Fast Genetic Programming and Artificial Developmental Systems on GPUs

Simon Harding and Wolfgang Banzhaf
Computer Science Department
Memorial University
Newfoundland, Canada
simonh, banzhaf@cs.mun.ca

Abstract

In this paper we demonstrate the use of the Graphics Processing Unit (GPU) to accelerate Evolutionary Computation applications, in particular Genetic Programming approaches. We show that it is possible to get speed increases of several hundred times over a typical CPU implementation, catapulting GPU processing for these applications into the realm of HPC. This increase in performance also extends to artificial developmental systems, where evolved programs are used to construct cellular systems. Feasibility of this approach to efficiently evaluate artificial developmental systems based on cellular automata is demonstrated.

1 Introduction

HPC has traditionally been executed on dedicated hardware. Often, special-purpose processors were designed by newly founded initiatives (see, for instance, [10], the history of the Connection Machine, or IBM's BlueGene project [1]) in order to produce the communication performance needed in parallel machines.

In recent years, however, off-the-shelf hardware has made inroads in this traditionally high-performance, high-budget field. With the advent of the LINUX operating system, it became possible to stack both inexpensive hardware and software, and to deliver large amounts of CPU cycles, as exemplified by the BEOWULF architecture [23]. Beowulf clusters are now installed around the world and provide a main driving force for information processing applications world-wide.

A new wave of commodity hardware is now on its way in the form of mass-produced graphics processors. Recently it has become possible to access the processing power of these graphic processing units (GPU) which serve as co-processors in most of the new generation desktop computers. For a general survey on algorithms implemented on GPUs the reader is referred to [18].

Modern GPUs are extremely good at performing parallel mathematical operations [25]. For example, discrete wavelet transformations [27], the solution of dense linear systems [7], physics simulations for games, fluid simulators [8], etc., have already been shown to execute faster on GPUs. However, until recently it was cumbersome to use this resource for general purpose computing. With the advent of easy-to-use programming tools for GPUs, a whole set of applications is presently being tested on GPUs. In effect, these tools open up a new world for HPC computing with commodity hardware.

The application area we are focussing on in this contribution is Evolutionary Computation [5, 17], and specifically Genetic Programming [2, 11]. In this field, it is well known that fitness evaluation is the most time consuming part of the process. This limits the types of problems that may be addressed by EC, and by GP in particular, as large numbers of fitness cases make GP runs often impractical. In this paper we demonstrate a method for using the GPU as an evaluator for genetic programming expressions, and show that there are considerable speed increases to be gained.

We briefly report on the evaluation of evolved mathematical expressions and digital circuits, as they are typically used to evaluate the performance of a genetic programming algorithm (a more detailed discussion can be found in [9]). We also investigate the use of the GPU to evaluate GP expressions used in a form of indirect genotype to phenotype mapping, called a developmental mapping. In a typical artificial cellular developmental system, there are a large number of cells, within an environment, that execute the same evolved program to produce the next state for the system.

Because capable hardware and software are new, there is relatively little previous work on using GPUs for evolutionary computation. For example [30] implements a evolutionary programming algorithm on a GPU, and finds that there is a 5-fold speed increase. Work by [6] expands on this, and evaluates expressions on the GPU. There all the operations are treated as graphics operations, which makes implemen-

21st International Symposium on High Performance Computing Systems and Applications(HPCS'07)
0-7695-2813-9/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

tation difficult and limits the flexibility of the evaluations. Yu et al [31], on the other hand, implement a Genetic Algorithm on GPUs. Depending on population size, they find a speed up factor of up to 20. Here both the genetic operators and fitness evaluation are performed on the GPU. Ebner et al, use human interaction to evolve aesthetically pleasing shader programs[4]. Here, linear genetic programming structures are compiled into shader programs. The shader programs were then used to render textures on images, which were selected by a user. However, the technique was not extended into more general purpose computation.

Any increase in the evaluation of evolved expressions will also benefit the emerging field of artificial developmental systems. In such systems, large numbers of the evolved expression might be executed in parallel, as an analogy to the cellular structure of organisms. In developmental systems, large multi celled solutions grow from smaller "embryo" systems.

The overview of the paper is as follows: Section 2 outlines the architecture of GPUs, Section 3 discusses the most important programming approaches to GPU processing. Section 4 briefly explains how Genetic Programming can be implemented on a GPU. Section 5 reports on running GP benchmarks on a GPU. In section 6 we show that cellular computation and chemical diffusion can be implemented efficiently using the GPU. Finally in section 7, we demonstrate the use of 2D ceullar automata on the GPU for generating high quality random numbers. This addresses an issues identified by [6], and would allow a complete evolutionary system to be executed on the GPU hardware, providing further performance gains.

2 The Architecture of Graphics Processing Units

Graphics processors are specialized stream processors used to render graphics. Typically, the GPU is able to perform graphics manipulations much faster than a general purpose CPU, as the graphics processor is specifically designed to handle certain primitive operations. Internally, the GPU contains a number of small processors that are used to perform calculations on 3D vertex information and on textures. These processors operate in parallel with each other, and work on different parts of the problem. First the vertex processors calculate the 3D view, then the shader processors paint this model before it is displayed. Programming the GPU is typically done through a virtual machine interface such as OpenGL or DirectX which provide a common interface to the diverse GPUs available thus making development easy. However, DirectX and OpenGL are optimized for graphics processing, hence other APIs are required to use the GPU as a general purpose device. There are many such APIs, and section 3 describes several of the more common ones.

For general purpose computing, we here wish to make use of the parallelism provided by the shader processors, see Figure 1. Each processor can perform multiple floating point operations per clock cycle, meaning that performance is determined by the clock speed and the number of pixel shaders and the width of the pixel shaders. Pixel shaders are programmed to perform a given set of instructions on each pixel in a texture. Depending on the GPU, the number of instructions may be limited. In order to use more than this number of operations, a program needs to be broken down into suitably sized units, which may impact performance. Newer GPUs support unlimited instructions, but some older cards support as few as 64 instructions. GPUs typically use floating point arithmetic, the precision of which is often controllable as less precise representations are faster to compute with. Again, the maximum precision is manufacturer specific, but recent cards provide up to 128-bit precision.

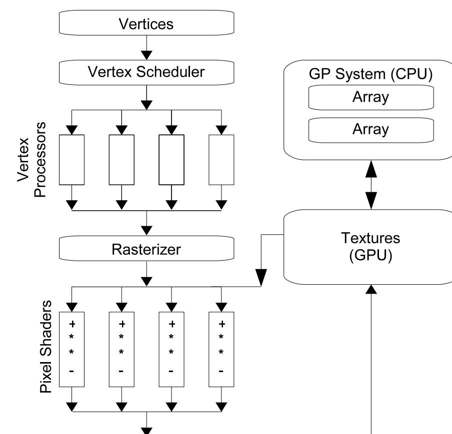

Figure 1. Arrays, representing the test cases, are converted to textures. These textures are then manipulated (in parallel) by small programs inside each of the pixel shaders. The result is another texture, which can be converted back to a normal array for CPU based processing.

The graphics card used in these experiments is a NVidia GForce 7300 Go, which is a GPU optimized for laptop use. It is underpowered compared to cards available for desktop PCs. Because GPUs are parallel and have very strict processing models, the computational ability of the GPU scales well with the number of pixel shaders. We would therefore expect to see major improvements to the performance of the benchmarks given here if we were to run it on such a GPU. According to [28], "an NVIDIA 7800 GTX 512 is capable

0-7695-2813-9/07 $20.00 © 2007 IEEE

of around 200 GFLOPS. ATI's latest X1900 architecture has a claimed performance of 554 GFLOPS". Since it is now possible to place multiple GPUs inside a single PC chassis, this should result in TFLOP performance for numerical processing at low cost. A further advantage of the GPU is that it uses less power than a typical CPU. Power consumption has become an important consideration in building clusters as it increases running costs and requires the use of air conditioning, or other cooling, increasing costs further.

3 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.

SH Sh is an open source project for accessing the GPU under C++ [19, 13]. 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.

Brook: Brook is another way to access the features on the GPU [22]. Brook takes the form of extensions to the C programming language, adding support for GPU specific data types. Applications developed with Brook are compiled using a special C compiler, which generates C++ and Cg code. Cg is a programming language for graphics, that is similar to C. One major advantage of Brook is that it can target either OpenGL or DirectX, and is therefore more platform independent than other tools. However, code must be compiled separately for each target platform. Brook appears to be a very popular choice, and is used for large applications, such as folding@home.

Accelerator: Recently a .Net assembly called Accelerator was released that provides access to the GPU via the DirectX interface [24]. 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.

This tool differs from the previous interfaces in that it uses lazy evaluation. 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, which 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.

In this paper we use the Accelerator package. The total time required to reimplement an existing parser tree based GP parser was less than two hours, which we would expect to be considerably less than using any of the other solutions presented here. As with other implementations, Accelerator is based on arrays implemented as textures. The API then allows one to perform parallel operations on the arrays. Conversion to textures, and transfer to the GPU is handled transparently by the API, allowing the developer to concentrate on the implementation of the algorithm. The available function set for operating on parallel arrays is similar to the other APIs. It includes element-wise arithmetic operations, square root, multiply-add, and trigonometric operations. There are also conditional operations and functions for comparing two arrays. The API also provides reduction operators, such as the sum, product, minimum or maximum value in the array. Further functions perform transformations, such as shift and rotate on the elements of the array.

The other systems described here present different variations on these functions, and generally offer functionality that allows different operations to be applied to different parts of the arrays.

4 Parsing a GP Expression

Typically parsing a GP expression involves traversing the expression tree in a bottom-up, breadth first manner. At each node visited the interpreter performs the specified function on the inputs to the node, and outputs the result as the node output. The tree is re-evaluated for every input set. Hence, for 100 test cases the tree would be executed 100 times.

Using the GPU we are able to parallelize this process, which means that in effect the tree only has to be parsed once - with the function evaluation performed in parallel. Even without the arithmetic acceleration provided by the GPU, this results in a considerable reduction in computation. Our GP interpreter uses a case statement at the evaluation of each node to determine what function to apply to the input values. If run on the GPU, the tree needs only to be executed once - removing the need for repeatedly accessing the case statement. The use of the GPU is illustrated in Figure 1. The population and genetic algorithm run on the CPU, with evaluations run on the GPU. The CPU converts arrays of test cases to textures on the GPU and loads a shader program into the shader processors. The Accelerator tool kit compiles each individuals GP expression into a shader program. The program is then executed, and the resulting texture is converted back in to an array. The fitness is determined from this output array.

The GP parser used here is written in C♯, and compiled

21st International Symposium on High Performance Computing Systems and Applications(HPCS'07)
0-7695-2813-9/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

using Visual Studio 2005. All benchmarks were done using the Release build configuration, and were executed on CLR 2.0 on Windows XP. The GPU is an NVidia GeForce 7300 GO with 512Mb video memory. The CPU used is an Intel Centrino T2400 (running at 1.83Ghz), with 1.5Gb of system memory.

5 Benchmarks

In this experiment, we investigate the speed up on both toy and real world problems, using a GP representation we chose to use here is Cartesian GP[16], but similar results should be obtained from other representations. Using CGP allowed us to strictly limit the length of the expressions, while retaining evolvabilty. This allowed us to vary the length of the expressions, and determine the effect on evaluation time. The evaluation time is the total time required to perform an evaluation, and therefore includes any set up time, moving of information to the GPU and the actual processing itself.

We implemented several types of fitness function, and then ran an evolutionary algorithm for 200 generations. The evolved expressions were evaluated on the CPU and then on the GPU, and each evaluation was timed for evaluation purposes. The results show the average number of times the GPU is faster at evaluating a given expression than the CPU. Results less than 1 mean that the CPU was faster at evaluating the expression, values above 1 indicate the GPU performed better.

5.1 Regression

We evolved functions that regressed over $x^6 - 2x^4 + x^2$ [11]. We tested the evaluation difference using a number of test cases. In each instance, the test cases were uniformly distributed between -1 to +1. We also changed the maximum length of the CGP graph. Hence, expression lengths could range anywhere from 1 node to the maximum size of the CGP graph. GP was run for 200 generations to allow for convergence. The function set comprised of $+$, $-$, $*$ and $/$. In C♯, division by zero on a float returns "Infinity", which is consistent with the result from the Accelerator library.

Fitness was defined as the sum of the absolute errors of each test case and the output of the expression. This can also be calculated using the GPU. Each individual was evaluated with the CPU, then the GPU and the speed difference recorded. Also the outputs from both the GPU and CPU were compared to ensure that they were evaluating the expression in the same manner. We did not find any instances where the two differed.

Table 1 shows results that for smaller input sets and small expressions, it was more efficient to evaluate them on the CPU. However, for the larger test and expression sizes the performance increase was dramatic.

	Test Cases			
Max Expression Length	10	100	1000	2000
10	0.02	0.08	0.7	1.22
100	0.07	0.33	2.79	5.16
1000	0.42	1.71	15.29	87.02
10000	0.4	1.79	16.25	95.37

Table 1. Results for the regression experiment. The results show the number of times faster evaluating evolved GP expressions is on the GPU, compared to CPU implementation. The maximum expression length is the number of nodes in the CGP graph.

5.2 Classification

In this experiment we attempted the classification problem of distinguishing between two spirals, as described in [11]. This problem has two input values (x and y coordinates of a point on a spiral) and has a single output indicating which spiral the point is found. In [11], 194 test cases are used. In these experiments, we extend the number of test cases to 388, 970 and 1940. We also extended the function set to include sin, cos, $\sqrt{x}$, x^y and a comparator. The comparator looks at the first input value to the node, and if it is less than or equal to zero returns the second input, 0 otherwise. The relative speed increases can be seen in Table 2. Again we see that the GPU is superior for larger numbers of test cases, with larger expression sizes.

5.3 Classification in Bioinformatics

In this experiment we investigate the behaviour on another classification problem, this time a protein classifier as described in [12]. Here the task is to predict the location of a protein in a cell, from the amino acids in the particular protein. We used the entire dataset as the training set. The set consisted of 2427 entries, with 19 variables each and 1 output. We investigated the performance gain using several expression lengths, and the results can be seen in Table 3. Here, the large number of test cases used results in considerable improvements in evaluation time, even for small expressions.

6 Cellular Developmental Systems

There are many forms of artificial developmental system, but here we limit our investigation to ones based on

	Test Cases			
Max Expression Length	194	388	970	1940
10	0.15	0.23	0.51	1.01
100	0.38	0.67	1.63	3.01
1000	1.77	3.19	9.21	22.7
10000	1.69	3.21	8.94	22.38

Table 2. Results for the two spirals classification experiment. The results show the number of times faster evaluating evolved GP expressions is on the GPU, compared to CPU implementation. The maximum expression length is the number of nodes in the CGP graph.

	Test Cases
Expression Length	2427
10	3.44
100	6.67
1000	11.84
10000	14.21

Table 3. Results for the protein classifcation experiment. The results show the number of times faster evaluating evolved GP expressions is on the GPU, compared to CPU implementation. The maximum expression length is the number of nodes in the CGP graph.

Number of cells	GPU Speed Up
256	0.21
4096	2.39
65536	20.54
262144	20.41
1048576	34.63

Table 4. Results showing the number of times faster evaluating diffusion on the GPU compared to the CPU.

	Number of cells				
Length	256	4096	65536	262144	1048576
10	0.73	1.01	5.99	14.94	22.26
25	0.75	1.02	5.58	14.54	23.23
50	0.77	1.04	5.76	14.13	23.97
100	0.79	1.11	5.91	15.47	28.12
1000	1.42	1.71	6.19	16.46	26.64

Table 5. Results showing the number of times faster evaluating the cellular automata rules using the GPU is than using the CPU. The cellular automata is a 2D square. The length is the maximum expression length of the CGP graph, and hence the maximum length of the expression. Where the speed up is above 1, the GPU is faster. We see that for large cellular automata, with large expressions that there is a significant increase in performance.

a cellular structure [3, 15, 20]. In cellular developmental systems, there are typically several cellular automata in operation. One executes rules evolved using genetic programming to simulate the cellular processes, others simulate the diffusion of chemicals. We show that using the GPU, both these types of cellular automata can be simulated with high performance gains.

Both cellular automata are implemented in similar ways. For each cell, a program (or set of rules) uses the current state of a cell, and its neighbourhood, to calculate the next state for that cell. Here we use a von Neumann neighbourhood of radius 1. For the diffusion cellular automata, each cell contains an amount of a given chemical, here represented as a floating point number. The input to the update program consists only of the values of its neighbouring cells. The cell states are represented as integers, with one value used to define an empty site. These programs require the state information of neighbouring cells and the state information of the chemical environment, as illustrated in figure 2. The update rule for the cell states is an evolved CGP program, in the same manner as used throughout this paper.

Each update program requires information about its neighbourhood, however there are no array operations provided by the API that allow access to other arrays with an offset. Therefore, several arrays are generated by shifting the state array up, down, left and right. This realigns the neighbouring cell information so that it can be conveniently used by the built in array operations.

For these benchmarks we performed short evolutionary runs (200 generations), using a simple fitness function (produce a specified cell pattern). Table 4 shows that for the larger environments, execution on the GPU gives a substan-

IEEE COMPUTER SOCIETY

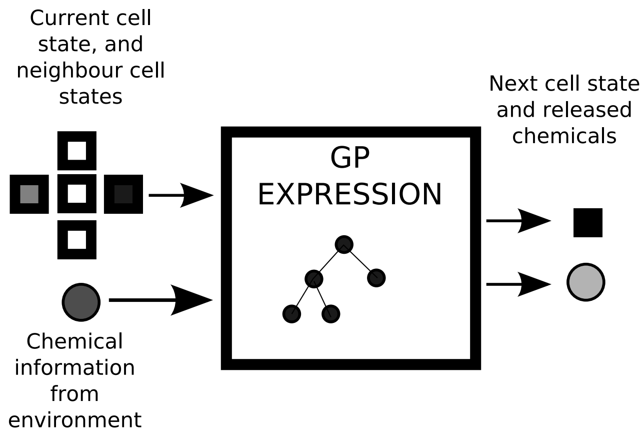

Figure 2. Each cell in the developmental system takes in information about its chemical environment and the states of the neighbouring cells. Using the evolved program, the next state for that cell is calculated. Iterating this process allows programs to grow into larger structures.

tial improvement in performance. Executing the GP expression produces similar results to those in section 5, . The results in table 5 show that for a larger number of cells and for longer programs, GPU execution is beneficial. We limited the expression length of the CGP program to 1000 nodes.

7 Random numbers using Cellular Automata

One function missing from GPU is the ability to generate random numbers. Random number generation is very important in evolutionary algorithms, as it forms the basis of the selection and recombination operators. One approach, used by [6], is for the CPU to produce randomly generated textures and pass them to the GPU. This however is a bottleneck in the system. We therefore suggest the use of cellular automata to produce high quality random numbers, directly in the GPU. Random sequences of bits can be generated using both 1D [29] and 2D[21, 14] cellular automata. We implemented a 2D cellular automata as described in [14]. The update rule consists of 3 XOR operations and 1 OR operation, that update the cells (a) for the next time step ($t + 1$):

$$a_{i,j}^{t+1} = (((a_{i,j}^{t}\ OR\ a_{i-1,j}^{t})\ XOR\ a_{i,j-1}^{t})\ XOR\ a_{i+1,j}^{t})\ XOR\ a_{i,j+1}^{t}$$

We found that producing the next state for a 1024x1024 texture using this cellular automata was 2.1 times faster than producing the random texture (using the C♯ standard random library) on the CPU and transferring it to the GPU. Random number generation took 42% of the CPU time, demonstrating that there is a signficant overhead to moving the information to the GPU.

8 Conclusions

This paper demonstrates that evaluation of genetic programming expressions can strongly benefit from using the graphics processor to parallelise the evaluations. With new development tools, it is now very easy to leverage the GPU for general purpose computation. However, there are a few caveats. Here we have reported on tests using Cartesian GP, however we expect similar advantages with other representations, such as tree and linear GP. We have also demonstrated the feasibility of using the GPU for artificial cellular developmental systems.

Few clusters are constructed with high performance graphics cards, which will limit the immediate use of these systems. It will require further benchmarking whether low end GPUs found in most PCs today provide a speed advantage. Given the computational benefits and the relatively low costs of fast graphics cards, it is likely that GPU acceleration for numerical applications will become widespread amongst lower priced installations.

Many typical GP problems do not have large sets of fitness cases for two reasons: First, evaluation has always been considered computationally expensive. Second, we currently find it very difficult to evolve solutions to harder problems. With the ability to tackle larger problems in reasonable time we have to also find innovative approaches that let us solve these problems. Traditional GP has difficulty with scaling. For example, the largest evolved multiplier has 1024 fitness cases [26]. In the same time it would take a CPU implementation to evaluate an individual with that many fitness cases, we could test more than 65536 fitness cases on a GPU. This leads to a gap between what we can realistically evaluate, and what we can evolve. Similarly for the developmental systems, the size of the environment used here is much larger than that typically found in the literature. However, we believe this approach makes experimentation using larger developmental systems more feasible.

For small sets of fitness cases, the overhead of transferring data to the GPU and for constructing shaders results in a performance decrease. It can be imagined that one would want to determine in practical applications when the advantage of GPU computing occurs and switch execution to the proper type of hardware. In this contribution, we have just looked at the most trivial way of parallelizing a GP system on GPU hardware. More sophisticated approaches to parallelisation will have to be examined in the future.

0-7695-2813-9/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

References

[1] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, et al. An Overview of the BlueGene/L Supercomputer. *Supercomputing, ACM/IEEE 2002 Conference*, pages 60–60, 2002.

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

[3] K. L. Downing. Developmental models for emergent computation. In A. M. Tyrrell, P. C. Haddow, and J. Torresen, editors, *International Conference on Evolvable Systems(ICES)*, volume 2606 of *Lecture Notes in Computer Science*, pages 105–116. Springer, 2003.

[4] M. Ebner, M. Reinhardt, and J. Albert. Evolution of vertex and pixel shaders. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, editors, *Proceedings of the Eighth European Conference on Genetic Programming (EuroGP 2005), Lausanne, Switzerland*, pages 261–270. Springer-Verlag, 2005.

[5] A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[6] K. L. Fok, T. T. Wong, and M. L. Wong. Evolutionary computing on consumer-level graphics hardware. *IEEE Intelligent Systems, to appear*, 2005.

[7] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 3–3, 2005.

[8] T. R. Hagen, J. M. Hjelmervik, K.-A. Lie, J. R. Natvig, and M. O. Henriksen. Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory*, 13:716–726, 2005.

[9] S. Harding and W. Banzhaf. Fast genetic programming on gpus. In *Proceedings EuroGP 2007*, LNCS. Springer, 2007, to appear.

[10] W. Hillis. *The Connection Machine*. MIT Press, 1989.

[11] J. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, Cambridge, Massachusetts, USA, 1992.

[12] W. B. Langdon and W. Banzhaf. Repeated sequences in linear genetic programming genomes. *Complex Systems*, 15(4):285–306, 2005.

[13] LibSh Wiki. Libsh sample code. `http://www.libsh.org/wiki/index.php/Sample_Code`.

[14] M. Madjarova, M. Kakuta, T. Obi, M. Yamaguchi, and N. Ohyama. Two-dimensional cellular automata for pseudorandom pattern generators and for highly secure stream ciphers. *Journal Optical Review*.

[15] J. F. Miller. Evolving a self-repairing, self-regulating, french flag organism. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. K. Burke, P. J. Darwen, D. Dasgupta, D. Floreano, J. A. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. M. Tyrrell, editors, *GECCO (1)*, volume 3102 of *Lecture Notes in Computer Science*, pages 129–139. Springer, 2004.

[16] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. P. et al., editor, *Proc. of EuroGP 2000*, volume 1802 of *LNCS*, pages 121–132. Springer-Verlag, 2000.

[17] M. Mitchell. *An Introduction to Genetic Algorithms*. Mit Pr, 1996.

[18] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. *Eurographics 2005, State of the Art Reports*, pages 21–51, 2005.

[19] RapidMind Inc. Libsh. `http://libsh.org/`.

[20] D. Roggen and D. Federici. Multi-cellular development: is there scalability and robustness to gain? In X. Yao, E. Burke, and J. L. et al., editors, *proceedings of Parallel Problem Solving from Nature 8, Parallel Problem Solving from Nature (PPSN) 2004*, pages 391–400, 2004.

[21] B. Shackleford, M. Tanaka, R. J. Carter, and G. Snider. Fpga implementation of neighborhood-of-four cellular automata random number generators. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 106–112, New York, NY, USA, 2002. ACM Press.

[22] Stanford University Graphics Lab. Brook. `http://graphics.stanford.edu/projects/brookgpu/`.

[23] T. Sterling, J. Salmon, D. Becker, and D. Savarese. *How to build a Beowulf: a guide to the implementation and application of PC clusters*. MIT Press Cambridge, MA, USA, 1999.

[24] D. Tarditi, S. Puri, and J. Oglesby. Msr-tr-2005-184 accelerator: Using data parallelism to program gpus for general-purpose uses. Technical report, Microsoft Research, 2006.

[25] C. Thompson, S. Hahn, and M. Oskin. Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis. In *Proceedings of the 35th International Symposium on Microarchitecture, Istanbul*, pages 306 – 317. IEEE Computer Society Press, 2002.

[26] J. Torresen. Evolving multiplier circuits by training set and training vector partitioning. In *ICES'03:From biology to hardware*, volume 2606, pages 228–237, 2003.

[27] J. Wang, P. A. H. T. T. Wong, and C. S. Leung. Discrete wavelet transform on gpu. In *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors*, pages C–41, 2004.

[28] Wikipedia. Flops — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=FLOPS&oldid=84987291`, 2006. [Online; accessed 1-November-2006].

[29] S. Wolfram. Random sequence generation by cellular automata. *Adv. Appl. Math.*, 7(2):123–169, 1986.

[30] M. L. Wong, T. T. Wong, and K. L. Fok. Parallel evolutionary algorithms on graphics processing unit. In *Proceedings of IEEE Congress on Evolutionary Computation 2005 (CEC 2005)*, volume 3, pages 2286–2293, 2005.

[31] Q. Yu, C. Chen, and Z. Pan. Parallel Genetic Algorithms on Programmable Graphics Hardware. *Lecture Notes in Computer Science*, 3612:1051, 2005.

0-7695-2813-9/07 $20.00 © 2007 IEEE