 2. Additionally, between every two steps, another step is made. In every lattice, the same initial state is used. This state, also called the R-pentomino, is a known unstable state (i.e. it continues to expand in an infinite lattice<sup>3</sup>). From this state, the Game of Life rules are applied 1000 times, (or: 1000 time steps are made). The results are shown in Figure 3.1.

### 3.4 Conclusion and discussion

As can be seen from Figure 3.1, all of our tested GPUs are faster than our CPU. The  $O(N^2)$  characteristic of the algorithm is very clear, not only for the CPU, but also for the GPUs,

---

<sup>2</sup>Some rounding is done slightly differently, and denormals are typically flushed to zero.

<sup>3</sup>In a finite lattice, as used here, the R-pentomino will stabilize after a while, but the effect on our measurements is negligible

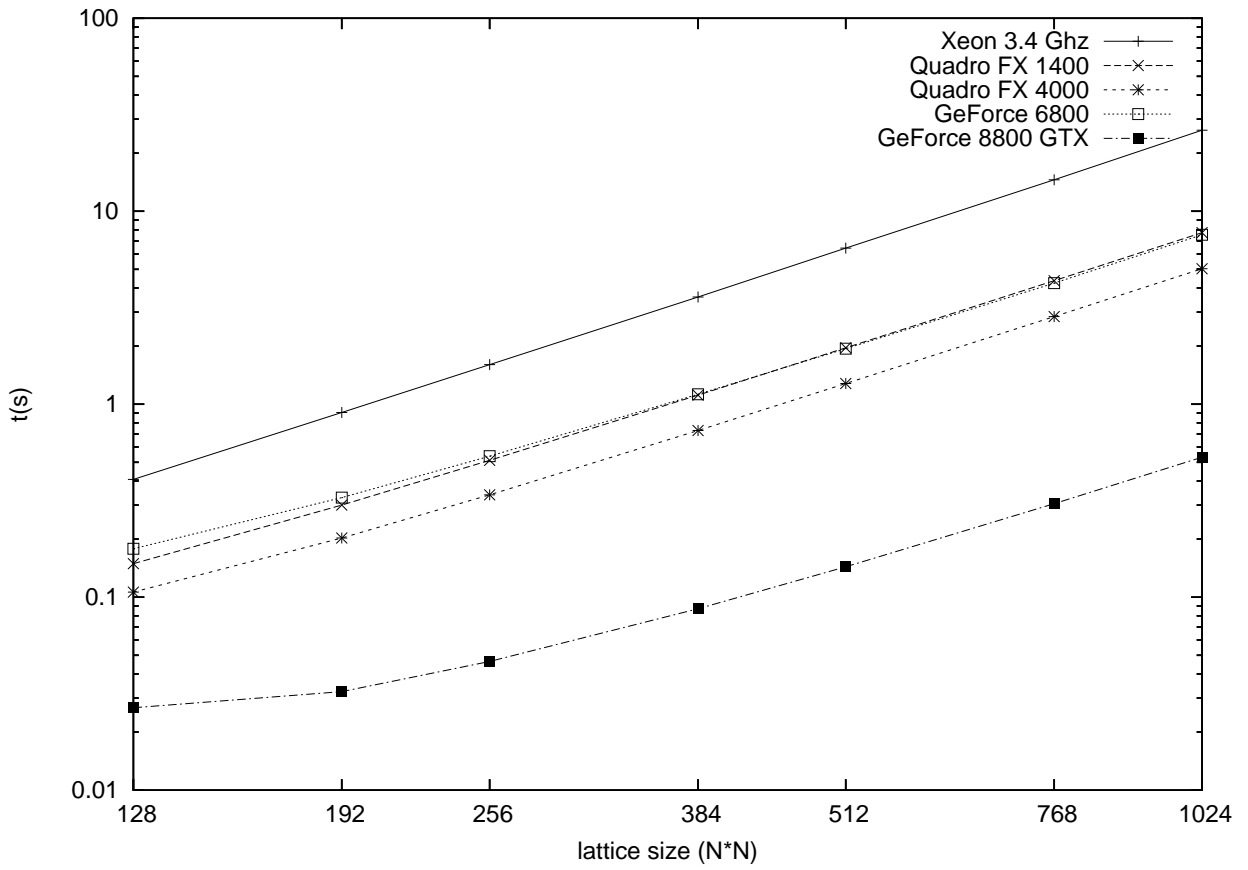


Figure 3.1: Performance comparison of the Game of Life on GPUs and CPU for 1000 time steps.

as indicated by 5 parallel lines. The relative speed can now be determined by comparing the different computing units with each other. The Quadro FX 1400 and the GeForce 6800 perform very similar (both are  $\sim 3.4$  times faster than the CPU), except that the GeForce 6800 has slightly more startup problems. The Quadro 4000 is a newer version, this shows in the results: over 5 times faster than the CPU. The newest GPU is by far the fastest of all. It performs 1000 time steps in over 1 million cells in a little more than half a second, approximately 50 times faster than on the CPU. These are of course very impressive results. Unfortunately, not all applications are as well suited to execute on the GPU as the Game of Life. In order to get an indication of how more serious problems behave on the GPU, the next chapter will show the design, implementation and performance of a gravitational  $N$ -body problem.

# Chapter 4

## High performance direct gravitational $N$ -body simulations on Graphics Processing Units

The previous chapters explained how GPUs work and how they can be used to speed up the Game of Life. This chapter gives another, more realistic example of how to use a GPU for generic computing. This chapter is largely based on a paper by Portegies Zwart, Belleman and Geldof accepted for publication in *New Astronomy*<sup>1</sup> [41].

### 4.1 Introduction

In astronomy, star clusters can be seen as a separate group of stars. The stars interact with each other through the gravitational force. A cluster with  $N$  stars can be represented as an  $N$ -body system [14]. Each body stands for a simple point-like star with some mass. Simulating the evolution of a star cluster is now similar to finding the solution of the gravitational  $N$ -body problem. This problem describes the movement of  $N$  particles through space where each particle interacts with every other particle. Particles closer to each other will have a greater impact, as do heavier particles. The  $N$ -body problem is analytically solved for  $N = 2$ , but as star clusters consist of many stars, a numerical method is needed. Such a method will not give an exact solution, but rather an approximation. This chapter introduces the  $N$ -body problem and our approach to make a GPU based solver for it. Section 4.2 describes the approach used to calculate the  $N$ -body problem. There are different time step schemes that can be used in this approach. Three of these are listed in section 4.3. Section 4.4 treats the prediction and correction of the positions and velocities briefly, while section 4.5 elaborates on how the accuracy of the entire system is checked. The GRAPE special purpose computer is introduced in section 4.6. Section 4.7 shows in detail what is needed to implement a GPU based program. Section 4.8 shows the performance of this GPU program compared to the

---

<sup>1</sup>A preprint of this article is available from <http://xxx.lanl.gov/abs/astro-ph/0702058>.

CPU version and the GRAPE version. This chapter concludes with a short discussion on the approach and results.

## 4.2 Direct summation

Different techniques exist to calculate the gravitational evolution of an  $N$ -body system. The most straightforward and accurate is that of direct summation. In this approach, the force exerted on each particle  $\mathbf{F}_i$  is computed by summing up the contributions of all other particles. This contribution is equal to the Newtonian force,  $\mathbf{F} = m\mathbf{a}$ . The formula for direct summation is shown in Equation 4.1.

$$\mathbf{F}_i \equiv m_i \mathbf{a}_i = m_i G \sum_{j=1, j \neq i}^N m_j \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}. \quad (4.1)$$

Here,  $m_j$  is the mass of particle  $j$  and  $\frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}$  is the relative acceleration of particle  $j$  related to particle  $i$ , where  $\mathbf{r}_i$  is the position of particle  $i$ .  $G$  is the Newton constant. Since this summation has to be done for all particles, the computational complexity of this algorithm is  $O(N^2)$ .

## 4.3 Time step schemes

Equation 4.1 updates a particle from an old time step, to a new one. Different schemes exist that try to keep a balance between minimizing the error and keeping a reasonable speed.

### 4.3.1 Shared time step scheme

In the simplest form, all particles are updated at each time step. This is also called the shared time step scheme. Besides being the simplest time step scheme, this form is also very suitable for parallel calculation. In that case, each particle can be executed on a different processor. Besides being the simplest of all direct summation forms, it is also the most accurate. A disadvantage of this method is that it does much unnecessary computation for slower particles.

### 4.3.2 Individual time step scheme

Instead of performing the force calculation for all particles in each time step, the algorithm can be optimized by using individual time steps [39]. The particles (or stars) each have a very different speed, acceleration and higher time order derivatives. One particle may travel extremely slow, while the other may travel extremely fast. Obviously, fast travelling particles need to be updated very often, and should therefore have a small time step size. If all particles are updated at each time step, this would mean that slow travelling particles change very little. When all particles have their own time step, only the fast travelling

particles will have to be updated very often. This saves a lot of computation, while the error rate is kept at a minimum.

### 4.3.3 Block time step scheme

The block time step scheme groups time steps into powers of two [17]. This creates a limited number of groups with multiple particles in it, making it again suitable for parallelization. As chapter 2 already indicated, parallelization is a crucial element in GPUs as well. The block time step scheme is therefore also for the GPU, the method of choice.

## 4.4 Prediction and Correction

After the calculation of force on a particle, the position needs to be updated. For this, all particles are needed. Also, all other particles need to be in the same time step. The fourth time-order ‘Hermite’ predictor-corrector scheme is used to predict these positions [18].

In this scheme the position ( $\mathbf{x}$ ) and velocity ( $\mathbf{v} \equiv \dot{\mathbf{x}}$ ) is calculated to the fourth order using the acceleration ( $\mathbf{a} \equiv \dot{\mathbf{v}}$ ) and jerk ( $\mathbf{k} \equiv \dot{\mathbf{a}}$ ). First, the positions and velocities of all particles are predicted using the acceleration and the jerk. Depending on the time step, this is typically a good way to approximate the actual data. After the prediction step, the acceleration and jerk are computed for the particles that need to be updated. Then a correction is made using the estimated higher order derivatives. Finally, this leads to the new position and velocity at time  $t + dt$ .

## 4.5 Accuracy of the implementation

Determining the accuracy of the implemented algorithm, is difficult. Since the problem has no analytical solution, there is no way to be certain what the error of any method is. Even when a result is compared to the ‘correct’ answer, a small error in the acceleration or jerk will grow exponentially in time in the position and velocity. A better way to check the accuracy of the system is to measure the energy of the system. The law of conservation of energy states that the total energy of any system may not change over time. Therefore, the energy of the system is measured before and after the calculations and the absolute value of the difference divided by the initial energy is taken as the error of the system ( $\frac{|\Delta E|}{E}$ ). Unfortunately, this means that extra computations are needed.

## 4.6 The GRAPE special-purpose computers

In the late 1990s, a breakthrough in direct-summation  $N$ -body simulations came, with the development of the GRAPE series of special-purpose computers, which achieved a spectacular increase in performance [19]. The speedup of the GRAPE is based on implementing the entire algorithm in hardware and placing many force pipelines on a chip. The latest

GRAPE-6, is still the fastest method for computing the direct-summation  $N$ -body simulation [20]. As the GRAPE uses the same block time step scheme, it can be used to compare the performance of our GPU method.

## 4.7 Implementation on a GPU

Mapping the  $N$ -body force calculation onto a GPU is not trivial. To exploit the computational power effectively, the mapping needs to be carefully designed. The strength of a GPU lies in its ability to perform many calculations in parallel. As explained in the previous chapter, the GPU program has to be launched from a CPU program. This GPU program will only perform the force calculation part of the algorithm. Again, input parameters are set using the graphics entities supported by Cg.

Unfortunately, the force calculation requires constantly evolving data as input. Each time the force calculation is required, all data that is required for this calculation has to be sent over as well. Particle data arrays are represented as 2-dimensional ‘textures’. Textures are normally used to represent pixel colour information with one single component (luminance, red, green, blue or alpha), three components (red, green and blue) or four components (red, green, blue and alpha). This can be used to our advantage when making textures with more than one value per particle, like the position or acceleration. The elements of both input and output textures will all be 32-bit floats. The required input textures are:

- mass (N)
- position (3N)
- velocity (3N)

and as output:

- acceleration (3N)
- jerk (3N)
- potential (N)

As indicated before, both acceleration and jerk need to be calculated. In order to monitor the error of the entire system, the potential energy is computed at each update. There is now a 7-dimensional output (3 for the mass, 3 for the acceleration and 1 for the potential energy) which does not fit in the standard maximum 4-dimensional RGBA output. A technique called multiple render targets (MRT) is used to allow multiple attachments with the framebuffer to be written.

Using an integer input value, the GPU program cycles through all particles that need to be updated and add the result to the output textures. Unfortunately, each texture can be either read-only or write-only, but not both at the same time. For the addition of the

force of a particle, we need the old value (read) to add the new force (write). Therefore the output textures (acceleration, jerk and potential) are represented by a double-buffered scheme, where after each execution of the GPU program, the textures are swapped between reading and writing.

In Appendix A, the GPU part of the implementation is shown. The CPU part is shown here:

1. Read all initial input (Mass, position, velocity, end time, etc.).
2. Initialize the GPU.
3. Send the mass, position and velocity of all particles to the GPU.
4. Let the GPU calculate the acceleration and jerk of all particles.
5. Retrieve the acceleration and jerk from the GPU.
6. Compute the initial time step for all particles.
7. Sort the particles according to the time steps in ascending order.
8. Update the time with the smallest time step (the time step of the top particle).
9. Search all particles with the same time step (note that because the particles are sorted by time step, that one index number which points to the lowest particle in the list is sufficient to know all particles that should be updated).
10. Predict the positions and velocities of all particles at the current time.
11. Send the masses, positions and velocities to the GPU along with the index value pointing the lowest particle in the list.
12. Let the GPU calculate the acceleration and jerk of all particles with an index lower than or equal to the given index value.
13. Retrieve the acceleration and jerk from the GPU.
14. Correct the positions, velocities and time steps of the updated particles using the acceleration and jerk.
15. Repeat from 7 until time  $\geq$  end time.
16. Finalize the GPU and CPU.

## 4.8 Performance

In this section we present the performance measurements done by executing the program on a typical  $N$ -body problem. The performance on two GPUs (NVIDIA Quadro 1400 FX and NVIDIA Quadro 8800 GTX) is compared to the performance on a CPU (Intel Xeon 3.40 GHz) and to the GRAPE-6Af. A Plummer model is run for 0.5 time unit [32][13]. To avoid startup problems, measurements start from 0.25 time unit.

Sometimes it may occur that two particles come very close to each other and start interacting with each other very rapidly. This behaviour is very undesirable here, as it may cause a tremendous amount of computation, which makes the performance test incomparable to other tests. For this reason, a softening parameter of  $\eta = 1/256$  is used to prevent two particles from getting too close. The time step is calculated as a function of the acceleration

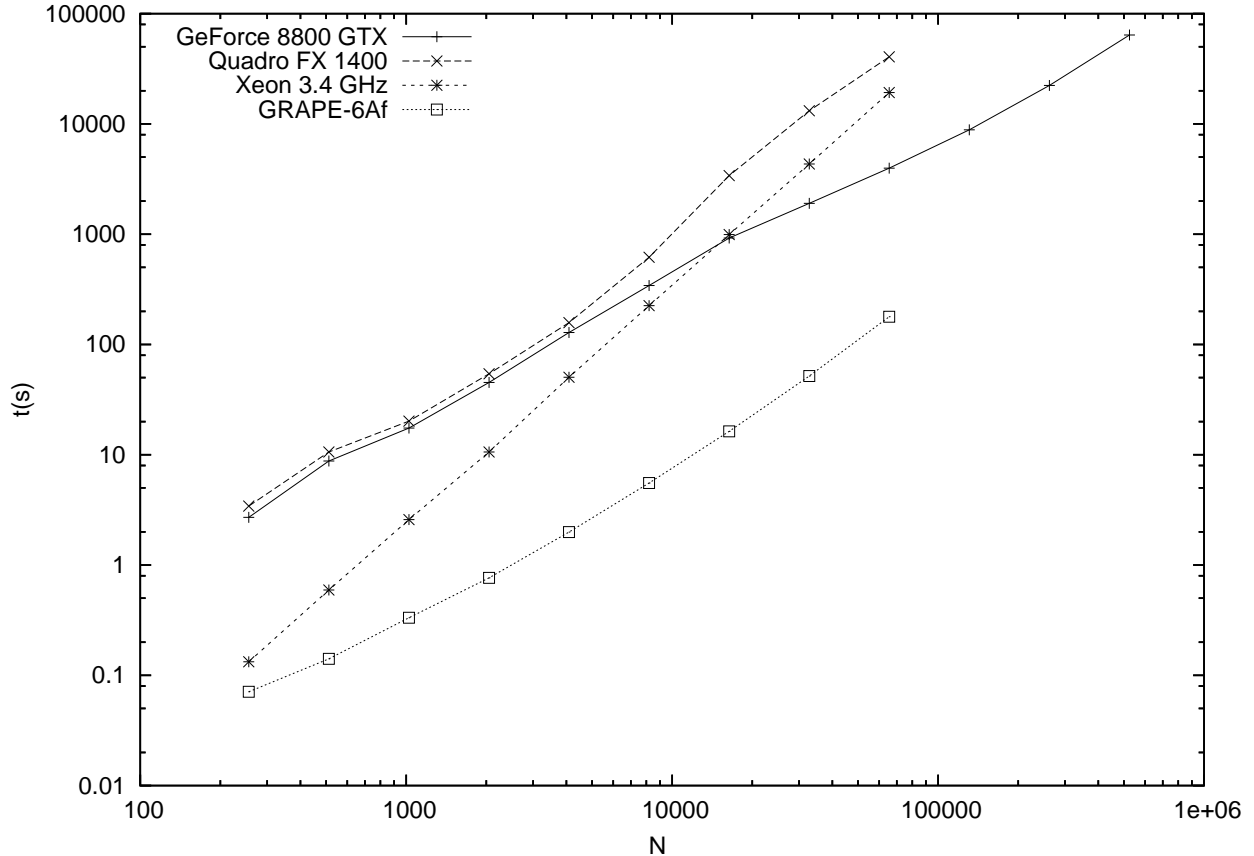


Figure 4.1: Performance comparison of the  $N$ -body problem on two GPUs, a CPU and a GRAPE-6Af.

and jerk, multiplied by a constant of 0.01. To assure that all particles are evaluated at least twice during the measurement, a maximum time step of 0.125 is chosen. The program is run with the number of particles (or bodies) varying from 256 to more than half a million in steps of powers of two. The simulation on the Quadro 1400 FX and the CPU was stopped at 64 thousand particles, as the measurements became very time consuming. The GRAPE was stopped at the same amount of particles because it lacks the memory required to store more particles. The results are shown in Figure 4.1.

#### 4.8.1 Precision

All tests were done in single 32-bit accuracy, because that is the highest accuracy a GPU can use natively (we will discuss GPU accuracy further in the next chapter). Both the CPU and the special purpose GRAPE computer calculate in double (or 64-bit) accuracy. To get an impression of the impact that calculating in single precision will have, we compare the error of the GRAPE in double precision to the error of the GPU in Figure 4.2. In order to show this error, the time step control parameter  $\nu$  is varied from 0.1 (fast, but inaccurate)

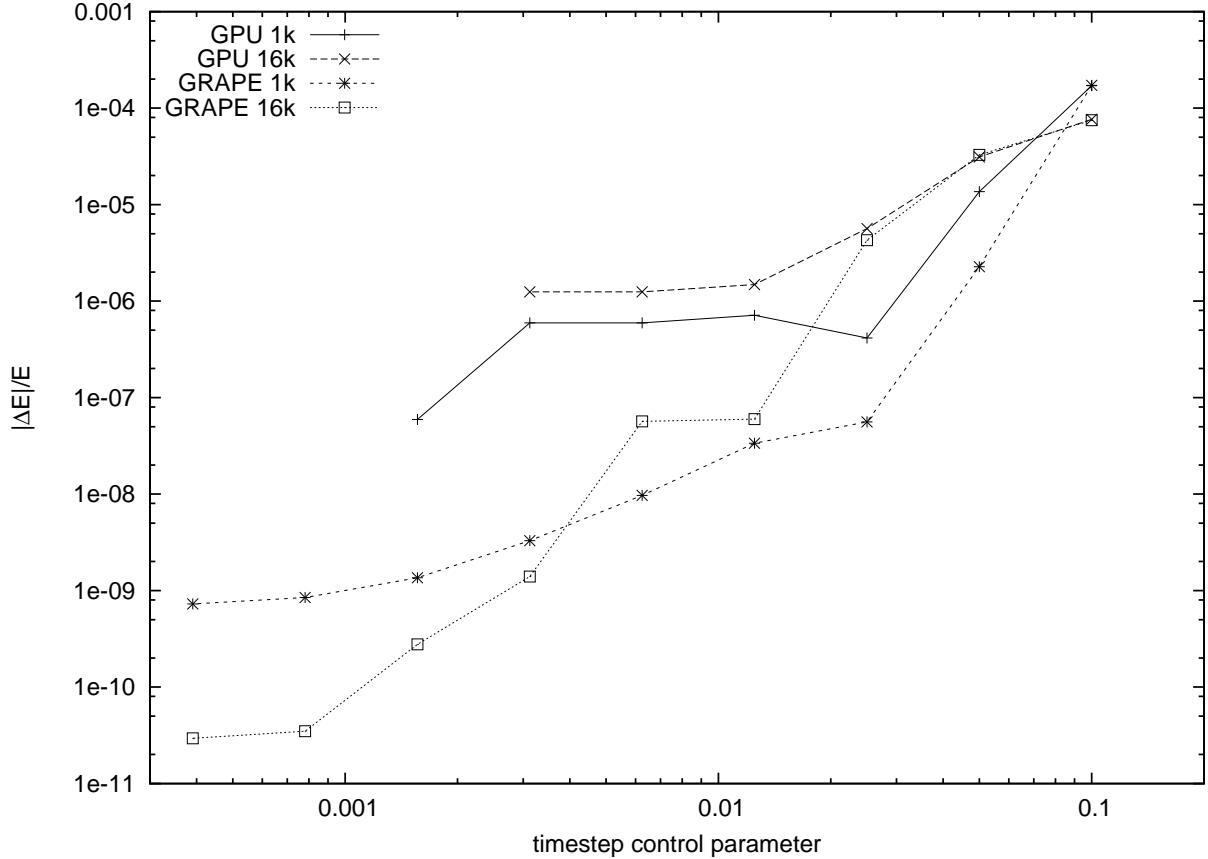


Figure 4.2: Comparison of the error on both GPU and GRAPE for  $N = 1024$  and  $N = 16384$  particles.

to  $0.1/2^8$  (very slow, but accurate). This is done for  $N = 1024$  and for  $N = 16384$ .

## 4.9 Conclusion and discussion

It can be clearly seen in Figure 4.1 that both GPU versions start out quite slow compared to both the CPU and the GRAPE versions. When the number of particles grows, the performance of the GPUs improve compared to the CPU version. This can be explained by the block time step scheme. With a low number of particles, the average block size which is sent to the GPU is also low. This means that all particle positions and velocities have to be sent to the GPU to calculate only one or several particles. The communication time is thus very high relative to the computation time. When the number of particles increases, the average number of particles in a block also increases. The communication time remains the same, but is relatively lower compared to the computation time. As the GRAPE is built for this algorithm, the communication overhead is much lower, which can be seen in Figure 4.1. Computation in the GeForce 8800 GTX appears to be even slightly faster on the GPU

than on the special purpose GRAPE computer. The price we have to pay for this gain in speed can be seen from Figure 4.2. The error in single precision is at least three orders of magnitude bigger than the error when calculation in double precision.

The limit on the number of particles that can be stored is much higher on the GPU than on the GRAPE. While the GRAPE used in our tests can not go further than 64k particles, the may store a lot more. This can be computed by dividing the GPU's memory by the amount of data per particle. Per particle, there should be enough memory to store the mass (1 float), position (3 floats), velocity (3 floats), acceleration (3 floats), jerk (3 floats) and potential (1 float) for input, and the acceleration, jerk and potential again as output (because of the double buffering). Each float takes up 4 bytes of space in memory, so this results in  $21 * 4 = 84$  bytes per particle. The Quadro FX 1400 contains 128 MB of memory, and will thus be able to store 1.5 million particles, while the GeForce 8800 GTX has 768 MB which may store over 9 million particles. Finally, GPUs are much cheaper than the GRAPE. Using this type of GPU, everybody can do complex  $N$ -body computations at home. Given the fact that even operating systems as Vista require fast graphics hardware, future GPUs will almost certainly be faster than the special purpose hardware currently available.

# Chapter 5

## Conclusion and discussion

This thesis provides an overview of the capabilities and performance of modern programmable Graphics Processing Units (GPUs). An overview was given of the hardware architecture of a GPU, the programming languages available to program them, and example applications in both graphics and generic computing problems. We focused on the use of GPUs for generic computing, first through an example implementation of Conway’s Game of Life and second to direct gravitational  $N$ -body simulations.

The project started as an exploration to the possibilities of the programmable GPU, since we had little knowledge of it. Soon, it became clear that the GPU is a highly underrated calculating device suitable for many scientific calculations. The project was therefore extended to use in a real time environment. The  $N$ -body problem was chosen, since experts in that field were available in our research group. We immediately realized that the  $N$ -body problem was not a ‘perfect fit’ to do on a GPU, as the Game of Life is. By accepting the challenge, we learned how the GPU can be exploited especially in those non-perfect cases.

### 5.1 Performance

As can be seen in the performance sections of the Game of Life and the  $N$ -body problem, there can be huge performance gains when using the GPU. The question people always immediately ask is: “How much faster is it then?”. The question is not as easy to answer though, since the performance gain depends on several things. Of course the speed and type of both CPU and GPU plays an important role. The GPU evolves very rapidly currently, so a state of the art GPU performs significantly better than a GPU that is designed two years ago. Besides this characteristic, the type of program is at least as important.

### 5.2 Precision

An important argument to decide not to use the GPU as a calculating device can be the precision. We have managed to maintain a precision similar to single 32-bits floating point precision on the CPU by using an offscreen framebuffer. Single 32-bits precision though, is

not enough for all applications. Though there have been attempts to use double precision using emulation [10] or correction [9], this will cause a significant reduction in performance.

### 5.3 Suitable applications

Perhaps the most important question the reader may have is: “Is my application suitable to perform on the GPU?” Therefore, we will point out exactly what applications are and are not suitable.

Both the  $N$ -body problem and the Game of Life performed faster on a GPU than on a CPU. The extend of this speedup is noteworthy. While the GPU is more than 50 times faster in the Game of Life, it is only 5-10 times<sup>1</sup> faster in the  $N$ -body problem. This difference is caused by the amount of communication needed in the  $N$ -body calculation. For an application to be suitable for implementation on a GPU, the ratio computation/communication should be high. Furthermore, the application must have a single instruction, multiple data (SIMD) nature. Without this characteristic, usage of the GPU is pointless, as the parallelism can not be exploited. Finally, GPUs are not optimized to handle conditional statements very efficiently. Especially loops should be avoided if possible in a GPU program. Note that executing a GPU program on a set of data, removes the loop over the array automatically from the CPU program.

---

<sup>1</sup>The speedup in the  $N$ -body problem heavily depends on the number of particles.

# Chapter 6

## Future work

During our project, we have tried to implement and experiment as much as possible. However, there are infinite possibilities for this project and each challenge takes time. Besides, new techniques are invented continuously, and with that, more and different options. Therefore, in this section we will outline suggestions for future work.

### 6.1 CUDA

The Compute Unified Device Architecture (CUDA) is the latest GPU programming environment and tries to bridge the gap between GPU and CPU programming languages [24]. Designed by NVIDIA, it will most likely be optimized for only NVIDIA hardware. CUDA tries to both simplify and speed up GPU programs using a new software and hardware architecture. Thus, CUDA currently only works on the newest NVIDIA hardware (the G80 architecture).

### 6.2 More shaders for the $N$ -body problem

In the gravitational  $N$ -body problem, we use the GPU to calculate the force calculation. This is computationally the heaviest part, but unfortunately we have to start this program for each block. This causes much communication overhead as the input and output parameters have to be send back and forth each time. A solution would be to write multiple shaders that not only calculate the forces, but also the prediction, correction, and all other intermediate calculation. Though it is not yet clear if this can be done efficiently, but if it can, this will mean a huge performance gain. The program would only need to communicate with the GPU twice: once to send the input, and once to retrieve the output. The implementation in this thesis did not use this method to allow a fair comparison with solutions that use the GRAPE in a similar configuration.

## 6.3 ATI/AMD fushion

In October 2006, Advanced Micro Devices (AMD) decided to buy ATI. As mentioned before in this thesis, ATI is the second biggest GPU manufacturer in the world, while AMD is the second biggest in the CPU industry. AMD's acquisition of ATI already reveals their plans for the future: A CPU and a GPU combined on one chip. This speculation is confirmed by information about a AMD project codenamed 'Fushion'. In an interview with CRN.com AMD's executive vice president Mario Rivas says: "With its Fusion program, AMD hopes to deliver multicore products using different kinds of processing blocks. A GPU, for example, will excel in multiple parallel computational tasks, while the CPU will take on heavy number-crunching duties. The Fusion-based processors, with the CPU and GPU integrated in a single architecture, should make the life of software programmers and application developers much easier." Recently, IBM announced to integrate the Cell in its mainframes<sup>1</sup>. Known from Sony's Playstation 3, the Cell uses SIMD very similar to a GPU.

## 6.4 A look in the future

The architecture of a GPU differs very much per generation of GPUs, it is not easy to compare two GPUs with each other. The performance of a GPU depends on the tasks it is required to do. In order to give a prediction for the future, a comparison must be made. One way to compare GPUs is to look at the potential maximum floating point operations per second (FLOPS). This number is not very representative for the actual performance, as this requires certain operations in a certain order repeated over and over again. Still, this number may be used as a relative comparison between GPUs. Figure 6.1 shows how GPUs manufactured by NVIDIA have evolved over the years, and how GPUs will perform if this trend continues. GPU performance increases a lot faster than CPU performance does currently. GPUs will, therefore, play an increasing role in scientific computing, and will likely become more a co-processor, than just a special purpose computer for graphics computations.

---

<sup>1</sup><http://www.research.ibm.com/cell/>

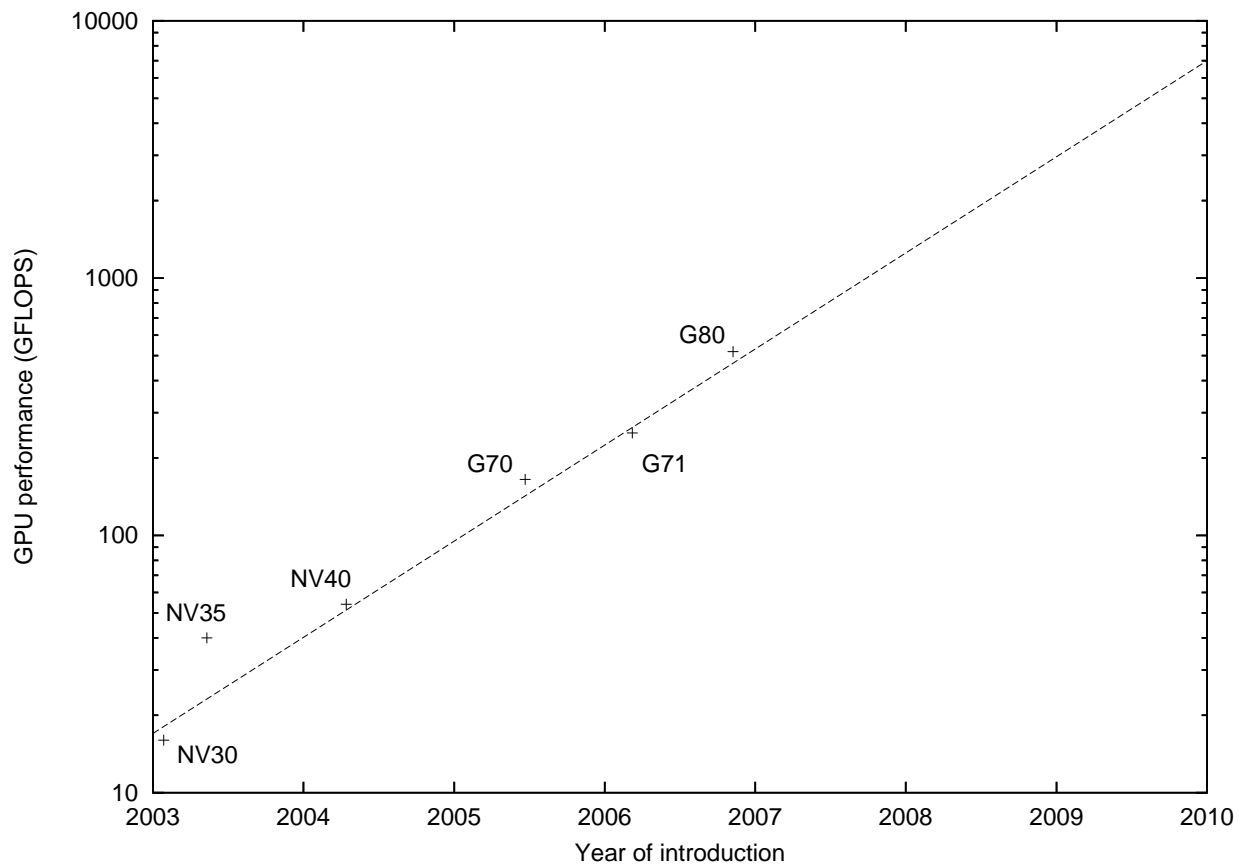


Figure 6.1: A prediction of future GPU performance. The dots represent the actual performance of the top NVIDIA GPU at that time. A line is put through these points and extended to indicate virtual future performances for GPUs.

# Bibliography

- [1] ATI. <http://ati.amd.com/>.
- [2] M. Botnen and H. Ueland. The GPU as a Computational Resource in Medical Image Processing. Technical report, Department of Computer and Information Science, Norwegian University of Science and Technology, 2004.
- [3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph. ISSN 0730-0301*, 23(3):777–786, 2004.
- [4] Debra L. Cook, John Ioannidis, Angelos D. Keromytis, and Jake Luck. CryptoGraphics: Secret Key Cryptography Using Graphics Cards. In *RSA Conference, Cryptographer's Track (CT-RSA)*, pages 334–350, February 2005.
- [5] Randima Fernando. *GPU Gems. ISBN 0321228324*. Addison-Wesley, 2004.
- [6] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. ISBN 0321194969*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [7] M Gardner. The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, 223:120–123, 1970.
- [8] D. Goddeke. GPGPU–Basic Math Tutorial. <http://www.mathematik.uni-dortmund.de/goeddeke/gpgpu/>. Technical report, FB Mathematik, Universitat Dortmund, November 2005. Ergebnisberichte des Instituts fur Angewandte Mathematik, Nummer 300.
- [9] Dominik Goddeke, Robert Strzodka, and Stefan Turek. Accelerating Double Precision FEM Simulations with GPUs. In Frank Hulsemann, Markus Kowarschik, and Ulrich Rude, editors, *Proceedings of the 18th Symposium on Simulation Technique (ASIM 2005)*, pages 139–144. SCS Publishing House e.V, September 2005.
- [10] Dominik Goddeke, Robert Strzodka, and Stefan Turek. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 2007. accepted for publication November 2006, to appear.

- [11] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.
- [12] Mark Harris. Mapping Computational Concepts to GPUs. In Matt Pharr, editor, *GPU Gems 2*. ISBN 0-321-33559-7, chapter 31, pages 493–508. Addison-Wesley, March 2005.
- [13] D.C. Heggie and R.D. Mathieu. Standardised Units and Time Scales. In P. Hut and S.L.W. McMillan, editors, *LNP Vol. 267: The Use of Supercomputers in Stellar Dynamics*, pages 233–+, 1986.
- [14] Erik Holmberg. On the Clustering Tendencies among the Nebulae ii. a study of encounters between laboratory models of stellar systems by a new integration procedure. *The Astrophysical Journal*, 94(3), November 1941.
- [15] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph. ISSN 0730-0301*, 22(3):908–916, 2003.
- [16] Youquan Liu, Xuehui Liu, and Enhua Wu. Real-Time 3D fluid simulation on GPU with Complex Obstacles. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*. ISBN 0-7695-2234-3, pages 247–256, Washington, DC, USA, 2004. IEEE Computer Society.
- [17] J. Makino. Optimal order and time-step criterion for Aarseth-type n-body integrators. *The Astrophysical Journal*, 369:200–212, March 1991.
- [18] J. Makino and S.J. Aarseth. On a Hermite integrator with Ahmad-Cohen scheme for gravitational many-body problems. *Publications of the Astronomical Society of Japan*, 44:141–151, April 1992.
- [19] J. Makino and M. Taiji. *Scientific Simulations with Special-Purpose Computers—the GRAPE Systems*. ISBN 0-471-96946-X. John Wiley & Sons, Ltd., April 1998.
- [20] J. Makino and M. Taiji. GRAPE-6: Massively-Parallel Special-Purpose Computer for Astrophysical Particle Simulations. *Publ. Astron. Soc. Japan*, 55:1163–1187, December 2003.
- [21] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph. ISSN 0730-0301*, 22(3):896–907, 2003.
- [22] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. ISBN 1568812299. AK Peters Ltd, 2004.
- [23] DirectX Microsoft Corp. <http://www.microsoft.com/windows/directx/>.

- [24] NVIDIA. Compute Unified Device Architecture Programming Guide. <http://developer.nvidia.com/cuda/>.
- [25] NVIDIA. <http://www.nvidia.com/>.
- [26] Marc Olano and Anselmo Lastra. A Shading Language on Graphics Hardware: The PixelFlow Shading System. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ISBN 0-89791-999-8, pages 159–168, New York, NY, USA, 1998. ACM Press.
- [27] OpenGL. <http://www.opengl.org/>.
- [28] John Owens. Streaming Architectures and Technology Trends. In Matt Pharr, editor, *GPU Gems 2*, chapter 29, pages 457–470. Addison-Wesley, March 2005.
- [29] Craig Peeper and Jason L. Mitchell. Introduction to DirectX 9 High Level Shading Language, 2003.
- [30] Matt Pharr and Randima Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. ISBN 0321335597. Addison-Wesley, mar 2005.
- [31] Pixar. The renderman interface specification (<http://renderman.pixar.com/products/rispec/>), 1989.
- [32] H. C. Plummer. On the Problem of Distribution in Globular Star Clusters. *MNRAS*, 71:470, 1911.
- [33] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ISBN 1-58113-374-X, pages 159–170, New York, NY, USA, 2001. ACM Press.
- [34] Randi J. Rost. *OpenGL(R) Shading Language*. ISBN 0321197895. Addison-Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.
- [35] Martin Rumpf and Robert Strzodka. Level Set Segmentation in Graphics Hardware. In *Proceedings of IEEE International Conference on Image Processing (ICIP'01)*, pages 1103–1106, 2001.
- [36] Martin Rumpf and Robert Strzodka. Nonlinear Diffusion in Graphics Hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 75–84, 2001.
- [37] Martin Rumpf and Robert Strzodka. Using Graphics Cards for Quantized FEM Computations. In *Proceedings VIIP Conference on Visualization and Image Processing*, pages 193–202, 2001.

- [38] OpenGL Utility Toolkit. <http://www.opengl.org/resources/libraries/>.
- [39] Sebastian von Hoerner. Die numerische Integration des n-Körper-Problems für Sternhaufen, II. *Zeitschrift für Astrophysik*, 57:47–82, 1963.
- [40] Sean Whalen. Audio and the Graphics Processing Unit. <http://www.node99.org/projects/gpuaudio/gpuaudio.pdf/>, 2005.
- [41] Simon Portegies Zwart, Robert Belleman, and Peter Geldof. High Performance Direct Gravitational N-body Simulations on Graphics Processing Units. 2007. Accepted for publication in *New Astronomy*.

# Appendix A

## The $N$ -body problem fragment program

The fragment shader program used in our implementation of the gravitational  $N$ -body problem. For practical reasons, we have put the output values for the potential energy and the jerk in one 4-dimensional texture.

```
void compute_acc_jerk_and_pot(
    in float2 coords      : TEXCOORD0, // 2D texture coordinate of this particle
    out float3 acc        : COLOR0,    // Output texture with acceleration
    out float4 jerkAndPot : COLOR1,    // Output texture with jerk and potential
    uniform samplerRECT accTexture,    // Input texture with all accelerations
    uniform samplerRECT jerkAndPotTexture, // ,, ,, ,, jerks & potentials
    uniform samplerRECT massTexture,    // ,, ,, ,, ,, masses
    uniform samplerRECT posTexture,     // ,, ,, ,, ,, positions
    uniform samplerRECT velTexture,     // ,, ,, ,, ,, velocitys
    uniform float eps2,                // Softening
    uniform float otherParticle,       // 1D index to other particle
    uniform float texSizeX,            // Width of all textures
    uniform float texSizeY,            // Height of all textures
    uniform float offset)              // Number of unused texture elements
{
    float coords1D, newCoords1D, otherMass,
           r2, xdotv, r2inv, rinv, r3inv, r5inv, xdotvr5inv;
    float2 newCoords;
    float3 pos, otherPos, vel, otherVel, dx, dv, thisAcc, thisJerkAndPot;

    // Get the data from the textures
    acc      = texRECT(accTexture, coords).rgb;
    jerkAndPot = texRECT(jerkAndPotTexture, coords).rgba;
    pos      = texRECT(posTexture, coords).rgb;
    vel      = texRECT(velTexture, coords).rgb;
}
```

```

// Convert the 2D texture coordinate to 1D and increase with otherParticle
// to obtain the coordinate of this iteration's other particle. Because our
// textures are defined as samplerRECT, texture elements must be addressed
// as (x+0.5,y+0.5). When converting to 1D, we must compensate for this
// offset).
coords1D = round(coords.y-0.5)*texSizeX + round(coords.x-0.5);
newCoords1D = coords1D + otherParticle;

// Skip over unused texture elements
if (newCoords1D + offset > texSizeX*texSizeY - 1)
    newCoords1D = newCoords1D - (texSizeX*texSizeY - offset);

// Convert the other particle's 1D coordinate to 2D. As above, we must add
// 0.5 to obtain correct texture element coordinates.
newCoords = 0.5 + float2( frac(newCoords1D/texSizeX)*texSizeX,
                        floor(newCoords1D/texSizeX) );

// Get the position, velocity and mass of this iteration's other particle
otherPos = texRECT(posTexture, newCoords).rgb;
otherVel = texRECT(velTexture, newCoords).rgb;
otherMass = texRECT(massTexture, newCoords).r;

// Now we can compute acceleration, jerk and potential
dx = otherPos-pos;
dv = otherVel-vel;
r2 = eps2 + dot(dx,dx);
xdotv = dot(dx,dv);
r2inv = 1.0/r2;
rinv = sqrt(r2inv);
r3inv = r2inv*rinv;
r5inv = r2inv*r3inv;
xdotvr5inv = 3.0*xdotv*r5inv;
thisAcc = otherMass*r3inv*dx;
thisJerkAndPot = otherMass*(r3inv*dv - xdotvr5inv * dx);
acc = acc + thisAcc;
jerkAndPot.rgb = jerkAndPot.rgb + thisJerkAndPot;
jerkAndPot.a = jerkAndPot.a - otherMass*rinv;
}

```