Fast Genetic Programming on GPUs

Simon Harding and Wolfgang Banzhaf

Computer Science Department, Memorial University, Newfoundland
{simonh,banzhaf}@cs.mun.ca
http://www.cs.mun.ca

Abstract. As is typical in evolutionary algorithms, fitness evaluation in GP takes the majority of the computational effort. In this paper we demonstrate the use of the Graphics Processing Unit (GPU) to accelerate the evaluation of individuals. We show that for both binary and floating point based data types, it is possible to get speed increases of several hundred times over a typical CPU implementation. This allows for evaluation of many thousands of fitness cases, and hence should enable more ambitious solutions to be evolved using GP.

Key words: Genetic programming, Graphics Card Acceleration, Parallel Evaluation

1 Introduction

It is well known that fitness evaluation is the most time consuming part of the genetic programming (GP) system. This limits the types of problems that may be addressed by GP, as large numbers of fitness cases make GP runs impractical. In some systems it is possible to accelerate the evaluation process using a variety of techniques. In this paper we present a method using the graphics processing unit on the video adapter. We study the evaluation of evolved mathematical expressions and digital circuits, as they are typically used to evaluate the performance of a genetic programming algorithm.

Different approaches have been used in the past for accelerating evaluation. For example, it is possible to co-evolve fitness cases in order to reduce the number of evaluations [1]. This, however, adds significant complexity to the algorithm, and does not guarantee an increase in performance under all circumstances. In other applications, one could select the number of fitness cases, e.g.. by stochastic sampling or other methods [2]. Should the system need to be tested against a complete input set, however, this approach would not be suitable. Another method involves compiling the evolved expression to executable code or even using binary code directly [3]. Writing expressions as native code or in a similar vain has many advantages [4]. The compiler or a hand-written algorithm can perform optimisations, e.g. by removing redundant code, which in addition to directly running the expression gives a significant increase in performance. The use of reflection in modern languages such as Java and C♯ provides for the possibility to compile and link code to the currently executing application.

Under some circumstances it is possible to offload the evaluation to more suitable hardware. When evaluating digital circuits, they can be loaded into a field programmable gate array (FPGA) and then executed on dedicated hardware [5]. This approach can provide large speed increases. However, the downloading of configurations into an FPGA can be a costly overhead. The biggest drawback to this approach is that it requires the use of external hardware, which may have to be specifically developed.

Recently it has become possible to access the processing power of the graphic processing unit (GPU). Modern GPUs are extremely good at performing parallel mathematical operations [6]. However, until recently it was cumbersome to use this resource for general purpose computing. For a general survey on algorithms implemented on GPUs the reader is referred to [7]. For example, discrete wavelet transformations [8], the solution of dense linear systems [9], physics simulations for games, fluid simulators [10], etc., have been shown to be executed faster on GPUs.

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. Using recent libraries we also show that putting the functions on the GPU to work is relatively painless. As many of these technologies are new, we include web links to sites containing the most recent information on the projects discussed.

Because capable hardware and software are new, there is relatively little previous work on using GPUs for evolutionary computation. For example [11] implements a evolutionary programming algorithm on a GPU, and finds that there is a 5-fold speed increase. Work by [12] expands on this, and evaluates expressions on the GPU. There all the operations are treated as graphics operations, which makes implementation difficult and limits the flexibility of the evaluations. Yu et al [13], 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[14]. 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.

To our knowledge, this contribution is the first study of general purpose Genetic Programming, executed on a graphics hardware platform. It makes use of the fact that GP fitness cases are numerous and can be executed in parallel. Provided there is a sufficient number of fitness cases (large datasets), a substantial speedup can be reached.

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 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.

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 [15], "an NVIDIA 7800 GTX 512 is capable 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, since it causes heat generation.

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

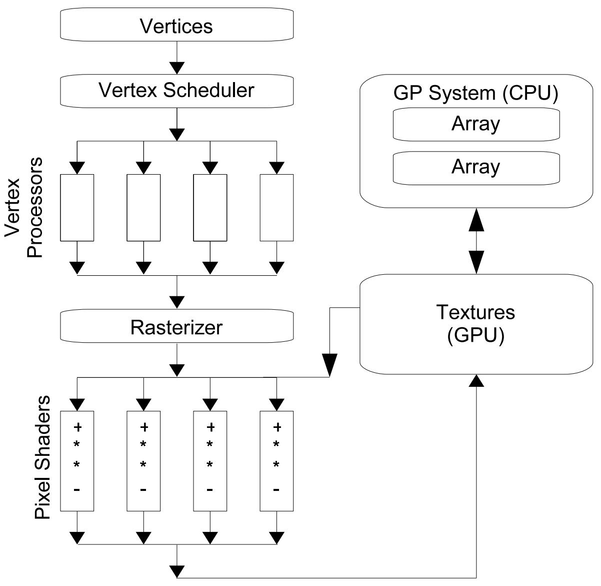

Fig. 1. Arrays, representing the test cases, are converted to textures. These textures are then manipulated (in parallel) by small programs inside each of the pixel shaders. The result is another texture, which can be converted back to a normal array for CPU based processing.

is intended to act as a guide to others. More information on these systems can be found at `www.gpgpu.org`.

SH Sh is an open source project for accessing the GPU under C++ [16, 17]. 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. In addition to providing general purpose computing, Sh also provides many routines for use in graphics programming. This feature is unique amongst the tools described here, and would be useful in visualisation of results.

Brook: Brook is another way to access the features on the GPU [18]. 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.

PyGPU: Another recent library allows the access of GPU functionality from the Python language [19]. PyGPU runs as an embedded language inside Python. The work is in its early stages, but results are promising. However it currently lacks the optimization required to make full use of the GPU. It requires a variety of extra packages to be installed into Python, such a NumPy and PyGame (which does not yet support the most recent Python release). Given the rise in popularity of Python for scientific computing, this implementation should prove useful in the future. Python itself, however, appears to have significant performance issues compared to C++ and JIT languages such as Java or C♯[1].

Accelerator: Recently a .Net assembly called Accelerator was released that provides access to the GPU via the DirectX interface [20]. 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.

[1] As usual, available benchmarks may not give a fair reflection to real world performance.

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.

5 Benchmarks

5.1 Configuration

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

In these experiments, GP trees were randomly generated with a given number of nodes. The expressions were evaluated on the CPU and then on the GPU, and each evaluation was timed for evaluation purposes. Timing was performed using calls to Win32 API QueryPerformanceCounter, which returns high precision timings. For each input size/expression length pair, 100 different randomly generated expressions were used, and results were averaged to calculate acceleration factors. Therefore our results show the average number of times the GPU is faster at evaluating a given tree size for a given number of fitness cases. Results less than 1 mean that the CPU was faster at evaluating the expression, values above 1 indicate the GPU performed better.

5.2 Floating point

In the first experiment, we evaluated random GP trees containing varying numbers of nodes, and exposed them to varying test case sizes. Mathematical functions $+$, $-$, $*$ and $/$ were used. The same expression was tested on the CPU and the GPU, and the speed difference was recorded. Results are shown in Table 1. For small node counts and fitness cases, the CPU performance is superior because of the overhead of mapping the computation to the GPU. For larger problems, however, there is a massive speed increase for GPU execution.

5.3 Binary

The second experiment compares the performance of the GPU at handling boolean expressions. In the CPU version, we use the C♯ boolean type - which is convenient, but not necessarily the most efficient representation. For the GPU, we tested two different approaches, one using the boolean parallel array provided by Accelerator, the other using float. The performance of these two representation is shown in Table 2. It is interesting to note that improvements are not guaranteed. As can be seen in the table, the speed up can decrease as expression size increases. We assume this is due to the way in which large shader programs are handled by either the Accelerator or the GPU. For example, the length of the shader program on the NVIDIA GPU may be limited, and going beyond this length would require repeated passes of the data. This type of behaviour can be seen in many of the results presented here.

We limit the functions in the expressions to AND, OR and NOT, which are supported by the boolean array type. Following some sample code provided with Accelerator, We mimicked boolean behavior using 0.0f as false, and 1.0f as true. For two floats, AND can be viewed as the minimum of the two values. Similarly OR can be viewed as the maximum of the two values. NOT can be performed as a multiply add, where the first stage is to multiply by -1 then add 1.

5.4 Real world tests

In this experiment, we investigate the speed up on both toy and real world problems, rather than on arbitrary expressions. The GP representation we chose to use here is CGP, but similar results should be obtained from other representations. CGP is fully described in [21]. In the benchmark experiments, the expression lengths were uniform throughout the tests. However, in real GP the length of the expressions vary throughout the run. As the GPU sometimes results in slower performance, we need to verify that on average, there is an advantage.

Regression We evolved functions that regressed over $x^6 - 2x^4 + x^2$ [22]. 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.

	Test Cases					
Expression Length	64	256	1024	4096	16384	65536
10	0.04	0.16	0.6	2.39	8.94	28.34
100	0.4	1.38	5.55	23.03	84.23	271.69
500	1.82	7.04	27.84	101.13	407.34	1349.52
1000	3.47	13.78	52.55	204.35	803.28	2694.97
5000	10.02	26.35	87.46	349.73	1736.3	4642.4
10000	13.01	36.5	157.03	442.23	1678.45	7351.06

Table 1. Results showing the number of times faster evaluating floating point based expressions is on the GPU, compared to CPU implementation. An increase of less than 1 shows that the CPU is more efficient.

Boolean implementation								
	Test Cases							
Expression Length	4	16	64	256	1024	4096	16384	65536
10	0.22	1.04	1.05	2.77	7.79	36.53	84.08	556.40
50	0.44	0.57	1.43	3.02	14.75	58.17	228.13	896.33
100	0.39	0.62	1.17	4.36	14.49	51.51	189.57	969.33
500	0.35	0.43	0.75	2.64	14.11	48.01	256.07	1048.16
1000	0.23	0.39	0.86	3.01	10.79	50.39	162.54	408.73
1500	0.40	0.55	1.15	4.19	13.69	53.49	113.43	848.29
Boolean implementation, using floating point								
	Test Cases							
Expression Length	4	16	64	256	1024	4096	16384	65536
10	0.024	0.028	0.028	0.072	0.282	0.99	3.92	14.66
50	0.035	0.049	0.115	0.311	1.174	4.56	17.72	70.48
100	0.061	0.088	0.201	0.616	2.020	8.78	34.69	132.84
500	0.002	0.003	0.005	0.017	0.064	0.25	0.99	3.50
1000	0.001	0.001	0.003	0.008	0.030	0.12	0.48	1.49
1500	0.000	0.001	0.002	0.005	0.019	0.07	0.29	1.00

Table 2. Results showing the number of times faster evaluating boolean expressions is on the GPU, compared to CPU implementation. An increase of less than 1 shows that the CPU is more efficient. Booleans were implemented as floating point numbers and as booleans. Although faster than the CPU for large input sizes, in general it appears preferential to use the boolean representation. Using floating point representation can provide speed increases, but the results are varied.

	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 3. 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.

	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 4. 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.

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 3 shows results that are consistent with the tests described in previous sections. 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.

Classification In this experiment we attempted the classification problem of distinguishing between two spirals, as described in [22]. 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 [22], 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 4. Again we see that the GPU is superior for larger numbers of test cases, with larger expression sizes.

Classification in Bioinformatics In this experiment we investigate the behaviour on another classification problem, this time a protein classifier as described in [23]. 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 5. Here, the large number of test cases used results in considerable improvements in evaluation time, even for small expressions.

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

Table 5. 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.

6 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 tested the system using Cartesian GP, however we expect similar advantages with other representations, such as tree and linear GP.

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 [24]. 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. The authors of this paper advocate developmental encodings, and using the evaluation approach introduced here we will be able to test this position.

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 kicks in 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.

Appendix: Code Examples

To demonstrate the ease of development, we include a small code sample showing the use of MS Accelerator from C♯. The first stage is to make arrays of the data to operate on. In a GP system these may be the fitness cases.

```
float[,] DataA = new float[4096, 4096];
float[,] DataB = new float[4096, 4096];
```

Next, the GPU has to be initialized, and the floating point arrays converted to parallel arrays:

```
ParallelArrays.InitGPU();
FloatParallelArray ParallelDataA =
new DisposableFloatParallelArray(DataA);
FloatParallelArray ParallelDataB =
new DisposableFloatParallelArray(DataB);
```

The parallel arrays are textures inside the GPU memory. Next, the shader program is specified by performing operations on the parallel arrays. However, the computation is not done until requested, as the shader program needs to be compiled, uploaded to the GPU shader processors and executed.

```
FloatParallelArray ParallelResult =
    ParallelArrays.Add(ParallelDataA, ParallelDataB);
```

Finally, we request that the expression is evaluated, and get the result from the GPU. The result is stored as a texture in the GPU, which needs to be converted back into a floating point array that can be used by the CPU.

```
float[,] Result = new float[4096, 4096];
ParallelArrays.ToArray(ParallelResult, out Result);
```

References

1. Lasarczyk, C., Dittrich, P., Banzhaf, W.: Dynamic subset selection based on a fitness case topology. Evolutionary Computation **12** (2004) 223–242
2. Banzhaf, W., Nordin, P., Keller, R., Francone, F.: Genetic Programming - An Introduction. Morgan Kaufmann, San Francisco, CA, USA (1998)
3. Nordin, P., Banzhaf, W., Francone, F.: Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover. In Spector, L., Langdon, W., O'Reilly, U.M., Angeline, P., eds.: Advances in Genetic Programming III, MIT Press, Cambridge, MA, USA (1999) 275–299
4. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer, New York, USA (2006)
5. Lau, W.S., Li, G., Lee, K.H., Leung, K.S., Cheang, S.M.: Multi-logic-unit processor: A combinational logic circuit evaluation engine for genetic parallel programming. In: EuroGP. (2005) 167–177

6. Thompson, C., Hahn, S., Oskin, M.: Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis. In: Proceedings of the 35th International Symposium on Microarchitecture, Istanbul, IEEE Computer Society Press (2002) 306 – 317
7. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A., Purcell, T.: A survey of general-purpose computation on graphics hardware. Eurographics 2005, State of the Art Reports (2005) 21–51
8. Wang, J., T. T. Wong, P.A.H., Leung, C.S.: Discrete wavelet transform on gpu. In: Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors. (2004) C–41
9. Galoppo, N., Govindaraju, N., Henson, M., Manocha, D.: Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference (2005) 3– 3
10. Hagen, T.R., Hjelmervik, J.M., Lie, K.A., Natvig, J.R., Henriksen, M.O.: Visual simulation of shallow-water waves. Simulation Modelling Practice and Theory **13** (2005) 716–726
11. Wong, M.L., Wong, T.T., Fok, K.L.: Parallel evolutionary algorithms on graphics processing unit. In: Proceedings of IEEE Congress on Evolutionary Computation 2005 (CEC 2005). Volume 3. (2005) 2286–2293
12. Fok, K.L., Wong, T.T., Wong, M.L.: Evolutionary computing on consumer-level graphics hardware. IEEE Intelligent Systems, to appear (2005)
13. Yu, Q., Chen, C., Pan, Z.: Parallel Genetic Algorithms on Programmable Graphics Hardware. Lecture Notes in Computer Science **3612** (2005) 1051
14. Ebner, M., Reinhardt, M., Albert, J.: Evolution of vertex and pixel shaders. In Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J., Tomassini, M., eds.: Proceedings of the Eighth European Conference on Genetic Programming (EuroGP 2005), Lausanne, Switzerland, Springer-Verlag (2005) 261–270
15. Wikipedia: Flops — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=FLOPS&oldid=84987291` (2006) [Online; accessed 1-November-2006].
16. RapidMind Inc: Libsh. (`http://libsh.org/`)
17. LibSh Wiki: Libsh sample code. (`http://www.libsh.org/wiki/index.php/Sample_Code`)
18. Stanford University Graphics Lab: Brook. (`http://graphics.stanford.edu/projects/brookgpu/`)
19. Lejdfors, C., Ohlsson, L.: Implementing an embedded gpu language by combining translation and generation. In: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, New York, NY, USA, ACM Press (2006) 1610–1614
20. Tarditi, D., Puri, S., Oglesby, J.: Msr-tr-2005-184 accelerator: Using data parallelism to program gpus for general-purpose uses. Technical report, Microsoft Research (2006)
21. Miller, J.F., Thomson, P.: Cartesian genetic programming. In et al., R.P., ed.: Proc. of EuroGP 2000. Volume 1802 of LNCS., Springer-Verlag (2000) 121–132
22. Koza, J.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT Press, Cambridge, Massachusetts, USA (1992)
23. Langdon, W.B., Banzhaf, W.: Repeated sequences in linear genetic programming genomes. Complex Systems **15**(4) (2005) 285–306
24. Torresen, J.: Evolving multiplier circuits by training set and training vector partitioning. In: ICES'03:From biology to hardware. Volume 2606. (2003) 228–237