A Data Parallel Approach to Genetic Programming Using Programmable Graphics Hardware

Darren M. Chitty
QinetiQ Malvern
Malvern Technology Centre
St Andrews Road, Malvern
Worcestershire, UK
WR14 3PS
dmchitty@qinetiq.com

ABSTRACT

In recent years the computing power of graphics cards has increased significantly. Indeed, the growth in the computing power of these graphics cards is now several orders of magnitude greater than the growth in the power of computer processor units. Thus these graphics cards are now beginning to be used by the scientific community as low cost, high performance computing platforms. Traditional genetic programming is a highly computer intensive algorithm but due to its parallel nature it can be distributed over multiple processors to increase the speed of the algorithm considerably. This is not applicable for single processor architectures but graphics cards provide a mechanism for developing a data parallel implementation of genetic programming. In this paper we will describe the technique of general purpose computing using graphics cards and how to extend this technique to genetic programming. We will demonstrate the improvement in the performance of genetic programming on single processor architectures which can be achieved by harnessing the computing power of these next generation graphics cards.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Automatic Programming

General Terms

Algorithms

Keywords

Genetic Programming, Graphics Cards, Data Parallelism

1. INTRODUCTION

For many years in the computer industry there has been a law which can be applied to predict the power of processors in computers. This law is known as Moore's Law [8] which specifies that the number of transistors on a microprocessor will double every eighteen months. It is generally accepted that the number of transistors on micro-processors directly relates to the computing power of these processors. Therefore, Moore's Law effectively predicts that computing power itself will double every eighteen months. However, it has been postulated that this law no longer holds true for Computer Processor Units (CPUs) as over the last few years the increase in CPU power has been limited. This is in part due to the physical limits of processing capability using silicon wafers being reached (there is a physical limit to how many transistors can placed on silicon wafers). A second aspect, which has had an effect, is that the main consumers of computer products are office workers. The main use of computer resources of the office environment is in spreadsheet and word processing applications, for which the power of current Pentium processors is sufficient. Thus, if there is no longer a consumer need for a product, then it is unlikely to be heavily invested in in order to be further developed.

However, there is one aspect of consumer computing that is still driven by a requirement for high processing power: the computer games industry. The computer games industry requires large amounts of computer processing power in order to be able to render realistic in-game graphics. There are two aspects to the computer games industry, console based gaming, which has dedicated hardware and Personal Computer (PC) based gaming. PC gaming requires specialist graphics cards to provide the cutting edge 3D graphics that are required by modern computer games. The continual consumer need for more realistic graphics in computer games has driven the development of graphics cards generating successively higher levels of computing power.

Thus, this continual increase in raw graphics card processing power does still adhere to Moore's law. Owens *et al* [9] have done a thorough review of graphics cards and their computing capabilities and Figure 1 shows how the power of graphics cards from NVidia and ATI compare to Intel Pentium processors over the last few years. It can clearly be seen from the graph that whilst the graphics cards are meeting or exceeding Moore's Law, Pentium processor power has only improved by small increments each year. We can also compare the power of two most powerful graphics cards available and the latest Intel Pentium processor. The latest NVidia graphics card to be released is the GeForce 8800GTX which has 620 million transistors and the latest ATI graphics card is the 1950XTX which has 384 million transistors. Compare this to the Pentium 3.7GHz Dual Core Processor which only has 320 million transistors. We can also compare the prod-

GECCO'07 July 7-11, 2007, London, England, United Kingdom
Copyright 2007 ACM 978-1-59593-697-4/07/0007 ...$5.00.

ucts from another aspect by using the number of floating point operations per second (GFLOPS) which can be carried out. The Pentium processor can achieve 25.6 GFLOPS compared to the ATI hardware which can achieve 240 GFLOPS. However, the NVidia graphics card can demonstrate a much greater performance with 520 GFLOPS.

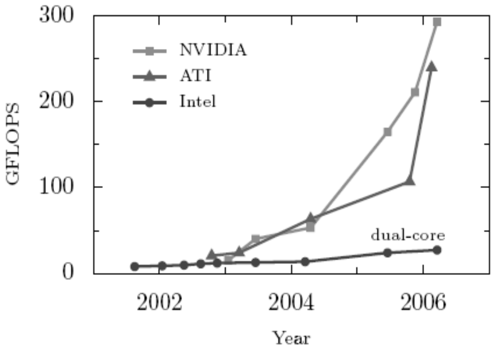


Figure 1: The performance of graphics cards vs CPUs over the last four years (taken from [9])

Therefore, much better results from genetic programming could be obtained by harnessing the graphics card computing power previously reserved for 3D graphics. Indeed the use of graphics cards to perform various computer intensive algorithms has grown considerably over the last few years. For instance, Li *et al* [6] have successfully implemented the Lattice Boltzman computation on a graphics card. Also Bernhard *et al* [1] have simulated spiking neurons on graphics hardware and Sumanaweera *et al* [10] have performed FFTs for medical image reconstruction on a graphics card. Also within the field of evolutionary computation, Yu *et al* [12] have implemented genetic algorithms on a graphics card, with both the crossover and mutation operators and the fitness evaluation function being performed on the graphics card. The authors were able to demonstrate up to a twenty fold increase in the performance of the genetic algorithm when using large population sizes. Therefore, we can expect a significant increase in the speed of genetic programming by using these graphics cards. This paper will discuss the architecture of these graphics cards describing how this performance gain in GFLOPs is achieved. We will then go on to describe the methodology of performing general purpose computing on these graphics cards and the extension of this technique to genetic programming. We will apply the technique to some classical genetic programming problems to demonstrate the increase in performance that can be gained from utilising these graphics cards.

2. GRAPHICAL PROCESSING UNITS

Graphics cards have a main processing area on them known as a Graphical Processing Unit (GPU). There are many varieties of graphics cards although at this time there are only two manufacturers of graphics cards that can perform the computations we require, namely NVidia and ATI. These graphics cards are high performance graphics cards aimed at computer gamers and as such are capable of fast, complex 3D graphics operations. In order for graphics cards to draw 3D images, many thousands of triangles are sent to the GPU which then converts them to pixels in a 2D image using specialist programs. These output pixels are then sent to screen to produce the 3D image. Kilgariff *et al* [4] have conducted a detailed study of the architecture of the series 6 graphics cards from NVidia. The architecture of GPUs is based on carrying out operations on textures using a set of internal processors known as vertex and fragment processors. Textures are small parts of an image (usually small triangles) which need to be drawn to the screen. The parallelism of GPUs comes from there being multiples of these processors and graphics pipelines which are capable of carrying out operations simultaneously on different parts of the input textures. The GPU on the latest graphics card from NVidia has a total of twenty four of these pipelines.

There are two types of processor on a GPU, vertex processors and fragment processors. A vertex processor is capable of Multiple Instruction Multiple Data (MIMD) operations and can perform scatter operations but not gather operations. A scatter operation is a process whereby data can be written in a non-sequential manner. A gather operation is a process whereby data can be collated. Fragment processors are capable of Single Instruction Multiple Data (SIMD) operations, are also capable of random access memory reads and can perform gather but not scatter operations. Fragment processors are more useful for non-graphics operations than vertex processors as they can produce a direct output and there are also more fragment processor pipelines on a GPU than vertex processor pipelines. There are two types of programs which can be run on a GPU specific to each processor known as vertex programs and fragment programs. Fragment programs take a set of inputs and will output a single four channel RGBA (Red Green Blue Alpha) value. Each element of the inputs will be independently processed by the fragment processors hence for a ten by ten input texture a fragment processor will be run a hundred times. With up to 24 fragment processors on the latest GPUs, this can lead to a large amount of parallelism resulting in the GFLOP performance gain of GPUs over CPUs. Figure 2 shows a high level representation of how textures from an application are processed by a GPU, flowing through the vertex and fragment processors and output to the screen.

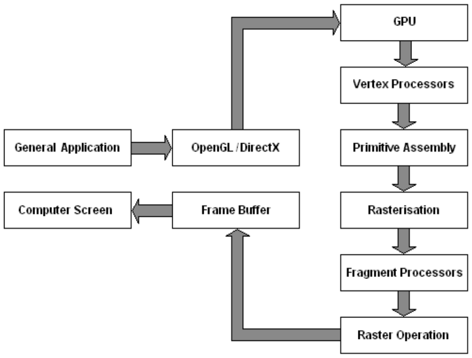


Figure 2: A simplified view of the interface between the GPU and applications

There is a second aspect of GPUs which can boost their performance level over CPUs other than parallel processing. The native memory structure of a GPU is two dimensional compared to that of a CPU which is one dimensional. Using a CPU it is possible to use multi-dimensional arrays although in the actual memory core they are reduced to a single dimension. On both CPUs and GPUs there is a cache which allows faster memory accesses to elements which are in the neighbourhood of the previous element accessed. In a single dimension memory structure there are only two neighbourhood paths. However in two dimensional memory structures there are four neighbourhood directions. Therefore there is a greater statistical likelihood that the next element to be accessed will be on this cache. Thus the two dimensional memory structure of a GPU will provide an additional performance gain over a CPU. It is worth noting that the memory structure of GPUs currently restrict the dimension size of a texture to a maximum of 4096 elements.

3. GENERAL PURPOSE PROGRAMMING ON GPUS

Due to the capability of GPUs to perform parallel operations for 3D graphics it soon became apparent that these processors could also be used to perform high performance computing tasks. As such a new field of research has developed which has developed techniques for exploiting graphics cards for the purposes of scientific computing. Thus a general methodology has been developed for performing general purpose programming on GPUs which involves three key steps:

- Conversion of data to graphics textures
- Construction of a fragment program
- Execution of the fragment program on the texture

3.1 Terminology

There are a number of differences in the terms that are used to describe general purpose programming and 3D graphics programming. In order to understand how to perform general purpose computing on GPUs we need to be able to relate these GPU concepts to general computing concepts.

The first concept we need to understand is how images on a graphics card relate to the data we wish to process. We have already discussed in section 2 how the memory layout on a GPU is in a two dimensional form compared to the one dimensional form of a CPU. Image data held in GPU memory is referred to as a texture which is in effect a 2D array. It is possible to have 3D textures on GPUs although there is a performance deficit from using them thus in this paper we will concentrate on 2D textures. Therefore, arrays of general data need to be transferred to a texture on the GPU which has two dimensions. It should also be noted that in order to access arrays on the CPU we simply use an index value. However to access elements of a texture a GPU needs what are known as texture coordinates which allows the elements that are being processed on each fragment or vertex processor to be calculated. This will ensure that each processor will only work on a single part of the input and only output a specific part of the result.

The second concept that needs to be considered is how programs are executed on GPUs, and how they compare to typical scientific or general purpose applications. Programs on GPUs are known as either fragment or vertex programs. These correspond to fragment or vertex processors of which there are many on a GPU and which can operate in parallel on an input texture. Let us consider the following equation:

$$x_i = x_i^2 + 3.14y_i + c \qquad \text{for } i = 1, \ldots, n \tag{1}$$

To implement this in C we would simply write a piece of code which consists of a `for` loop with the equation executed for n data elements. Thus there are two elements of computation, the equation and the increment of the `for` loop. However, if we had n processors then we would only require the equation aspect as each processor could perform the equation for each data element. This type of operation is known as a SIMD operation or a data parallel approach. The calculation inside the `for` loop can thus be considered to be a kernel program, a scalar template that converts several inputs into a single output. Also, because the equation is performed on each data point in succession, there are no dependencies between data points. GPU programs work in the same manner, generating a new texture of pixels from a number of input textures. Because the data is independent and there are many processors, the program can be run on different input elements simultaneously. Hence it can be seen that kernel programs are similar to the programs executed on a GPU.

A third concept we need to consider is the relationship between running normal programs and running 3D graphics programs. In order to run an equation on some data we would simply call the desired function, passing the data to it. However, in order to cause the GPU program to execute on the textures which contain our data, we need to render or draw the target with the target being the output texture from the fragment program.

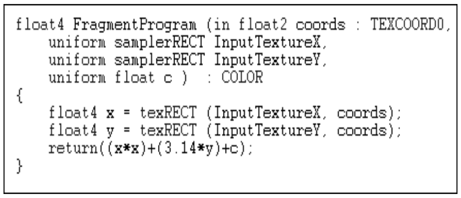
In conclusion, we have introduced the three key concepts that are needed to understand how to implement general programming on GPUs. These can be summarised as follows:

- Textures = Arrays
- GPU programs = Kernel programs
- Rendering = Program execution

3.2 Tools For Using Graphics Cards

Generally, directly interfacing with GPUs is a very complex operation. However, there are two software toolkits available which can provide an interface to graphics programming. These are known as DirectX, which is a Microsoft architecture, and the Open Graphics Library (OpenGL) [3] (Open Graphics Library) which is an open source implementation of a specification for a low level API for writing 3D graphics applications. The OpenGL format is managed by a consortium of companies known as the ARB, the OpenGL Architecture Review Board. Companies on the board include NVidia, ATI, and Intel. The OpenGL format is hardware independent and also operating system independent as opposed to DirectX which is targeted at the Windows platform. Additionally, the OpenGL format is a much more popular format than the DirectX format due to the capability for it to be extended much more rapidly to deal with GPU developments. The frame buffer object in OpenGL is an example of this as it provides a mechanism for allowing the GPU programs to write directly to a texture rather than a frame buffer. This is useful as whatever is written to the frame buffer is modified into a value in the interval [0,1] which is not suitable for our purpose. Therefore we have decided to concentrate on the OpenGL format in this paper.

The operations that a GPU performs using fragment or vertex processors can be stored on the GPU in the form of complete programs. The programs that a GPU can understand are written in a form of machine code which is difficult to write or understand. Therefore, NVidia has provided a free toolkit [7] known as C for Graphics (Cg) which is downloadable from their website and allows users to construct programs in a C like language which can then be compiled by the toolkit into GPU programs. Many of the functions that are available in C are also available in Cg. The toolkit also provides most of the boolean and mathematical operators that are available in C but also provides additional vector types and vector operators. For example, a type exists in Cg known as a float4 which is a vector of four floating point numbers. These vectors have operators available which enable them to be added and multiplied together. In order to transfer the Cg program to a GPU the toolkit provides a set of functions which compile and load these programs on to the GPU in real time. In figure 3 we show an example of a program written in the Cg format which can be compiled by the toolkit and subsequently executed on a GPU.



```
float4 FragmentProgram (in float2 coords : TEXCOORD0,
    uniform samplerRECT InputTextureX,
    uniform samplerRECT InputTextureY,
    uniform float c )  : COLOR
{
    float4 x = texRECT (InputTextureX, coords);
    float4 y = texRECT (InputTextureY, coords);
    return((x*x)+(3.14*y)+c);
}
```

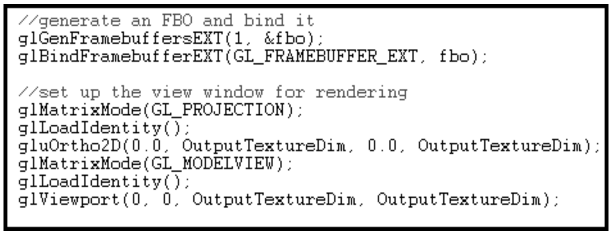
Figure 3: An example fragment program

Some aspects of using the Cg programming language should be noted. The first of these is that all branches of `if` statements are executed by the GPU. However, the result of all the branches that are not of interest are masked. Thus there can be less of a performance gain if the Cg program contains a large number of `if` statements. Secondly, programs executed on GPUs are not capable of generating random numbers. Therefore, if a set of random numbers is required, they must be supplied as an input texture to the program.

3.3 Initialising A GPU

We have discussed that in order to use a GPU for general purpose programming, we need to copy data over to the GPU, provide a GPU program to do the processing, and run this program by rendering the target. To be able to perform these operations we will use the OpenGL toolkit described in section 3.2. To render an output texture we firstly need to set up what is known as a frame buffer. This is part of the GPU memory whereby the image that would normally be displayed on screen can be read. However, the frame buffer has a problem in that the values stored use the RGBA encoding with values set to between 0 and 255 for each channel. Thus, the frame buffer is unable to represent 32 bit floating point values. However, OpenGL provides an off-screen buffer which removes this restriction allowing full 32 bit floating point precision. This buffer is known as the Frame Buffer Object (FBO) and in order to start performing general purpose computing on a GPU we need to initialise one of these objects.

Also, in order to render a target texture which will contain the output data, we need to convert the graphics view from three dimensions to two dimensions. In addition, we need a one-to-one mapping between the data contained in the textures and the output pixels. The method required to obtain this is to use a view which can provide a one-to-one mapping from geometry coordinates (rendering) to texture coordinates (input) and pixel coordinates (output). To achieve this, OpenGL allows us to specify a view port using an orthogonal projection. An example of the steps needed to initialise the GPU is provided in figure 4.

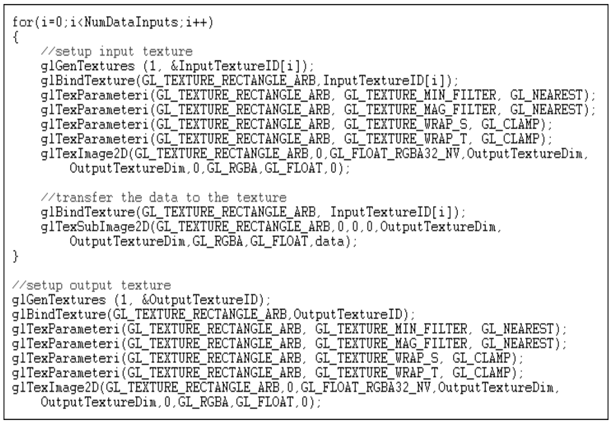


```
//generate an FBO and bind it
glGenFramebuffersEXT(1, &fbo);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);

//set up the view window for rendering
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, OutputTextureDim, 0.0, OutputTextureDim);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, OutputTextureDim, OutputTextureDim);
```

Figure 4: Initialisation steps needed to use GPU

3.4 Copying Data To A GPU

Once we have set up the structures to use the GPU we then need to transfer to the GPU the data we wish to use as input in the form of textures. The textures we will consider here are 2D in nature and have equal dimension sizes. It is possible to use 3D textures and also different dimension sizes although this causes inefficiencies in the data processing which can slow the operation of the GPU. These textures are of an IEEE 32 bit floating point form. To copy the data into a texture OpenGL provides the functions we require to create a texture of the right size. Hence, a 1D data array of size n will convert to a $\sqrt{n}$ by $\sqrt{n}$ size texture that can hold the data. Obviously, there will need to be some spare capacity in the texture if n is not a square number. The next step is to then bind the input texture to the output target texture. We can then simply transfer the input data to the texture using OpenGL functions. Some example lines of code which demonstrate this are shown in figure 5.



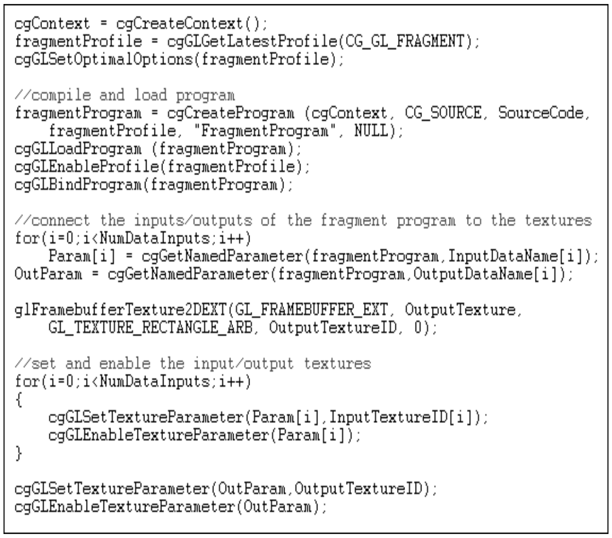
```
for(i=0;i<NumDataInputs;i++)
{
    //setup input texture
    glGenTextures (1, &InputTextureID[i]);
    glBindTexture(GL_TEXTURE_RECTANGLE_ARB,InputTextureID[i]);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,0,GL_FLOAT_RGBA32_NV,OutputTextureDim,
        OutputTextureDim,0,GL_RGBA,GL_FLOAT,0);

    //transfer the data to the texture
    glBindTexture(GL_TEXTURE_RECTANGLE_ARB, InputTextureID[i]);
    glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,0,0,0,OutputTextureDim,
        OutputTextureDim,GL_RGBA,GL_FLOAT,data);
}

//setup output texture
glGenTextures (1, &OutputTextureID);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,OutputTextureID);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,0,GL_FLOAT_RGBA32_NV,OutputTextureDim,
    OutputTextureDim,0,GL_RGBA,GL_FLOAT,0);
```

Figure 5: How to copy data arrays into textures and on to the GPU

3.5 Creating a Fragment Program

When the data has been transferred onto the GPU, we then need to provide a program to manipulate the data in order that we obtain our desired output. There are two types of programs which can be run on a GPU relating to vertex and fragment processors. Due to the number of fragment processors being greater in number on a GPU, we will use fragment programs to achieve the greatest degree of parallelism. A fragment program must be in a machine code format which the GPU can recognise. However, the Cg toolkit as described in section 3.2 provides a C like language which can be compiled into the correct format that the GPU recognises. Fragment programs are typically compact and consist of only a few instructions which implement kernel programs. An example of a fragment program for general purpose programming on a GPU is shown in figure 3. It can be seen that the fragment program has inputs just like a C function. These consist of the input textures, and the texture coordinates which are required parameters. A fragment program can also contain an optional number of constant inputs such as c in our earlier example. The texture coordinates are computed by the GPU in the graphics pipeline before the fragment program is executed on one of the processors. This ensures that the fragment processor operates on the correct elements of the textures. To convert the texture information to the correct type we simply use the tex2D function and the texture coordinates. We can then calculate the equation as we would normally for a single data point and return the result into the target texture.

There are several steps we need to go through in order to generate a fragment program which can be used by the GPU. First we need a CGcontext object in order to be able to use the Cg toolkit. We also need to specify which type of program we will require, a vertex type program or a fragment type program. We can then compile a Cg program (such as that in figure 3) using the Cg toolkit functions provided. If it compiles successfully we can then load it onto the GPU. An example of the steps required to achieve this are shown in figure 6. We also need to ensure that we correctly bind the correct texture inputs to the relevant input arguments of the fragment program.

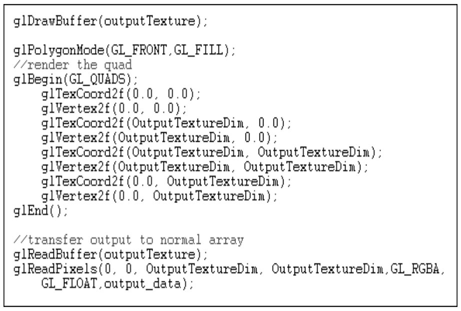


```
cgContext = cgCreateContext();
fragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
cgGLSetOptimalOptions(fragmentProfile);

//compile and load program
fragmentProgram = cgCreateProgram (cgContext, CG_SOURCE, SourceCode,
    fragmentProfile, "FragmentProgram", NULL);
cgGLLoadProgram (fragmentProgram);
cgGLEnableProfile(fragmentProfile);
cgGLBindProgram(fragmentProgram);

//connect the inputs/outputs of the fragment program to the textures
for(i=0;i<NumDataInputs;i++)
    Param[i] = cgGetNamedParameter(fragmentProgram,InputDataName[i]);
OutParam = cgGetNamedParameter(fragmentProgram,OutputDataName[i]);

glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, OutputTexture,
    GL_TEXTURE_RECTANGLE_ARB, OutputTextureID, 0);

//set and enable the input/output textures
for(i=0;i<NumDataInputs;i++)
{
    cgGLSetTextureParameter(Param[i],InputTextureID[i]);
    cgGLEnableTextureParameter(Param[i]);
}

cgGLSetTextureParameter(OutParam,OutputTextureID);
cgGLEnableTextureParameter(OutParam);
```

Figure 6: Example of compiling and loading fragment programs

3.6 Executing The Fragment Program

Once we have loaded the data and the fragment program on to the GPU we then need to execute the program on the data. This is achieved by rendering or drawing the target. We can load many fragment programs on to the GPU, however only one can be ready to be executed at any one time. Thus we need to enable the program we wish to use and also we need to ensure that the output from the program is written to a texture attached to the FBO. Once this has been achieved the target output texture is ready to be rendered. Hence, given that we are using a one-to-one mapping, we simply need what is known as a quad which fills the viewport. This quad is generated using OpenGL whereby the four corners of the quad are specified. Therefore, the size of the quad is of dimensions $\sqrt{n}$ by $\sqrt{n}$. The rasteriser will subsequently interpolate each pixel (or array element) in the quad and generate a fragment for each pixel. These are then passed to the fragment processors which operate in parallel in a data stream operation. Exemplar code to achieve this is shown in figure 7. All that remains is then to transfer the results from the FBO into a data array. To do this we use the OpenGL functions glReadBuffer and glReadPixels.



```
glDrawBuffer(outputTexture);

glPolygonMode(GL_FRONT,GL_FILL);
//render the quad
glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(0.0, 0.0);
    glTexCoord2f(OutputTextureDim, 0.0);
    glVertex2f(OutputTextureDim, 0.0);
    glTexCoord2f(OutputTextureDim, OutputTextureDim);
    glVertex2f(OutputTextureDim, OutputTextureDim);
    glTexCoord2f(0.0, OutputTextureDim);
    glVertex2f(0.0, OutputTextureDim);
glEnd();

//transfer output to normal array
glReadBuffer(outputTexture);
glReadPixels(0, 0, OutputTextureDim, OutputTextureDim,GL_RGBA,
    GL_FLOAT,output_data);
```

Figure 7: How to render a target to execute a fragment program

4. A PARALLEL IMPLEMENTATION OF GENETIC PROGRAMMING FOR GPUS

Genetic Programming (GP) [5] is the process of using evolutionary processes to create a program which can generate the desired output from a set of inputs. Therefore, it can be seen that in GP the data remains static whilst the programs change. GP can be a slow process as evaluating large numbers of programs is computationally intensive. However GP is inherently parallel in that each candidate program can be evaluated independently from the others. This means that the technique can be implemented on multi-processor machines or distributed over a network. Thus the parallelism is derived from many candidate solutions being evaluated at the same time. This is different from how the parallelism in GPUs is obtained, whereby different data points can be processed simultaneously for a single candidate program. Therefore, the parallel implementation of GP on a GPU will only evaluate a single candidate program at a time. This technique is known as a data parallel approach and has been previously applied to GP by Tufts [11] however this was implemented on a distributed system.

In order to implement GP on a graphical processing unit we need to implement a system whereby the data remains constant and the fragment programs change. This is the reverse of traditional general purpose applications on GPUs which we described in section 3, whereby the fragment programs remain constant and the data changes for new problem instances. These applications (such as image processing) usually require large amounts of data with multiple passes of the fragment program over the input data. However, GP uses the same data throughout with multiple candidate programs being run on the same data with the overall aim of determining the best program. Therefore, we need to maintain the data as textures on the GPU such that we do not need to repeatedly transfer data from the CPU memory to the GPU memory, which is a slow process. We instead modify the program each time before we render an image in order to execute the candidate program. To modify the fragment program being run, we need to construct the program in a Cg format from the genetic string using a Cg format; and then compile and load it on to the GPU. This is significantly different to a standard GP approach which involves iterating through recursive functions which can evaluate expressions as specified by the chromosome. Thus standard GP is in effect an interpreted system whereby the GP expression is formulated and executed at run time. A significant performance gain from the GPU approach is immediately evident due to the capability to be able to compile and load fragment programs in real time using the Cg toolkit.

The high level steps that are needed to implement a data parallel version of GP on a graphics card are specified below:

1. Initialise OpenGL and setup graphics window
2. Copy data inputs into separate textures
3. Create a cgContext object
4. Generate the initial population
5. For each individual:
   - Convert chromosome to Cg source code using recursive functions
   - Compile and load program using Cg Toolkit
   - Bind program and textures to the cgContext object
   - Render the Cg program on the data
   - Get output from the GPU
   - Compare result with desired output and assign a fitness to the individual
6. For each generation:
   - Generate a new population using crossover and mutation
   - Evaluate population by performing step 5
7. Cleanup and output best result

Hence we see that the major difference between standard GP and a version implemented on a GPU is in the evaluation of the chromosomes. In standard GP we recursively evaluate each subtree in the chromosome which form a closed expression. We perform this recursion for each set of data points or fitness cases. The recursive aspect interprets each element of the chromosome and converts it to the corresponding terminal or operator.

With the GPU implementation of GP, the recursive function which interprets the chromosome remains. However, instead of the expressions being evaluated, each is written to a string which forms a complete fragment program. Each separate subtree expression is stored in a temporary variable which is then used as an input to other expressions further up the tree. If we consider the following equation:

$$f(x) = x^3 - y + z^2 \qquad (2)$$

then the GP-Tree is shown in Figure 8 and the corresponding fragment program is shown in Figure 9.

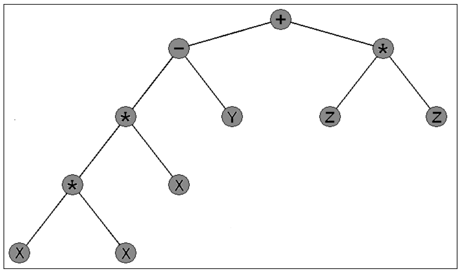
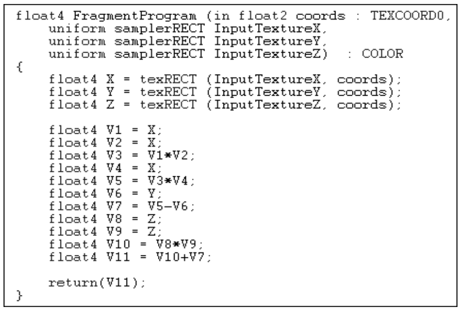


Figure 8: A GP-Tree example



```
float4 FragmentProgram (in float2 coords : TEXCOORD0,
    uniform samplerRECT InputTextureX,
    uniform samplerRECT InputTextureY,
    uniform samplerRECT InputTextureZ)  : COLOR
{
    float4 X = texRECT (InputTextureX, coords);
    float4 Y = texRECT (InputTextureY, coords);
    float4 Z = texRECT (InputTextureZ, coords);

    float4 V1 = X;
    float4 V2 = X;
    float4 V3 = V1*V2;
    float4 V4 = X;
    float4 V5 = V3*V4;
    float4 V6 = Y;
    float4 V7 = V5-V6;
    float4 V8 = Z;
    float4 V9 = Z;
    float4 V10 = V8*V9;
    float4 V11 = V10+V7;

    return(V11);
}
```

Figure 9: A Cg fragment program (generated from Figure 8)

However, there are some aspects which need to be observed, such that changing the GPU program whilst retaining the input data on the graphics card is relatively fast. A cgContext object is used to attach the data and fragment programs together on the GPU in order to perform the rendering. Typically, in general purpose GPU programming it is customary to create and destroy this context each time we render a program. However, whilst conducting the research in this paper, it was found that it was not necessary to destroy the context for each run of a program. This results in a considerable reduction in the runtime of the GPU Genetic Programming algorithm.

In addition, whilst we have shown how to implement GP on GPU hardware it should be mentioned that the technique is not applicable to all problem domains. The main area for which this GP technique can be used is the area of classification problems whereby GP attempts to generate a set of outputs from a set of inputs which will match a desired outcome. Examples of this include data mining tasks and symbolic regression problems. However, it is envisaged that applications that would not be applicable to this technique are simulation based problems. For example, robot control tasks to achieve a given behaviour require simulated runs which is in essence a single data instance with a large number of inputs which produces multiple outputs. Therefore, the fragment processors will be unable to operate in parallel. What is necessary for this technique to be useful is large volumes of data. We will discuss in the results section the level of speedup that can be obtained for successively larger volumes of data.

5. PROBLEM DOMAINS

To evaluate the performance of using a GPU to perform genetic programming, we will apply the technique to three example problem types. These problems are symbolic regression, a classification task using the Fisher Iris data set [2], and a multiplexer problem. These problems have been chosen as they are relatively distinct from each other. In the symbolic regression problem a single line equation is evolved whilst the iris classification task is a multi-classification task and the multiplexer problem is a two class classification problem. A description of each of the problems follows:

Objective	Find a symbolic function which will map a single input to a single output whereby the target function is $f(X) = x^4 + x^3 + x^2 + x$
Terminal operands	x, 1.0
Terminal operators	+, *, /, and -
Fitness	Sum of the error over all fitness cases

Table 1: Symbolic regression example 1 problem parameters

Objective	Find a symbolic function which will map a single input to a single output whereby the target function is $f(x) = 2.76x^2 + 3.14x$
Terminal operands	x, [0.0..10.0]
Terminal operators	+, *, /, and -
Fitness	Sum of the error over all fitness cases

Table 2: Symbolic regression example 2 problem parameters

5.1 Symbolic Regression

Symbolic regression problems involve finding a symbolic mathematical expression which matches a given set of input and output values. Thus the objective is to find a mathematical function which will map the inputs to the outputs. The function which we shall investigate is described as follows:

$$f(x) = x^4 + x^3 + x^2 + x \quad (3)$$

with x in the range [-1,1].

For this example we will use 100 fitness cases and the parameters we will use are described in table 1. In addition, we also use a second symbolic regression problem described by equation (4) for which it is harder to find a solution:

$$f(x) = 2.76x^2 + 3.14x \quad (4)$$

with x in the range [-1,1].

We will use 400 fitness cases for this problem to demonstrate the relationship between execution times and the number of fitness cases. The parameters we will use for this problem are described in table 2.

5.2 Fisher Iris Data Set Classification

The Fisher Iris dataset is a well known multi-class classification problem that uses Iris flowers. There are three types of Iris in the dataset, Iris Setosa, Iris Veriscolor and Iris Virginica. There are four input parameters for this problem consisting of the width and length of the petals and the width and length of the sepals. We labeled the Iris flowers using numerical constants [1,2,3] which we added as terminal constant values. We will use the full data set of 150 examples to use as our fitness case. The fitness measure will be the number of correct classifications found. The problem parameters are shown in table 3.

5.3 11-Way Multiplexer

Multiplexers are examples of addressing problems whereby instances have k address bits and 2^k data bits. The value in the address section specifies a data element. Thus for correct instances the data element specified by the address bits will be set to one and for incorrect instances this data bit will be set to zero. In this problem we will use three address bits and thus eight data bits. This means that there are 2048 individual instances which will form the fitness cases we will use. The parameters we will use for this problem are shown in table 4.

Objective	Find a symbolic classification rule which correctly labels the three types of iris data set
Terminal operands	Petal Length, Petal Width, Sepal Length, Sepal Width, constant in the interval [0.0..10.0] and three constants [1,2,3]
Terminal operators	+, *, /, -, =, >, <, AND and OR
Fitness	The number of correct classifications made over 150 fitness cases

Table 3: Fisher Iris data classification problem parameters

Objective	Find a symbolic classification rule which correctly labels the output of an 11-way multiplexer
Terminal operands	Each address and data element of the 11-way multiplexer
Terminal operators	=, !=, AND and OR
Fitness	The number of correct classifications made over 2048 fitness cases

Table 4: 11-way Multiplexer classification problem parameters

6. RESULTS

We generated the results presented in this paper using a 1.7GHz Pentium 4 processor with 1GB of memory. The graphics card used was an NVidia GeForce 6400 GO which has sixteen fragment processors. We compared the data parallel implementation of GP on GPU with a standard interpreted GP approach. All results were generated using a population size of 500 individuals evolved over 50 generations with crossover of 1.0 and mutation of 0.001. All the results were averaged over 50 trial runs.

In table 5 we compare the execution times of the standard GP approach and the data parallel implementation using a GPU. It can be seen that for instances of problems with only a few fitness cases that the GPU approach to GP has longer execution times. However, for problem instances with a larger number of fitness cases we can see a significant performance gain from the GPU technique over the standard GP algorithm. To further illustrate this point we compare in figure 10 the execution times of both algorithms using the harder symbolic regression problem and steadily increasing the number of fitness cases. It can be clearly seen from the graph that the increase in execution time compared to the increase in the number of fitness cases is strictly linear for a standard genetic algorithm approach. Compare this to the GPU implementation and it can be seen that increasing the number of fitness cases has no effect on the execution time of the algorithm. We can also see that at the lowest level of fitness cases that the standard technique outperforms the GPU technique However, we can see that once the number of fitness cases exceeds approximately 200 then the data parallel GPU approach becomes the faster technique. In figure 11 we show the increase in the execution time of the GP GPU technique for considerable larger volumes of fitness cases using the same symbolic regression problem. We can see that a significant number of fitness cases are required to affect the execution time of the GPU approach to GP, approximately 10000 to 15000 cases.

Problem	Standard GP	GPU GP
Symbolic Regression 1	122	300
Symbolic Regression 2	2531	277
Iris Classification	226	252
11-way Multiplexer	9115	304

Table 5: Average execution time in seconds for each problem instance

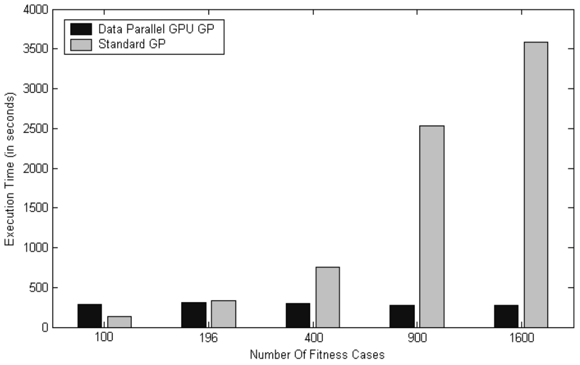


Figure 10: Execution timings using a symbolic regression problem

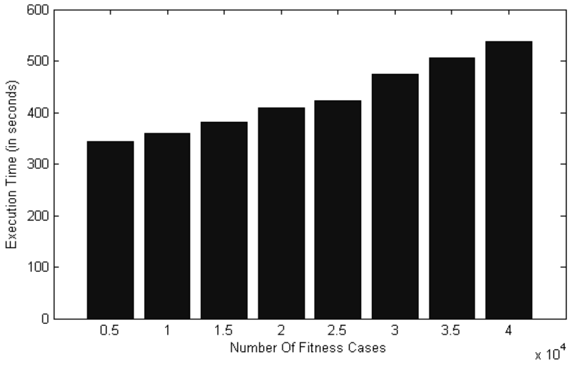


Figure 11: Execution timings for GPU GP and significantly larger numbers of fitness cases

7. CONCLUSIONS

In this paper we have presented the technique of performing general purpose computing on graphics cards and moreover, we have extended this technique to genetic programming. Thus we implemented a much faster version of genetic programming by using a data parallel approach which can use the graphics card. Therefore it is now possible for parallel genetic programming to be run on single processor systems which have a graphics card.

Our results have shown that although there is little improvement for small numbers of fitness cases, when this number becomes much larger then considerable gains can be made using the GPU implementation of GP. It should also be noted that there is significant room for improvement. The results generated here were generated using a mobile NVidia GeForce GO 6400 graphics card which has only 16 fragment processors whilst the latest NVidia graphics card has 24, a significant improvement. Also, whilst the GPU was evaluating a candidate program, the CPU was idle waiting for the GPU to return. Thus it may be possible to implement a system whereby the CPU is also utilised perhaps by having a standard GP approach evaluating candidate programs at the same time as the GPU is operating. Furthermore, it should also be possible to combine the data parallel approach using GPUs and a standard parallel implementation of GP to increase increase the speed of GP. It should be possible to distribute genetic programming over a network of computers each one of which has an NVidia graphics card. Thus we should see a similar performance gain over a standard parallel implementation of genetic programming.

8. REFERENCES

[1] F. Bernhard and R. Keriven. Spiking Darren Neurons on GPUs. In *Computational Science – ICCS 2006*, volume 3994, pages 236–243. Springer, 2006.

[2] R. A. Fisher. The Use of Multiple Measurements in Taxonomic Problems. *Annual Eugenics*, 7:179–188, 1936.

[3] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL Shading Language Version 1.10.59. *http://www.opengl.org/documentation/oglsl.html*, 2004.

[4] E. Kilgariff and R. Fernando. The GeForce 6 Series GPU Architecture. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 29. ACM Press, 2005.

[5] J. R. Koza. *Genetic Programming: On the Means of Programming Computers by Means of Natural Selection*. MIT Press, 1992.

[6] W. Li, X. Wei, and A. Kaufman. Implementing Lattice Boltzmann Computation on Graphics Hardware, 2003.

[7] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, volume 14, pages 896–907, 2003.

[8] G. E. Moore. Cramming More Components onto Integrated Circuits. 38:56–59, 1965.

[9] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A Survey of General–Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 2007.

[10] T. Sumanaweera and D. Liu. Medical Image Reconstruction with the FFT. In *GPU Gems 2*, pages 471–491. Addison-Wesley, 2005.

[11] P. Tufts. Parallel Case Evaluation for Genetic Programming, 1995.

[12] Q. Yu, C. Chen, and Z. Pan. Parallel Genetic Algorithms on Programmable Graphics Hardware. volume 3612, pages 1051–1059. Springer, 2005.