M. O'Neill, L. Vanneschi, S. Gustafson, A.I. Esparcia Alcazar, I. De Falco,
A. Delea Cioppa, E. Tarantino, *eds.*, EuroGP 2008, 26-28 March, p73–85, LNCS 4971

A SIMD interpreter for Genetic Programming on GPU Graphics Cards

W. B. Langdon and Wolfgang Banzhaf

Mathematical and Biological Sciences University of Essex, UK
Computer Science, Memorial University of Newfoundland, Canada

Abstract. Mackey-Glass chaotic time series prediction and nuclear protein classification show the feasibility of evaluating genetic programming populations *directly* on parallel consumer gaming graphics processing units. Using a Linux KDE computer equipped with an nVidia GeForce 8800 GTX graphics processing unit card the C++ SPMD interpretter evolves programs at Giga GP operations per second (895 million GPops). We use the RapidMind general processing on GPU (GPGPU) framework to evaluate an entire population of a quarter of a million individual programs on a non-trivial problem in 4 seconds. An efficient reverse polish notation (RPN) tree based GP is given.

1 Introduction

Whilst modern computer graphics cards deliver extremely high floating point performance for personal computer gaming, the same low cost consumer electronics hardware can be used for desktop (and even laptop) scientific applications [Owens *et al.*, 2007]. However today's GPUs are optimised for a single program multiple data (usually abbreviated Single Instruction Multiple Data SIMD) mode of operation. GPU also place severe limits on data flow. Porting existing applications is non-trivial. Nevertheless [Fok *et al.*, 2007] were able to show speed ups from 0.62 to 5.02 when they ported evolutionary programming to a GPU. They ran EP mutation, selection and fitness calculation on their GPU. Each stage being done by fixed specially hand written GPU programs. [Harding and Banzhaf, 2007] were able to show far higher (peak) speed ups when they ran the fitness evaluation of cartesian genetic programming on a GPU. [Chitty, 2007] used Cg to precompile tree GP programs on the host CPU before transferring them one at a time to a GPU for fitness evaluation. Both groups obtained impressive speed ups by running many test cases in parallel. We demonstrate a SIMD interpreter which runs 204 800 programs simultaneously on the GPU on one or more test cases.

A decade ago [Juille and Pollack, 1996] demonstrated a SIMD GP system for a Maspar MP-2 super computer on a number of problems. The MP-2 was a general purpose supercomputer, costing in the region of $\$10^5$ in the mid 1990s. Its peak theoretical performance came from its many thousands of processing

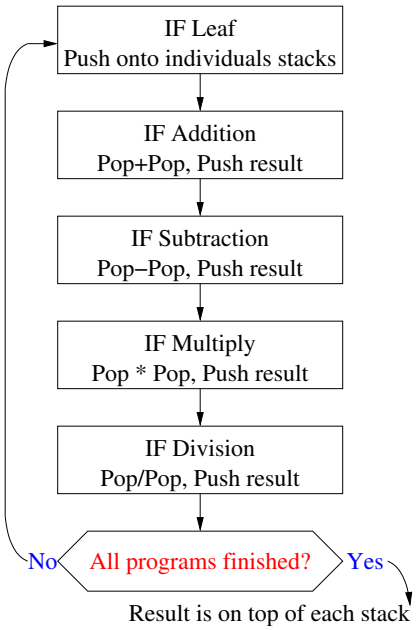

Fig. 1. The SIMD interpreter loops continuously through the whole genetic programming terminal and function sets for everyone in the population. GP individuals select which operations they want as they go past and apply them to their own data and their own stacks.

elements (PE) and the rapid bidirectional 2D data mesh interconnecting them. Jullie's coevolutionary problems were able to exploit the rapid transfer between neighbouring PEs. Less than 200 MP-2 were sold whereas a successful GPU typically has up to 128 independent processors and can be found in literally millions of homes. Even a top of the range GPU can be had for about £350.

In GPUs data describing scenes are imagined to flow into the processors, which transform them and transmit them onto the next processing stage (or to the user's screen). Typically recursion is not used. Part of the GPUs speed comes from specialising this data stream and avoiding the possibility of expensive side-to-side interaction. This restriction enables the GPU to schedule work freely without user intervention between the available processors. Indeed adding more processors can improve performance immediately without redesigning the application. However it makes it difficult to do some operations. The GPU should not be regarded as a "general purpose" computer. Instead it appears to be best to leave some (low overhead) operations to the CPU of the host computer.

Previously the parallelism of GPUs has been exploited by evaluating an individual's fitness by running it simultaneously on multiple training examples. Here we evaluate the entire GP population in parallel. Multiple training examples are not needed. How is this possible on a Single Instruction Multiple Data computer? Essentially the trick is to use one interpreter as the "single instruction"

stream and treat the programs it interprets as "multiple data" items. Figure 1 shows the essential inner loop of the SIMD interpreter. The loop runs on every computing element in the GPU. One complete cycle around the loop is used to evaluate each leaf and function in the GP tree. E.g. five instructions (push $+ - \times$ and $\div$) are needed for each primitive. In the SIMD interpreter, the role of the interpreted data item is to select which of the five is used. (The results of other four are discarded.) Effectively each GP individual acts as a sieve saying which operation it wants performed next. Whilst this introduces a new overhead, use of `cond` instructions to skip the four unwanted instructions and the speed of the GPU makes our approach viable. The SIMD interpreter can support more than four functions, but, in principle, the overhead increases with the size of the function set. While multi-ops, conditionals, loops, jumps, subroutines and recursion are possible, they are not included in these benchmarks.

The next section discusses some other previous parallel GP systems. The section following it discusses possible implementation avenues and why we chose RapidMind. This is followed by descriptions of our two benchmarks (Sections 4 and 5). Whilst Section 6 describes the performance of the interpreter in practise and relates it to other work. This is followed by a discussion, future work (Section 7) and our conclusions (Section 8).

2 Parallel Genetic Programming

While most GP work is conducted on sequential computers, the algorithm typically shares with other evolutionary computation techniques at least three computationally intensive features, which make it well suited to parallel hardware. 1) Individuals are run on multiple independent training examples. 2) The fitness of each individual could be calculated on independent hardware in parallel. 3) Lastly sometimes experimenters wish to assign statistical confidence to the stochastic element of their results. This typically requires multiple independent runs of the GP. The, comparative, ease with which EC can exploit parallel architectures has lead to the expression "embarrassingly parallel".

Early work includes Ian Turton's use of a GP written in Fortran running on a Cray super computer [Turton *et al.*, 1996]. Koza popularised the use of Beowulf workstation clusters where the population is split into separately evolving demes with limited emigration between compute nodes [Andre and Koza, 1996; Bennett III *et al.*, 1999] or workstations [Page *et al.*, 1999]. Indeed as [Chong and Langdon, 1999; Gross *et al.*, 2002] showed by using Java and the Internet, the GP population can be literally spread globally. Alternatively JavaScript can be used to move interactive fitness evaluation to the user's own home but retain elements of a centralised population [Langdon, 2004].

Others have used special purpose hardware. For example, while [Eklund, 2003] used a simulator, he was able to show how a linear machine code GP might be run very quickly on a field programmable gate array using VHDL to model sun spot data. However his FPGA architecture is distant from a GPU.

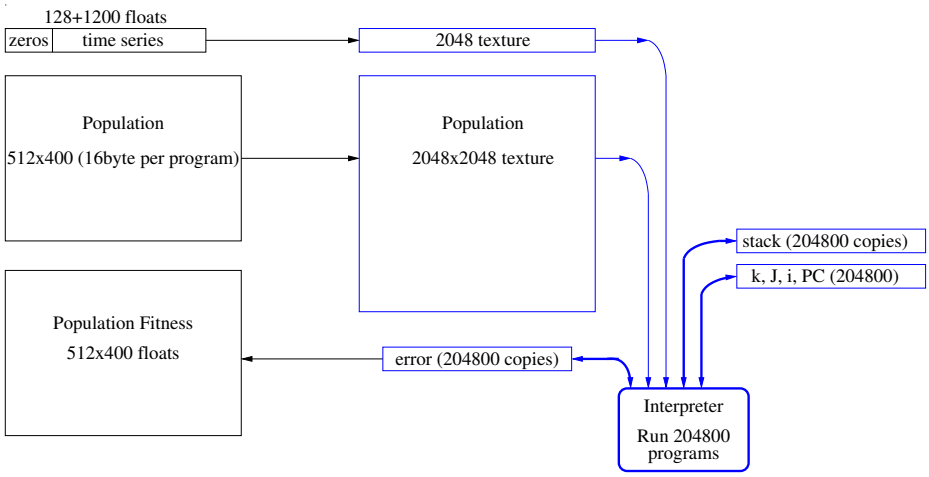

Fig. 2. Major data structures for Mackey-Glass. At the start of the run the interpreter is compiled on the CPU (left hand side). It and the training data are loaded onto the GPU (righthand side). Every generation the whole population is transfered to the GPU. Each individual is interpreted using its own stack and local variables (k, J, i, PC) and its RMS error is calculated. The error is is used as the programs' fitness. All transfers are made automatically by RapidMind.

In summary GP can and has been parallelised in multiple ways to take advantage both of different types of parallel hardware and of different features of particular problem domains. We propose a new way to exploit the inherent parallelism available in modern low cost mass market graphics hardware. Towit a GP SIMD interpreter for GPUs.

3 Programming Graphics Cards

Perhaps unsurprisingly the first uses of graphics processing units (GPUs) with genetic programming were for image generation [Ebner *et al.*, 2005] & its refs.

[Harding and Banzhaf, 2007, Section 3] described the various major high level language tools for programming GPUs (Sh, Brook, PyGPU and microsoft Accelerator). nVidia has two additional tools: CUDA and Cg (C for graphics [Fernando and Kilgard, 2003]). CUDA is specific to nVidia's GPUs. While Sh [McCool and Du Toit, 2004] is still available from SourceForge, its development is effectively frozen at Sh 0.8.0 and McCool recommends using its replacement from RapidMind. Unlike Sh, RapidMind is not free, however www.rapidmind.net issues licences, code, tutorials and documentation to developers. They host a developers' forum and offer prompt and effective support. Like Sh, RapidMind is available for both microsoft directX and unix OpenGL worlds and is not tied to a particular manufacturer's GPU hardware. Indeed recently they started to support parallel programming on the cell processor. However C++ code written for RapidMind's libraries is not portable to other systems. Another nice feature

of RapidMind is that it frees the C++ programmer from the need to learn graphics jargon and conceals many hardware limitations.

4 Mackey-Glass

The Mackey-Glass chaotic time series is described in [Langdon and Banzhaf, 2005b; Langdon and Banzhaf, -]. Briefly the GP is given historical data from a series of 1200 points one time step apart and asked to predict the next value. It is allowed to see data up to 128 time steps in the past. Figure 2 and Table 1 describe our implementation.

Table 1. GPU GP Parameters for Mackey-Glass time series prediction.

Function set:	ADD SUB MUL DIV operating on floats
Terminal set:	Registers are initialised with historical values of time series. D128 128 time steps ago, D64 64, D32 32, D16 16, D8 8, D4 4, D2 2 and finally D1 with the previous value. Time points before the start of the series are set to zero (cf. zeros top of Figure 2). Constants 0, 0.01, 0.02, ... 1.27
Fitness:	RMS error
Selection:	fine grained binary tournament demes [Langdon, 1998], non elitist, Population size 512×400
Initial pop:	ramped half-and-half 1:3 (depth 1 to 3. 50% of terminals are constants)
Parameters:	50% subtree crossover. 50% mutation (point 22.5%, constants 22.5%, subtree 5%). Max tree size 15, Max tree depth 4.
Termination:	50 generations

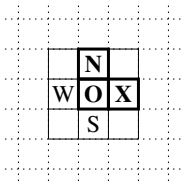

Fig. 3. The GP population is spread one per grid square in two dimensions. If **N**orth is better than **O**rigin, it is copied over it. But if **O**rigin is better, **O** is copied over **N**. (No change if equally fit.) After selection, crossover may occur between **O** and **X**. To promote mixing, 50% of crossovers swap which parent supplies the root node, so a child produced by crossover is equally likely to inherit its root from either parent. Also the neighbourhood pairing rotates 90° every generation. E.g. next generation, crossover will be between **O** and **S**.

4.1 Fine Grained Diffusion Model of Overlapping Demes

While it is not needed for operation with GPU, we used a fine grained diffusion model of overlapping demes [Langdon, 1998], see Figure 3. This allows a low selection pressure and ready visualisation, cf. Figure 4.

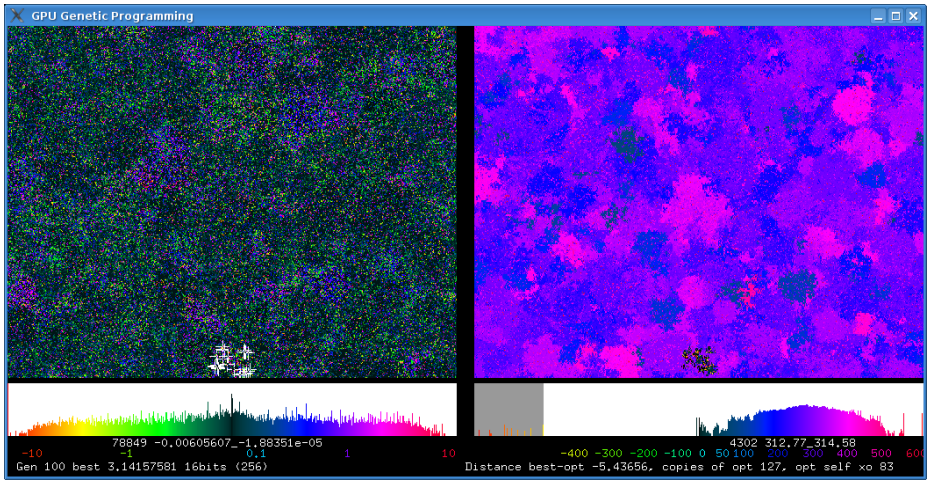

Fig. 4. Screen shot of 512×400 GP population after 100 generations. Colour indicates fitness (left) and syntax (right). Below are two histograms (log scale) showing distribution of population by fitness and genotypic distance from the first optimal solution. (Colour scales below each histograms.) Crossover is producing large numbers of unfit leafs (vertical lines at 540 and 600) [Poli *et al.*, 2007]. Local convergence and the production of species is visible (esp. right). See http://www.cs.ucl.ac.uk/staff/W.Langdon/pi2_movie.html and Google videos for animation and more explanation.

4.2 Subtree Crossover and Mutation

In these experiments, the crossover and mutation rates were chosen so that all of the next population are produced either by crossover or mutation (but not both). This ensures almost all children are different from their parents.

Koza's [Koza, 1992] crossover was implemented for linearised reverse polish notation. However there is no bias towards using functions rather than terminals as crossover points. If a pair of crossover points would cause either offspring to be too big or too deep, both are rejected and a new pair chosen again.

One of three types of mutation are used: subtree mutation, point mutation and constant creep mutation. In subtree mutation a subtree is chosen uniformly at random and replaced with a subtree created by the ramped half-and-half (depth 0:1, i.e. leaf or 1 function+2 leafs) algorithm used to create the initial population. If the mutation point is already at the maximum depth, then the subtree is replaced by a randomly chose leaf. If the mutant tree is too big it is rejected and the mutation process restarted with a newly chosen mutation point.

Point mutation does not change the size or shape of the parent tree. A mutation point is uniformly chosen and replaced by a function or leaf with the same arity using the same random selection technique as was used in the initial population. Repeated mutations are applied until, the mutated tree is syntactically different from its parent.

Table 2. Mackey-Glass prediction error after 50 generations in ten runs (multiplied by 128 as was used in [Langdon and Banzhaf, 2005a]).

											Mean
RMS error×128	4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69
Solution size	9	11	9	9	13	9	9	9	9	9	9.6
Run time secs	167.3	168.0	167.5	167.5	167.3	167.4	167.5	167.5	167.5	167.6	167.5

In constant creep mutation, one of the constant leafs in the tree is chosen at random. (If there are no constants, point mutation is used instead.) It is changed by just enough to give the next constant's value. (I.e. by ± 0.01 in the Mackey-Glass experiments).

4.3 Mackey-Glass Model Accuracy

The results of ten independent GP runs on the GPU are summarised in Table 2. The tight limit on tree size (15) and depth (4) lead to similar but smaller solutions than those reported for tree GP [Langdon and Banzhaf, 2005a, Table 2]. In 4 of 10 cases the results are better than the ten FXO (i.e. the smallest and fastest) subtree runs. The GPU GP runs are faster than all but two CPU runs despite having a population more than 400 times as big and performing full floating point calculations rather than 8 bit integer ones.

Table 3. GPU SIMD GP Parameters for protein localisation.

Function set:	ADD SUB MUL DIV operating on floats
Terminal set:	Number (integer) of each of the 20 amino acids in the protein. (Codes B and Z are ambiguous. Counts for code B were split evenly between aspartic acid D and asparagine N. Those for Z, between glutamic acid E and glutamine Q.) 128 unique constants chosen from tangent distribution (50% between -10.0 and 10.0)
Fitness:	$\frac{1}{2}$True Positive rate + $\frac{1}{2}$True Negative rate [Langdon and Barrett, 2004]
Selection:	fine grained binary tournament demes [Langdon, 1998], non elitist, Population size 1024×1024
Initial pop:	ramped half-and-half 2:5 (50% of terminals are constants)
Parameters:	50% subtree crossover. 50% mutation (point 22.5%, constants 22.5%, subtree 5%). Max tree size 63, Max tree depth 8.
Termination:	1000 generations

5 Evolving a Million Individuals for 1000 Generations Protein Location Prediction

The system was expanded to cope with: 1) a population of a million programs. 2) bigger trees. 3) deeper trees. 4) Randomised sub-selection of training cases. (See Table 3.) The task chosen was to predict the location of proteins within the cell given only their amino acid composition [Langdon and Banzhaf, -]

[Harding and Banzhaf, 2007]. A 1024 by 1024 population of programs of up to 63 tree elements and maximum depth of 8 was run on 200 of 1213 randomly chosen proteins selected for training. Compared to [Langdon and Banzhaf, -, Table 5], in terms of predictive accuracy on unseen proteins (cf. Figure 5) this run produced better results than one technique (FXO) and the same accuracy but a smaller solution than the other technique (two point crossover, 2XO). However the main point is a graphics card can readily evolve millions of GP programs over thousands of generations.

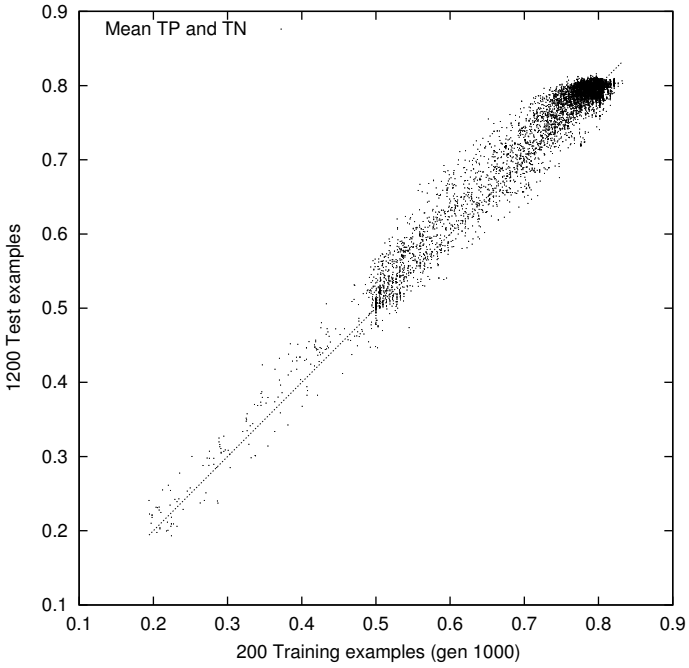

Fig. 5. Fitness on 200 randomly chosen training cases in generation 1000, versus fitness on 1200 unseen proteins The strong correlations shows GP has learnt for random samples and (better yet) GP models have avoided over fitting.

6 Performance of SIMD Interpreter

6.1 Overhead of Opcode Selection

The interpreter's performance is summarised in Table 4.

We wished to estimate the overhead of the SIMD loop scheduling all of the primitives and then discarding the results of all but the 20% that are needed. To do this we selected a typical evolved Mackey-Glass program and timed how long it took the interpreter to run it. Secondly we hand build an version of the interpreter specific for this program, where every operation is needed and no

Table 4. Speed (millions GP operations/sec) of GPU interpreter on an nVidia GeForce 8800 GTX. Terminal sets $\mathcal{T}$ include inputs and 128 constants.

Experiment	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{F}\rvert$	Population	program size	test cases	Speed (M GPops^{-1})
Mackey-Glass	8+128	4	204 800	11.0	1200	895
Mackey-Glass	8+128	4	204 800	13.0	1200	1056
Protein	20+128	4	1 048 576	56.9	200	504
Laser$_a$	3+128	4	18 225	55.4	151 360	656
Laser$_b$	9+128	8	5 000	49.6	376 640	190

results are discarded. Rather than the expected five to one ratio, the standard SIMD interpreter is only 2.89 times slower than the specialised one.

A plausible explaination is that: on the GPU floating point operations such as addition and multiplication, which form the GP function set, are extremely fast. It is the GP terminals (which make up 54% of the program) which take longer since they collect the data. The functions only manipulate data already on the stack. This asymmetry in the costs of items in the SIMD dispatch loop means the addition of a few very fast operations has proportionately less impact than was expected.

Potentially this means we could expand the function set to include trigonometry, log, exponentiation, etc. Many of these are directly implemented by the GPU. While increasing the function set would not be free, the additional overhead should be small.

6.2 GPU Speed up

The Mackey-Glass interpreter was recoded with minimum changes to run on the CPU. A 2211MHz AMD Athlon 64 Processor 3500+ CPU evolved 50 gens of a population of 204 800 trees in 1129.59 seconds. I.e. 7 times longer than the GPU.

7 Discussion

In previous work [Harding and Banzhaf, 2007] used the GPU exclusively for running training cases for cartesian genetic programming and showed impressive speed up in some cases but that improvement was highly variable. Indeed using the GPU was slower than the CPU in a few cases. GP program size and number of training examples per fitness evaluation appear to be critical. We have shown a way of actually executing a traditional tree GP population on the GPU card. It replaces the cost of compiling each member of the population on the CPU by the overhead of running an interpreter on the GPU. Harding's results mostly show that the GPU gives a big performance gain where the compiled GP program is run many times and the programs are large. However if programs are run few times the cost of the compiler and transfer to the GPU may not be repaid. There appears to be a nonlinearity (perhaps in the cost of starting the compiler) so that the relatively small cost of running short programs appears large compared to the cost of compiling them and transferring them to

the GPU. With our more traditional interpreter approach, the population is transfered without compilation overhead to the GPU and the speedup from running in parallel on the GPU appears to be more consistent. We obtain a speed up of more than an order of magnitude for very small programs.

7.1 Implementation Issues

We found that the GPU would give good performance if it was given reasonable chunks of work to do. Say between 1 and 10 seconds. Then the time to transfer the population and the training data into the GPU and fitness vector out is ok.

RapidMind on a Linux platform uses the GNU C++ compiler GCC and GDB debugger. Unfortunately GCC's error reporting can be hard to interpret since RapidMind (like Sh) makes heavy use of templates. RapidMind's cross compiler for the GPU worked seamlessly.

In Sections 4 and 5 the interpreter explicitly loops through all the fitness cases. The GPU can also vectorise computation across cases. We did this recently for a small population and executed 50 000 cases $\times$ pop size in parallel. However, we anticipate it usually remains better to loop through the test cases and so reduce data communication and concentrate computation in fewer threads.

7.2 No protected division: Closure

Special cases, like divide by zero, are handled by non data values *nan* and *inf*. The interpreter does not check for divide by zero. Undoubtedly this makes it faster. In effect, the GPU's floating point hardware supplies closure for us.

However we still need to be wary. Potentially large numbers of randomly generated programs, or even offspring of evolved individuals, may have invalid fitness. Filling the population with them may inhibit or even prevent GP successfully evolving.

7.3 Reverse Polish Notation expression Stack Depth

The GPU does not allow arbitrary *write* access to large arrays. Indeed forcing the data flow out of the GPU to be streamlined is required to enable tasks to be easily shared between the 128 processors and so is partly responsible for the GPUs speed. However it does make it difficult to implement a stack. Therefore it was necessary to simulate a stack using joins. ([Ernst *et al.*, 2004] suggest a somewhat complicated way to implement a GPU stack. It requires at least two passes. [Lefohn *et al.*, 2006] use Cg to efficiently implement a stack. Neither approach is feasible in RapidMind 2.0.1.) Joins work fine for small stacks. Indeed with a stack depth of 4 the interpreter flew at more than a billion GP primitives per second (speed up of more than 12). When the depth was doubled to 8 (for the Mackey-Glass and protein prediction experiments) it imposed about a 30% performance penalty. It appears that a stack limit of 12 or 16 would be feasible. While this may seem restrictive, it is worth remembering that all the original GP experiments [Koza, 1992] were conducted in Lisp with a depth limit of 17.

7.4 Non-Tree GP, GP without a stack

If the compilation overhead is too heavy, our interpreter approach may be attractive. It could be readily applied to cartesian GP [Harding and Banzhaf, 2007] and to linear GP. Typically both approaches use a small number of registers and do not require the use of a stack. Hence a linear genetic program could be interpreted directly on a GPU without incurring the stack overhead or consequent depth limit.

We have deliberately limited ourselves to demonstrating a traditional tree GP actually running on the GPU. We have been prepared to pay the overhead of the instruction loop scheduling one thing at a time. However evolution can often take advantage of muddled situations. We could imagine an evolutionary system in which the program did not wait for exactly the next required instruction to come around. But instead the program could say I will take the result of several instructions, whichever is scheduled first. This might be implemented by the interpreter looking for any bit in a bit mask being set, or an opcode lying in some range, or some other form of fuzzy match between what the program wants and what the interpreter is doing now. Of course the order of the actions of the interpreter might also be evolved. While this form of coevolution is unlikely to yield immediate speed ups on today's problems, it might be a route to meta evolution on more interesting problems in future.

7.5 Possible extensions

We have shown reliable speed ups can be obtained using a SIMD interpreter to execute a GP population on the GPU. [Fok *et al.*, 2007] have already shown (albeit for EP) that a GPU can implement mutation and selection. Although genetic programming mutation is more complex, we anticipate it too could be implemented on the GPU. Indeed, although [Fok *et al.*, 2007] shied away from crossover, We expect GP crossover could also be performed by the GPU. As GPUs continue to improve, the whole GP may be run by them.

8 Conclusions

By using a postfix (RPN) rather than a prefix (Lisp) representation, we have replaced recursive calls by an explicit stack. Avoiding recursion and using `cond` to select opcodes enabled us to run tree genetic programming with mega populations actually on the GPU. Speed up depends on terminal set, training set size, etc. but parallel operation can yield a speed up of 7–12. Typically a modern GPU interprets hundreds of millions of GP operations per second. Indeed in one case, we exceeded a billion GP ops per second. This is about **0.1 peta GP opcodes per day for $500**.

The SIMD interpreter could be readily adapted to linear GP. Indeed a linear GP system would avoid the overheads associated with simulating a stack. It might be possible to extended it to other types of GP.
C++ code available `ftp://cs.ucl.ac.uk/genetic/gp-code/gpu_gp_1.tar.gz`

Acknowledgements

We would like to thank Simon Harding, Nolan White, Paul Price (MUN) and Joanna Armbruster and Nick Holby. Experiments run at Memorial University.

References

Andre and Koza, 1996. A parallel implementation of genetic programming that achieves super-linear performance. In H. R. Arabnia, ed., *Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Apps.*, pp1163–1174. CSREA.

Bennett III *et al.*, 1999. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In Banzhaf *et al.*, eds., *GECCO-99*, pp1484–1490. pdf

Chitty, 2007. A data parallel approach to genetic programming using programmable graphics hardware. In Dirk Thierens *et al.*, eds., *GECCO-07*, pp1566–1573. ACM.

Chong and Langdon, 1999. Java based distributed genetic programming on the internet. In Banzhaf *et al.*, eds., *GECCO-99*, p1229. Full text in CSRP-99-7.

Ebner *et al.*, 2005. Evolution of vertex and pixel shaders. In M. Keijzer *et al.*, eds., *EuroGP-2005*, *LNCS* 3447, pp261–270. Springer. doi

Eklund, 2003. Time series forecasting using massively parallel genetic programming. In *Proc. of Parallel and Distributed Processing Int. Symposium*, pp143–147. doi

Ernst *et al.*, 2004. Stack implementation on programmable graphics hardware. In B. Girod, *et al.*, eds., *Proc. Vision, Modeling, and Visualization Conference*,255–262

Fernando and Kilgard, 2003. *The Cg Tutorial*. Addison-Wesley.

Fok *et al.*, 2007. Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, 22(2):69–78. doi

Gross *et al.*, 2002. R. Gross, K. Albrecht, W. Kantschik, and W. Banzhaf. Evolving chess playing programs. In W. B. Langdon *et al.*, eds., *GECCO-02*, pp740–747.

Harding and Banzhaf, 2007. Fast genetic programming on GPUs. In M. Ebner *et al.*, eds., *EuroGP-2007*, *LNCS* 4445, pp90–101. Springer. doi

Juille and Pollack, 1996. Massively parallel genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., eds., *Advances in GP 2*, pp339–358. MIT Press. pdf

Koza, 1992. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

Langdon and Banzhaf, 2005a. Repeated patterns in tree genetic programming. In M. Keijzer *et al.*, eds., *EuroGP-2005*, *LNCS* 3447, pp190–202. Springer. pdf

Langdon and Banzhaf, 2005b. Repeated sequences in linear genetic programming genomes. *Complex Systems*, 15(4):285–306. pdf

Langdon and Banzhaf, -. Repeated patterns in genetic programming. *Natural Computation*. doi:10.1007/s11047-007-9038-8.

Langdon and Barrett, 2004. Genetic programming in data mining for drug discovery. In A. Ghosh and L. C. Jain, eds., *Evolutionary Computing in Data Mining*,211-235.

Langdon, 1998. *Genetic Programming and Data Structures*, Kluwer.

Langdon, 2004. Global distributed evolution of L-systems fractals. In M. Keijzer *et al.*, eds., *EuroGP'2004*, *LNCS* 3003, pp349–358. Springer. pdf

Lefohn *et al.*, 2006. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, January 2006.

McCool and Du Toit, 2004. *Metaprogramming GPUs with Sh*. AK Peters.

Owens *et al.*, 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113.

Page *et al.*, 1999. Smooth uniform crossover with smooth point mutation in genetic programming. In R. Poli *et al.*, eds., *EuroGP'99*, *LNCS* 1598, pp39–49. Springer.

Poli *et al.*, 2007. On the limiting distribution of program sizes in tree-based genetic programming. In M. Ebner *et al.*, eds., *EuroGP-2007*, *LNCS* 4445, pp193–204.

Turton *et al.*, 1996. Some geographic applications of genetic programming on the Cray T3D supercomputer. In C.Jesshope and A.Shafarenko, eds., *UK Parallel'96*. Springer.