Chapter 15

ACCELERATING GENETIC PROGRAMMING THROUGH GRAPHICS PROCESSING UNITS

Wolfgang Banzhaf[1], Simon Harding[1], William B. Langdon [2] and Garnett Wilson [1]

[1]*Dept. of Computer Science, Memorial University of Newfoundland, St. John's, NL, Canada;*
[2]*Dept. of Mathematical Sciences, Essex University, UK.*

Abstract Graphics Processing Units (GPUs) are in the process of becoming a major source of computational power for numerical applications. Originally designed for application of time-consuming graphics operations, GPUs are stream processors that implement the SIMD paradigm. The true degree of parallelism of GPUs is often hidden from the user, making programming even more flexible and convenient. In this chapter we survey Genetic Programming methods currently ported to GPUs.

Keywords: graphics processing units, parallel processing

1. Introduction

There is a notorious drag on Genetic Programming. Program search spaces are huge, and often difficult to navigate (Banzhaf et al., 1998; Langdon and Poli, 2002). Statistical measures for performance frequently require a large number of runs of GP systems in order to arrive at significantly safe statements. Finally, even running a single program might cause grief, due to the potentially large number of fitness cases required for a GP system to evolve a reasonable model of the underlying problem. In the extreme case, fitness cases have to be drawn from a probability distribution, and there will never be the same fitness case shown to the system again. To make things worse, measurements or simulations used to provide the data for a single fitness case evaluation might run from microseconds to days.

A simple calculation can easily demonstrate the amount of processing power required. Suppose we use a standard tree GP system, with each node being evaluated within $10^{-6}s$. If we assume that the problem is realistically difficult

for individual programs to regularly reach a depth limit of 17 using binary function nodes, an individual program will evaluate in approximately $0.131\ s$. Taking part of that size (not every program will be of maximum depth), 10^5 nodes, and multiplying with standard parameters for a GP run (number of fitness cases 10^3, population size 10^3, generations until end of run 10^3, and number of runs for statistical purposes 10^2), we end up with a runtime of $10^{10} s$, or $317\ yrs$. Even going to a billion node evaluations per second, which is certainly on the optimistic side, will require us to run the experiment for $116\ days$. So realistically, only experiments which can knock off a factor of 10^3 can reasonably be expected to be done.

Clearly, these types of considerations lead to restrictions on the types of problems presently being subjected to Genetic Programming, as well as to evaluation limitations in terms of number of fitness samples, or length of programs. This, in turn, will change the evolvability and quality of solution, or, in the former case, even prohibit certain problems from being addressed by GP.

With GPUs, the situation is bound to change fundamentally. “As of 2007, the fastest PC processors perform over 30 GFLOPS. GPUs in PCs are considerably more powerful in pure FLOPS. For example, in the GeForce 8 Series the NVIDIA 8800 Ultra performs around 576 GFLOPS on 128 processing elements. This equates to around 4.5 GFLOPS per element, compared with 2.75 per core for the Blue Gene/L supercomputer. It should be noted that the 8800 series performs only single precision calculations, and that while GPUs are highly efficient at calculations they are not as flexible as a general purpose CPU.” (Wikipedia, 2008) With the advent of that kind of processing power, more problems become accessible, and now it is feasible for researchers to try a whole set of problems of a real-world application - rather than 1 or 2 demonstrations.

2. Various Sources of Speed-up for Genetic Programming

Problems of resource demand have always been with GP, but it appears as if the GP community has been pretty successful in circumventing the worst obstacles. It is clear that GP is very amenable to speedup through various methods. Here we’ll just shortly outline a few of the sources for this potential.

First, there clearly are some factors that allow independent decomposition of tasks which lend themselves easily to parallelization:

1) Individual programs are evaluated using multiple independent fitness cases;

2) Populations consist of individuals which could be evaluated on independent hardware in parallel;

3) Repeated (independent) runs for producing appropriate statistical confidence levels can be executed simultaneously on different hardware.

Other factors do not lend themselves to easy parallelization because they require dependence, yet may also be exploited:

4) Evolutionary generations which build on each other

5) Program execution which usually requires a sequence of execution steps to be performed

6) Evaluation of single nodes which might be highly complex and serial processes

We think the above mentioned parallelization potential corresponds to the following causes:

1) Difficulty of the problem;

2) Population size chosen;

3) Statistical significance of results;

4) Speed of evolution with a given representation;

5) Size of the search space;

6) Complexity of the problem domain.

Issues (1) to (3) are more easy to address in a parallel computing environment while issues (4) to (6) are more difficult, but not impossible, to address.

There is probably no generic solution to problem (6), but accelerating the evaluation of single nodes is certainly possible, either directly (see below in the case of image filters) or indirectly by using proxy evaluation (Ziegler and Banzhaf, 2003). It is more difficult to address the dependency problems of (4) and (5), although attempts have been made to relax the sequentiality of programs, see, e.g. (Banzhaf and Lasarczyk, 2004). As for 4), an entire area of research has sprung up around more efficient and more easily evolvable representations, which has a bearing on the speed of evolution. But a radical approach like dissolving the sequence of generations is not possible, given the very nature of evolution, although some inroads can be made in dissolving the dependency, e.g. the success of steady-state algorithms or evaluation queueing.

In general, it can be expected that the hardware landscape is now sufficiently heterogeneous that each case will have another optimum for the expoitation of parallel resources. It remains to be seen how adaptive algorithms (forthcoming 2008) will be able to take advantage of these resources.

3. Classical Parallelization

Early work on parallelization of Genetic Programming already includes SIMD architecture approaches (Juille and Pollack, 1995) and a GP system written in FORTRAN running on a Cray super computer (Turton et al., 1996). Even earlier was work porting classifier systems / GAs to Thinking Machines SIMD architecture of the Connection Machine (CM) back in the 1980s (Robert-

son and Riolo, 1988) which even preceded the coevolution of parallel sorters by Hillis (Hillis, 1990).

Applications were the driving force for parallel approaches to GP (Oussaidene et al., 1996; Stoffel and Spector, 1996), no wonder given the restrictions on single runs discussed earlier. Koza at al. (Andre and Koza, 1996; Bennett III et al., 1999) popularised the use of transputer boards and later Beowulf workstation clusters where the population is split into separately evolving demes with limited emigration between compute nodes or workstations (Page et al., 1999). Multipopulation models were more systematically studied by (Fernández et al., 2003). New techniques became also available through progress with parallel hardware (Folino et al., 2003b; Folino et al., 2003a).

A number of groups demonstrated the use of the Internet for parallel GP, e.g., (Chong and Langdon, 1999; Gross et al., 2002; Folino et al., 2006). In these approaches the GP population can be literally spread over the globe. Alternatively JavaScript has been used to move interactive fitness evaluation to the user's own home but retain elements of a centralised population (Langdon, 2004). In recent times, cloud computing has been added to the options.

Others have used special purpose hardware. For example, (Eklund, 2003) used a simulator and was able to show how a linear machine code GP can be run very quickly on a field programmable gate array under VHDL to model sun spot data. FPGA were used already early on as acceleration tools for fitness evaluation (Koza et al., 1997). Martin employed a special C-compiler for FPGAs to run a GP system (Martin, 2001). However, FPGA architectures are intrinsically cumbersome to handle for an average programmer, and should be considered tools for the specialist, in contrast to Genetic Programming on GPUs.

One important consideration with every sort of parallel approach is the price one pays for hardware to accelerate computation. Manufacturers have exploited the need of scientists for computation for a long time. Cray and other parallel computers, including recent machines like IBM's BlueGene/L, carry a high pricetag.

Computing clusters, on the other hand, consist of commodity hardware and are therefore much cheaper. However, they also have high and continued operating costs, just as commercial parallel machines. Over the lifetime of even a small cluster, the operating costs can become several times that of the initial hardware investment (Feng et al., 2002). The costs come from a variety of sources, such as system adminstration, power, cooling and space. Being able to use a single desktop machine mitigates most of these costs.

In recent years, it has become recognized that GPUs provide much more economical means of achieving extremely parallel computation. To provide some indicative statistics, in mid-2007 the latest Intel CPU, the Core 2 Extreme QX6800, was capable of over 37 GFLOPS (INTEL, 2008). The CPU also had

a suggested retail price of $999 US at release, providing the consumer with a cost of approximately $27/GFLOP.

One of the latest benchmarked PC NVIDIA graphic cards, on the other hand, the GeForce 8800 GTX, provides 520 GFLOPS (NVIDIA, 2006) at an initial retail price of US $599 for $1.15/GFLOP. The NVIDIA 8800 has been used in much of the research presented in this chapter.

As one of the largest drivers of GPU power, computer gaming has emerged. Video game consoles are often sold at a loss in terms of hardware at product launch. This makes them a very economical choice for parallel power for parallel computation research. For instance, as early as 2005, the MS Xbox 360 claimed to provide 1 TFLOP (overall system performance, using both CPU and GPU computation) (XBOX, 2008). The most economical Xbox 360 package at that time sold for $299 US, providing a cost of $0.29/GFLOP. In this chapter, we describe how the Xbox 360 can be used to perform both general purpose computation, and GP.

4. The GPU platform and its potential

Graphics Processing Units are specialized stream processors, useful for rendering graphics applications. Typically, a GPU is able to perform graphics manipulations at a much higher speed than a general purpose CPU, since the graphics processor is specifically designed to handle certain primitive operations which occur frequently in graphics (game) applications. Internally, the GPU contains a number of small processors (see Figure 15-1) that are used to perform calculations on 3D vertex information and textures. A texture is a collection of pixels, in the form of a 2D image.

GPUs are constructed so that the simple vertex and shader processors work in parallel and in a pipeline fashion. The vertex processors calculate the 3D view, then the shader processors paint the model before it is displayed. Depending on the power of a GPU, the amount of parallel processors currently runs from 2 to 64 on both the vertex and the shader level. The fact that GPUs have the ability to perform restricted parallel processing has elicited considerable interest among researchers with applications requiring intensive parallel computation. Although the type of parallel processing used by GPUs, SIMD, is not the most general model of parallel computing, the sheer amount of raw processor power and the comparable low cost on graphics cards make it an attractive choice for a large number of applications, e.g. in scientific computing.

A GPU processes textures and outputs a vector of four floating point numbers for each texture element (texel) processed, traditionally corresponding to RGBA (red, green, blue, and alpha, for transparency) channels of a color. The two components of a GPU architecture that a user can control are the set of vertex processors and the set of pixel (or fragment) processors. An effect file, which

is a program to control the GPU, is divided into two parts corresponding to the architecture: a pixel shader and a vertex shader. The vertex shader program transforms input vertices based on camera position, and then each set of three resulting vertices computes a triangle from which pixel (fragment) output is generated and sent to the pixel processors. The shader program instructs the pixel shaders (processors) to ”shade” each pixel in parallel and produce the final pixel with associated RGBA values for final output. Even though the latest GPUs (such as all those used this paper) use unified architectures, where the shader processors can handle vertex or pixel commands, the two functionalities are still separated when composing effect files.

While early GPUs tended to be severely restricted in the type and number of instructions they could execute, newer GPU architectures foresee the tendency to make freer use of the processing resources offered. Today, the number of instructions executable on a shader processor is not limited any more, and the type of instructions is broad, like all floating-point operations, instead of narrow. General purpose applications of GPUs (GPGPU) tend to take advantage of pixel shader programming rather than using the vertex processors, mainly because there are typically more pixel- than vertex-shaders on a GPU and the output of the pixel shaders is fed directly to memory (Harris, 2005). In terms of traditional data structures and execution, GPU textures are analogous to arrays, the shader program is like a Kernel program, and rendering effectively executes the program on array elements in parallel.

Programming a GPU

In this section we provide a brief overview of some of the general purpose computation toolkits for GPUs that are available. This is not an exhaustive list, but is intended to act as a guide to others. Generally speaking, the field is quickly advancing and readers are advised to carefully check the internet for the newest resources.

Sh and RapidMind: Sh is an open source project for accessing the GPU under C++ (RapidMind, 2008; LibSh Wiki, 2008). Many graphics cards are supported, and the system is platform independent. Many low level features can be accessed using Sh, however these require knowledge of the mechanisms used by the shaders. The Sh libraries provide typical matrix and vector manipulations, such as dot products and addition-multiplication operators. Sh has now been developed into a commercial product called RapidMind. A key benefit of the RapidMind toolkit is the ability to target other multicore platforms, such as the Cell/BE processor which is found in Playstation3. The usefulness of RapidMind for genetic programming has already been demonstrated for parallel evaluation of entire populations with more than 1 million individuals, all run

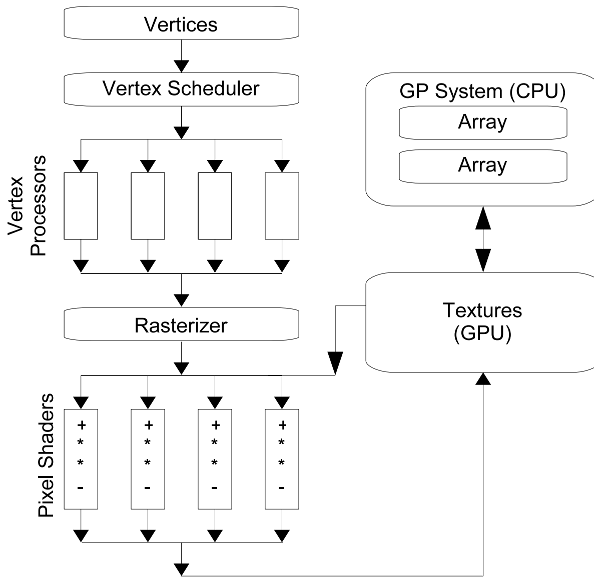

Figure 15-1. General sketch of GPU architecture and a GP implementation. Two sets of parallel stream processors are used to accelerate computation, vertex processors and pixel shaders. Only the second set of processors is used in these implementations.

in parallel (Langdon and Banzhaf, 2008), and we briefly mention its application here to Bioinformatics problems in Section 5.0.

Accelerator: Microsoft also has a prototype .Net assembly called Accelerator that provides access to the GPU via the DirectX interface (Tarditi et al., 2006). The system is completely abstracted from the GPU, and presents the end user with only arrays that can be operated on in parallel. Unfortunately, the system is only available for the Windows platform due to its reliance on DirectX. However, the assembly can be used from any .Net programming language.

Accelerator shares with RapidMind the design feature that operations are not performed on the data until the evaluated result is requested. This enables a certain degree of real time optimization, and reduces the computational load on the GPU. In particular, optimisation of common sub expressions will reduce the creation of temporary shaders and textures. The movement of data to and from the GPU can also be efficiently optimized, which reduces the impact of the relatively slow transfer of data out of the GPU. The compilation to the shader model occurs at run time, and hence can automatically make use of the different features available on the supported graphics cards.

We have previously used Accelerator for several different genetic programming tasks (Harding and Banzhaf, 2007a; Harding and Banzhaf, 2007b; Hard-

ing, 2008). In Section 5.0 of this paper we demonstrate the evolution of image filters using this toolkit.

CUDA and Cg: The GPU manufacturer NVIDIA also provides two toolkits for GPU programming. The Cg (C for graphics) toolkit allows one to develop shader programs for OpenGL or DirectX (NVIDIA, 2008; Mark et al., 2003). Cg provides extensions to the C language (such as data types) to make it suitable for GPU programming. It was designed to simplify shader programming, which was typically performed at the assembly language level. By interpretting textures and other graphics primitives as general purpose arrays, it is possible to implement general purpose programs. Regression and classification problems, using genetic programming, have already been successfully demonstrated (Chitty, 2007).

CUDA ("Compute Unified Device Architecture") is an NVIDIA hardware specific language for programming their more recent stream processors. Again, it is based on the C programming language and allows the user to write shader programs at a high level. CUDA also supports multiple GPU devices - and allows for different programs/data on each device. The toolkit provides fine grained control of how the GPU is utilised - potentially allowing for highly efficient use.

HLSL: Microsoft's High Level Shader Language (HLSL) is similar to Cg and CUDA. It is also a C-style language that can give some direct programmability of the shaders, but with the design tradeoff that they are used in conjunction with (rather than coded as part of) the source files in the higher level languages like C# and C++. With the current version of HLSL it is no longer necessary to use assembly-type instructions to access the shader functionality, and the language features C-like data, vector, and matrix types. Programs can be divided into C-like functions, and HLSL flow control includes both static and dynamic branching and looping. HLSL can be used to develop shader programs for DirectX that can work on both the Windows and XBox platforms. This has allowed Linear Genetic Programming to be implemented on the XBox and to benefit from the GPU (Wilson and Banzhaf, 2008).

It is interesting to note that Moore's Law is also valid for the growth of GPU performance over recent years, however with a different doubling parameter: While the doubling time for CPU performance is on the order of 18 months, doubling time for GPU performance is around 9-12 months. Thus, over time, the advantage of applications programmed for GPUs have over their CPU counterparts is bound to grow even faster.

The newest development on the hardware market are multiple GPU solutions. This is the next iteration of hardware improvement. For instance, the latest NVIDIA cards are essentially 2 of last years GPUs on one board. When coding for multiple devices, CUDA allows for completely different processes to run on each device.

5. Applications

At present most publications on using genetic programming with graphics hardware have been more concerned with proof of principle or demonstration of practical technology and algorithms rather than applying GPUs.

Large datasets are available in many different communities. For the demonstrations presented here, we chose one from the intrusion detection system community which was published for a competition in connection with the KDD conference 1999. We also discuss an image processing application, and a bioinformatics application. We finally demonstrate the feasibility of GPU computing on a gaming console.

The KDD 1999 IDS dataset

The full KDD Cup 1999 data set contains over four million entries. The data mining task was to find a network intrustion detector, that could discriminate between different types of network behaviour. The data itself has been critized, but we feel that the size of the dataset presents some challenges to conventional implementations. Because of memory restrictions on our current graphics card, we restrict ourselves to a subset of 10%, which contains 494,021 entries, each with 42 values. All input types were converted to floating point numbers (mapping string tokens to unique numbers). The dataset requires approximately 80 Mb of memory once in RAM.

Benchmark tests were performed using the RapidMind development platform, under Windows XP (Athlon 5200 CPU, 2 Gb RAM with a NVIDIA 8800 GTX, 768 Mb). We implemented Cartesian GP (CGP) (Miller and Thomson, 2000) for both the RapidMind GPU backend, and a straight C++ version. This allows us to easily compare timing for both a parallel and a conventional implementation. The CPU implementation uses only a single core. We chose not to use the RapidMind C++ backend as the overhead of invoking the native compiler is excessive - the system is better suited to situations where the parallel parts of the code only need to be compiled once.

For both the CPU and GPU, we used the function set of Table 15-1. We peformed evolutionary runs using both the GPU and CPU implementations, and recorded the number of Genetic Programming Operations Per Second (GPOps). Each run consited of evaluating 9005 individuals (200 generations, 50 indivdiduals with 5 elite individuals per population). We performed 10 runs of each implementation, and computed the timings based on statistics collected for each generation of each run.

For the GPU, peak performance was 694 million GPOps and an average of 388 million GPOps. In contrast, the CPU implementation peaked at 92 million GPOps with an average of 41 million GPOps.

On the GPU an individual took an average of 5.86 ms to evaluate, compared to 43.54 ms on the CPU – a speed up of 7.4 times on average. These timings include any overhead for data transfer and compilation of shader programs.

Image Filters

The evolution of image filters using Genetic Programming is a relatively unexplored task (but see (Poli, 1996)). This is most likely due to the high computational cost of evaluating the evolved programs. We have used a GPU implementation to tackle the challenge of reverse engineering image filters, i.e. to find the mapping between an image and the output of a filter applied to it. The filters we investigate in this paper are from the open source image processing program GIMP (GNU, 2008). To perform reverse engineering, again CGP is used to evolve programs acting as filters. These programs take a pixel and its neighbourhood from an image, and compute the next value of this central pixel. The convolution kernel is run on each pixel in an image producing a new image.

For much of the previous work on this problem a single, low resolution (256 x 256 pixel) image is used in the evaluation stage. This approach can be expected to result in over-fitting to a particular image. Thanks to the acceleration through the use of a GPU we were able to employ more images to help overcome such issues; here 16 different images (Figure 15-2) were used largely taken from the USC-SIPI image repository (12 used for fitness evaluation, and 4 for validation). This allows us to be confident that evolved filters will generalise well. As we are employing the GPU for acceleration, it is possible to test all images at the same time and obtain both the fitness and validation score simultaneously.

The original input images were combined together to form a larger image. A filter was applied using GIMP. We then employed the evolutionary algorithm to find the mapping between the input image and the output images. The fitness function attempts to minimize the error between the desired output (the GIMP processed image) and the output from the evolved filters.

Table 15-1 lists the available functions. A simple evolutionary algorithm with population of size 25, tournament selection (size 3) and elitism (best 3 individual) was used. We allowed evolution to run for 50,000 evaluations.

With this technique it was possible to evolve many different types of filters, such as *erode, dilate, emboss* and various edge detectors (see (Harding and Banzhaf, 2008) for more examples). Here we will illustrate the process with the Sobel filter, which is a type of directionless edge detector. Figure 15-3 shows how the evolved filter compares to the GIMP implementation.

Calculating the Genetic Programming Operations Per Second (GPOps) for our test system (NVIDIA 8800 GTX, AMD Athlon 3500+, Microsoft Accelerator API) a peak performance of 292 million GPOps and an average of 194 million GPOps was reached. Different operations appear to take different times

Table 15-1. CGP Function set used in the different applications. "x" means a particular instruction was used. The constants refer to the respective node's parameter.

Function	Description	KDD	Filters
ITERATION	Return the current iteration index		x
ADD	Add the two inputs	x	x
SUB	Subtract the second input from the first	x	x
MULT	Multiply the two inputs	x	x
DIV	Divide the first input by the second	x	x
ADD CONST	Add a constant to the first input	x	x
MULT CONST	Multiply the first input by a constant	x	x
SUB CONST	Subtract a constant from the first input	x	x
DIV CONST	Divide the first input by a constant	x	x
SQRT	Return the square root of the first input	x	x
POW	Raise first input to the power of second	x	x
COS	Return the cosine of the first input		x
SIN	Return the sin of the first input		x
NOP	No operation - return the first input		x
CONST	Return a constant		x
ABS	Return the absolute value of first input	x	x
MIN	Return the smaller of the two inputs		x
MAX	Return the larger of the two inputs		x
CEILING	Round up the first input	x	x
FLOOR	Round down the first input	x	x
FRAC	Return the fractional part of number		x
LOG2	Log (base 2) of the first input		x
RECIPRICAL	Return $1/firstinput$		x
RSQRT	Return $1/\sqrt{firstinput}$	x	x
CROOT	Return the cube root of first input	x	

Figure 15-2. The training and validation image set. All images are presented simultaneously to the GPU. The first column of images is used to compute the validation fitness, the remaining twelve for the training fitness. Each image is 256 by 256 pixels, with the entire image containing 1024 by 1024 pixels.

to execute. Across all the image filters we evolved, the peak performance was 324 Million GPOps, with an average of 145 GPOps. Using a CPU implemented reference driver, we were able to run the same experiments on the CPU. Execution yielded only 1.2 million GPOps, i.e. a factor of 100 times slower than the GPU. Considering the difference in performance, it would be impractical to run these experiments on the CPU with this size of image set.

A Bioinformatics Application

The applications from Bioinformatics are using the GPU primarily to speed up the fitness evaluation. In (Langdon and Harrison, 2008) GP was used to find a small set of GeneChip probes which indicated long term (10 or more years) survival of Breast Cancer.

From 1987 to 1989 during breast surgery tumour samples were saved and subsequently analysed using two GeneChips (HG-U133A and HG-U133B). These two chips together collect more than a million expression values for 251 patients (Miller, 2005). A major part of the data mining exercise was simply to use evolution to decide which of these are useful at predicting the patients future. GP was run multiple times in a series of passes to winnow the useful data

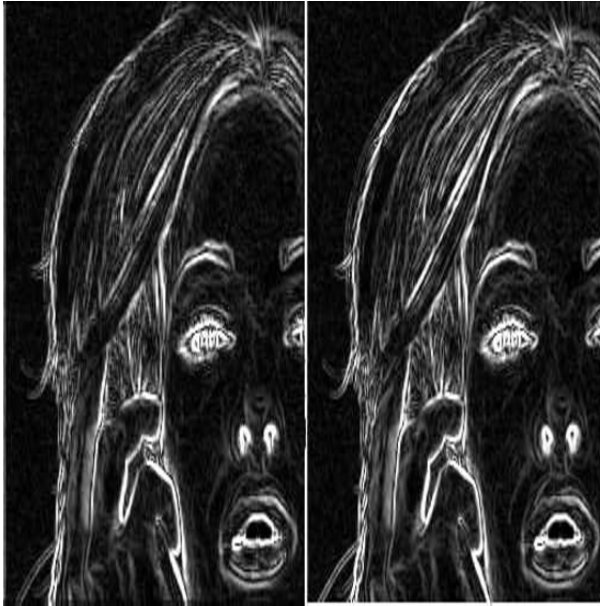

Figure 15-3. Evolving a Sobel edge detector. The left image shows the output from the evolved filter. The right image is the output from the GIMP filter. The two outputs are visually very similar.

from the remaining chaff (Langdon and Buxton, 2004). This involved running the GP system with a population of five million programs hundreds of times on hundreds of megabytes of training data. The final predictor was a non-linear function of only a handful of the million initial variables.

A second study (Langdon, 2008) also involved the use of GeneChips. However, GP was used here to locate problems with Affymetrix technology itself rather than looking at the medical results of applying that technology. Affymetrix Chips provide many (typically 22) separate readings for each human gene. If the GeneChip was well designed, all the readings should be well correlated. However our measurements of the correlation found thousands of cases where this was not true (Upton et al., 2008). GP was used to find patterns in the data to indicate which readings were unreliable. This has led to discussion about the underlying physics of the GeneChip, what may cause the problems and how they might be avoided in future designs (Upton et al., 2008).

XBOX Application

We have used the GPU processing power of a video game console, namely of the Xbox 360 (Andrews and Baker, 2006) for general computation. In late 2006, Microsoft launched XNA Game Studio Express, which integrated with

C# Studio Express, and thus became the first video game console to provide its users with access to its GPU. By using XNA with HLSL, we designed GPGPU applications and ported them to an Xbox 360 for execution. The application itself was a Linear GP program, which implemented both fitness evaluation and mutation using the GPU. Production with the XNA framework targeted to the Xbox 360 presented interesting design challenges.

An XNA project interface mandates that initialization (Initialize), update of program logic (Update), and rendering of graphics (Draw) methods be implemented. The program runs by repeatedly updating the Update and Draw methods — it is designed to be a video game that is constantly checking its logic status and updating the graphics on the screen. The Draw method is thus the main component of our GPGPU implementation, as this is where the shader programs on the GPU are called from. Rather than use a typical loop construct for LGP tournament execution, the repeated execution of the Draw method is harnessed to conduct generational tournaments over numerous trials.

The Draw method, as appropriate, would process the textures using HLSL programs loaded at compile-time or otherwise conduct CPU-side GP tournament processes. The population of LGP individuals is represented as multiple textures. In particular, two textures represent the content of the instructions, and another texture represents the registers. Multiple passes are used to examine the instruction in the first two textures and place subresults in the third register texture. A single large texture represents the fitness cases. Mutation was implemented with a texture of randomly chosen probabilities and a corresponding texture of potential chromosome replacements. The work (Wilson and Banzhaf, 2008) established, for the first time, how to use a video game console GPU for general computation in any capacity, and how to use a console in general (both CPU and GPU) for genetic programming. GPU implementations on the XBox 360 were found to be moderately faster than their CPU equivalents, but this is likely indicative of the tightly integrated nature of the CPU and GPU in the Xbox 360 architecture. Practically speaking, the results demonstrate that future GPGPU programmers of the Xbox 360 need not be as focused on performance consequences of placing functionality with the GPU as opposed to the CPU.

6. Perspective

In this chapter we have motivated the use of GPUs for Genetic Programming through the excessive computational demands GP poses on every system. We have explained the general approach to using a GPU, given an overview of currently usable software systems, and demonstrated by way of example a number of interesting applications of GP on GPUs. Our work succeeded in porting various aspects of linear, tree and graph GP systems onto GPU platforms.

It is too early to predict which types of parallelization of GP using GPUs will in the end be the most effective. For example, it would be helpful to identify categories of fitness functions that may and may not be suitable for implementation on GPUs. It appears that at first glance, many GP problems are easily mappable onto the GPU hardware. Since it can be expected that in the future the APIs will hide even more functionality implementations should become easier to acheive.

We hope that this chapter motivates many more researchers to turn their attention to this new platform that promises to revolutionize how Genetic Programming is performed.

Acknowledgment

We would like to thank Nolan White for helping us get the hardware working, and RapidMind for providing us with their code free of charge. WB further acknowledges support by NVIDIA.

References

Andre, David and Koza, John R. (1996). A parallel implementation of genetic programming that achieves super-linear performance. In Arabnia, Hamid R., editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume III, pages 1163–1174, Sunnyvale. CSREA.

Andrews, J. and Baker, N. (2006). Xbox 360 system architecture. *IEEE Micro*, 26:25–37.

Banzhaf, W., Nordin, P., Keller, R., and Francone, F. (1998). *Genetic Programming - An Introduction*. Morgan Kaufmann, San Francisco, CA, USA.

Banzhaf, Wolfgang and Lasarczyk, Christian W. G. (2004). Genetic programming of an algorithmic chemistry. In O'Reilly, Una-May, Yu, Tina, Riolo, Rick L., and Worzel, Bill, editors, *Genetic Programming Theory and Practice II*, chapter 11, pages 175–190. Springer, Ann Arbor.

Bennett III, Forrest H, Koza, John R., Shipman, James, and Stiffelman, Oscar (1999). Building a parallel computer system for $18,000 that performs a half peta-flop per day. In Banzhaf, Wolfgang, Daida, Jason, Eiben, Agoston E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1484–1490, Orlando, Florida, USA. Morgan Kaufmann.

Chitty, Darren M. (2007). A data parallel approach to genetic programming using programmable graphics hardware. In *GECCO 07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1566–1573, New York, NY, USA. ACM.

Chong, Fuey Sian and Langdon, W. B. (1999). Java based distributed genetic programming on the internet. In Banzhaf, Wolfgang, Daida, Jason, Eiben, Agoston E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1229, Orlando, Florida, USA. Morgan Kaufmann. Full text in technical report CSRP-99-7.

Eklund, Sven E (2003). Time series forecasting using massively parallel genetic programming. In *Proceedings of Parallel and Distributed Processing International Symposium*, pages 143–147.

Feng, W., Warren, M., and Weigle, E. (2002). The bladed beowulf: A cost-effective alternative to traditional beowulfs. In *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, page 245, Washington, DC, USA. IEEE Computer Society.

Fernández, F., Tomassini, M., and Vanneschi, L. (2003). An Empirical Study of Multipopulation Genetic Programming. *Genetic Programming and Evolvable Machines*, 4(1):21–51.

Folino, G., Forestiero, A., and Spezzano, G. (2006). A Jxta based asynchronous Peer-to-Peer Implementation of Genetic Programming. *JOURNAL OF SOFTWARE*, 1(2):13.

Folino, G., Pizzuti, C., and Spezzano, G. (2003a). A scalable cellular implementation of parallel genetic programming. *Evolutionary Computation, IEEE Transactions on*, 7(1):37–53.

Folino, G., Pizzuti, C., and Spezzano, G. (2003b). Ensemble Techniques for Parallel Genetic Programming Based Classifiers. *Genetic Programming: 6th European Conference, EuroGP 2003, Essex, UK, April 14-16, 2003: Proceedings*.

GNU (2008). Gnu image manipulation program (GIMP). www.gimp.org. [Online; accessed 21-January-2008].

Gross, R., Albrecht, K., Kantschik, W., and Banzhaf, W. (2002). Evolving chess playing programs. In Langdon, W. B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M. A., Schultz, A. C., Miller, J. F., Burke, E., and Jonoska, N., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 740–747, New York. Morgan Kaufmann Publishers.

Harding, S. and Banzhaf, W. (2007a). Fast genetic programming on GPUs. In Ebner, M., O'Neill, M., Ekart, A., Vanneschi, L., and Esparcia-Alcazar, A., editors, *Proc. 10th Europ. Conference on Genetic Programming, Valencia, Spain*, volume 4445 of *LNCS*, pages 90 – 101. Springer.

Harding, S. and Banzhaf, W. (forthcoming, 2008). Image filters evolved with genetic programming. Technical Report, Department of Computer Science, Memorial University of Newfoundland.

Harding, Simon (2008). Evolution of image filters on graphics processor units using cartesian genetic programming. In *WCCI*. IEEE.

Harding, Simon and Banzhaf, Wolfgang (2007b). Fast genetic programming and artificial developmental systems on GPUs. In *Proceedings of the Symposium on High Performance Computing Systems (HPCS-2007)*. IEEE, DOI: http://doi.ieeecomputersociety.org/10.1109/HPCS.2007.17.

Harris, M. (2005). Mapping computational concepts to gpus. In Pharr, M. and Fernando, R., editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation*, pages 493–508. Addison-Wesley, Boston, MA.

Hillis, W.D. (1990). Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42:228–234.

INTEL (accessed March 20, 2008). Tom's hardware's 2007 cpu charts.

Juille, H. and Pollack, J.B. (1995). Parallel genetic programming and fine-grained SIMD architecture. *Working Notes for the AAAI Symposium on Genetic Programming*, pages 31–37.

Koza, John R., Bennett III, Forrest H, Hutchings, Jeffrey L., Bade, Stephen L., Keane, Martin A., and Andre, David (1997). Rapidly reconfigurable field-programmable gate arrays for accelerating fitness evaluation in genetic programming. In Koza, John R., editor, *Late Breaking Papers at the 1997 Genetic Programming Conference*, pages 121–131, Stanford University, CA, USA. Stanford Bookstore.

Langdon, W. B. (2004). Global distributed evolution of L-systems fractals. In Keijzer, Maarten, O'Reilly, Una-May, Lucas, Simon M., Costa, Ernesto, and Soule, Terence, editors, *Genetic Programming, Proceedings of EuroGP'2004*, volume 3003 of *LNCS*, pages 349–358, Coimbra, Portugal. Springer-Verlag.

Langdon, W. B. (2008). Evolving GeneChip correlation predictors on parallel graphics hardware. In *Proceedings of the IEEE World Congress on Computational Intelligence*, Hong Kong. IEEE. Forthcoming.

Langdon, W. B. and Buxton, B. F. (2004). Genetic programming for mining DNA chip data from cancer patients. *Genetic Programming and Evolvable Machines*, 5(3):251–257.

Langdon, W. B. and Harrison, A. P. (2008). GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*. Special Issue. On line first.

Langdon, W. B. and Poli, Riccardo (2002). *Foundations of Genetic Programming*. Springer-Verlag.

Langdon, W.B. and Banzhaf, W. (2008). A SIMD interpreter for genetic programming on GPU graphics cards. In O'Neill, M., Vanneschi, L., Esparcia-Alcazar, A., and Ebner, M., editors, *Proc. 11th Europ. Conference on Genetic Programming, Valencia, Spain*, volume 4971 of *LNCS*, pages 70 – 79. Springer.

LibSh Wiki (accessed March 11, 2008). Libsh sample code. http://www.libsh.org/wiki/index.php/Sample_Code.

Mark, W., Glanville, S., and Akeley, K. (2003). Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics*.

Martin, Peter (2001). A hardware implementation of a genetic programming system using FPGAs and Handel-C. *Genetic Programming and Evolvable Machines*, 2(4):317–343.

Miller, J.F. and Thomson, P. (2000). Cartesian genetic programming. *Genetic Programming, Proceedings of EuroGP*, 1802:121–132.

Miller, L. et al. (2005). An expression signature for p53 status in human breast cancer predicts mutation status, transcriptional effects, and patient survival. *PNAS*, 102:13350 – 13355.

NVIDIA (accessed March 11, 2008). http://developer.nvidia.com/page/cg.

NVIDIA (November 2006). NVIDIA Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview.

Oussaidene, M., Chopard, B., Pictet, O.V., and Tomassini, M. (1996). Parallel genetic programming: An application to trading models evolution. *Genetic Programming*, pages 357–380.

Page, J., Poli, R., and Langdon, W. B. (1999). Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study. In Poli, Riccardo, Nordin, Peter, Langdon, William B., and Fogarty, Terence C., editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 39–49, Goteborg, Sweden. Springer-Verlag.

Poli, Riccardo (1996). Genetic programming for image analysis. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 363–368, Stanford University, CA, USA. MIT Press.

RapidMind (accessed March 11, 2008). Libsh. http://libsh.org/.

Robertson, G.G. and Riolo, R.L. (1988). A tale of two classifier systems. *Machine Learning*, 3:139–159.

Stoffel, Kilian and Spector, Lee (1996). High-performance, parallel, stack-based genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 224–229, Stanford University, CA, USA. MIT Press.

Tarditi, David, Puri, Sidd, and Oglesby, Jose (2006). Msr-tr-2005-184 accelerator: Using data parallelism to program GPUs for general-purpose uses. Technical report, Microsoft Research.

Turton, Openshaw, and Diplock (1996). Some geographic applications of genetic programming on the Cray T3D supercomputer. In Jesshope, Chris R. and Shafarenko, Alex V., editors, *UK Parallel'96*, pages 135–150, University of Surrey. Springer.

Upton, G.J.G., Langdon, W.B., and Harrison, A.P. (in preparation, 2008). GGGG probes in microarrays are not gene-specific.

Wikipedia (accessed March 11, 2008). http://www.wikipedia.org/wiki/flops.

Wilson, G. and Banzhaf, W. (2008). Linear Genetic Programming GPGPU on Microsoft's XBox 360. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2008), June 1-6, 2008, Hong Kong, China*.

XBOX (accessed March 20, 2008). Xbox 360 technical specifications.

Ziegler, Jens and Banzhaf, Wolfgang (2003). Decreasing the number of evaluations in evolutionary algorithms by using a meta-model of the fitness function. In Ryan, Conor, Soule, Terence, Keijzer, Maarten, Tsang, Edward, Poli, Riccardo, and Costa, Ernesto, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 264–275, Essex. Springer-Verlag.