A SIMD interpreter for Genetic Programming on GPU Graphics Cards

W. B. Langdon

Departments of Mathematical and Biological Sciences, University of Essex, UK
3 July 2007, Computer Science Technical Report CSM-470 ISSN 1744-8050

When you are in the swamp killing alligators, the thing to remember is that you are not supposed to be killing alligators; you are supposed to be draining the swamp.

Abstract

Mackey-Glass chaotic time series prediction and non-nuclear protein classification show the feasibility of evaluating genetic programming populations on SPMD parallel computing consumer gaming graphics processing units. The C++ framework with a regular disk less Linux KDE desktop equipped with a single leading nVidia GeForce 8800 GTX graphics processing unit card is demonstrated evolving programs at Giga GP operation per second (895 million GPops). The RapidMind general processing on GPU (GPGPU) framework supports evaluating 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.

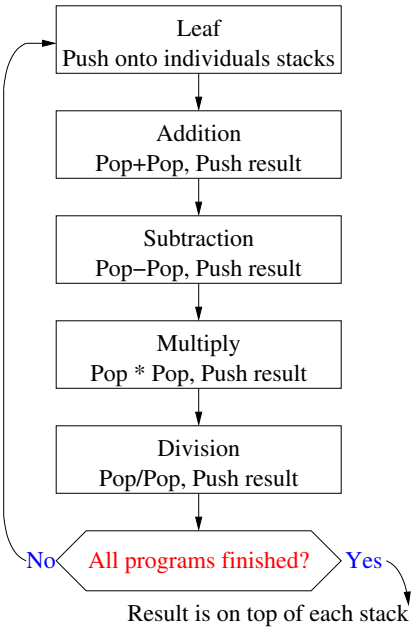

Figure 1: The SIMD interpreter loops continuously through the whole genetic programming terminal and function sets for everyone in the population. GP individuals select which operations they want as they go past and apply them to their own data and their own stacks. Unwanted results are discarded. When a required instruction is executed by a program, the program moves onto waiting for its next operation. If branches, goto jumps, loops and function calls could be implemented. Boolean and very short integer operations can be implemented by lookup tables. Combining tables for different operations reduces the number of options in the loop and so could make GP faster.

1 Introduction

Whilst modern computer graphics card 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. 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 he ran the fitness evaluation of a special version of genetic programming on a GPU. The impressive speed ups were obtained by running multiple test cases in parallel. We demonstrate a SIMD interpreter which runs 204 800 programs simultaneously on the GPU on one or more test cases.

Our approach uses a high level language (C++) which imposes some overhead compared to the conventional assembler approach to programming GPUs. See Figures 2 and 3. Similarly conventional interpreted high level languages (such as Java, Perl, JavaScript) have varying degrees of overhead compared to C or C++. Likewise each GP interpreter is usually also associated with an overhead.

Juille demonstrated a SIMD GP system for a Maspar MP-2 super computer on a couple of problems [Juille and Pollack, 1996]. 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 elements (PE) and the rapid bidirectional 2D data mesh interconnecting them. Jullie's coevolutionary problems were able to exploit the rapid transfer between neighbouring PE. Less than a couple of hundred 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 less than £400.

GPUs have no notion of neighbouring processors. Data describing scenes are imagined to flow into the processors, which transform them and transmit them onto the next processing stage (or the user's screen). 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 decide 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 (GPGPU). Instead it appears to be best to leave some (low overhead) operations to the CPU of the host personal computer or laptop.

The SIMD GP interpreter has two novel sources of overhead. First since it (in principle) runs all programs simultaneously, short programs take as long to execute as long ones. So (to a first approximation) run time is dominated by that of long programs. This depresses average performance. For example, if the biggest tree is 8 terminals and 7 functions but the average tree has only 11 nodes; the expected average performance might be reduced by a factor of 11/15. Secondly to allow GP individual to behave differently, yet be interpreted by a single program simultaneously. The SIMD interpreter dispatches every possible instruction at every point in the tree. See Figure 1. Effectively each GP individual acts as a sieve saying which operation it wants performed next. While multi-ops, conditionals, loops, jumps, subroutines and recursion are possible they are not included in these benchmarks.

The following section discusses some other previous parallel GP systems. The next section 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).

```
const int NP; //Number of programs in population, multiple of GPU_NP
const int LEN =63+1;//Maximum GP individual length, plus 1 stop code
const int GPU_NP = 4*1024*1024/LEN; //22bit GPU limit

Array<1,Value1i>    Train(ntrain*ninputs);
Array<1,Value1bool> Class(ntrain);
Array<1,Value1ub>   PROG(LEN*GPU_NP); //unsigned 8-bit byte
Array<1,Value1i>    prog0 = grid(GPU_NP); //used to simulate indexOf

#define OPCODE ::PROG[PC+(prog0*LEN)]
#define WHICHACID OPCODE-ZeroIndex
#define Acid     Train[(PROTEIN*Value1i(ninputs))+WHICHACID]
#define PUSH(V)  join(join(V,stack(0,1,2)),stack(3,4,5,6))
#define CONST    Value1f(10)*tan(Pie*OPCODE)
#define OP1()    stack = cond(OPCODE<FirstInput,PUSH(CONST),stack)
#define OPacid() stack = cond(OPCODE>=FirstInput && OPCODE<=LastLeaf,PUSH(Acid),stack)
//conditionally POP stack (fake by using rotation)
#define OP3(XCODE,OP) \
     stack(0) = cond(Value1ub(XCODE)==OPCODE,OP,stack(0)); \
        stack = cond(Value1ub(XCODE)==OPCODE,join(stack(0,2,3,4),stack(5,6,7,1)),stack);

rapidmind::Program m_update = RM_BEGIN {
  In<Value1i> prog0;
  Out<Value1f> roc;
  Value<8,float> stack;
  Value1i pos; pos= 0; Value1i neg; neg= 0;
  Value1i TP;  TP = 0; Value1i TN;  TN = 0;
  for(int k=0;k<6;k++) {//two loops (k,J) needed to cycle through 1200 example proteins
    Value1i J = 0;
    FOR(J,J<200,J++) {
      Value1i PROTEIN  = (k*200+J);
      Value1i PC; PC=0;
      FOR(PC,PC<(LEN-1),PC++) {
        OP1(); //constant leaf
        OPacid(); //input leaf
        OP3(OPADD,stack(1)+stack(0));
        OP3(OPSUB,stack(1)-stack(0));
        OP3(OPMUL,stack(1)*stack(0));
        OP3(OPDIV,stack(1)/stack(0));
      } ENDFOR
      pos =cond(  Class[PROTEIN],                              pos+1, pos);
      neg =cond((!Class[PROTEIN]),                             neg+1, neg);
      TP  =cond(  Class[PROTEIN]  &&   stack(0)>=Value1f(0),  TP+1,  TP);
      TN  =cond((!Class[PROTEIN]) && (!stack(0)>=Value1f(0)), TN+1,  TN);
    } ENDFOR J
  }//endfor k
  roc = (Value1f(0.5)*TP)/pos + (Value1f(0.5)*TN)/neg;
} RM_END;
```

Figure 2: C++ RapidMind code for SIMD GP interpreter. The inner PC loop implements the loop shown in Figure 1. On a GPU `cond` returns either its second or third argument conditionally on its first.

```
// Access the internal RapidMind arrays where the data is stored
// Ensure the training data, previously read from file, is copied onto the GPU
int*  input_train = Train.write_data();
bool* input_class = Class.write_data();
memcpy(input_train,alltrain,ntrain*ninputs*sizeof(int));
memcpy(input_class,allclass,ntrain*sizeof(bool));

unsigned char* Pop = new unsigned char[LEN*NP];

float soutput_error[NP]; //fitness (lower is better)

int eval_Pop() {
for(int n=0;n<(NP/GPU_NP);n++) {
  // Ensure the GP population is copied onto the GPU
  unsigned char* input_PROG = PROG.write_data();
  memcpy(input_PROG,&Pop[n*GPU_NP*LEN],LEN*GPU_NP);

  Array<1,Value1f> error = gpu.m_update(prog0); //Run GPU

  const float* result = error.read_data();
  for(int i=0;i<GPU_NP;i++) {
    soutput_error[i+n*GPU_NP] = (isinf(result[i]) || isnan(result[i]))?
                                FLT_MAX : 1-result[i];//AUROC
  }
}//endfor each GPU sized element of Pop
}
```

Figure 3: Evaluating the whole population on the GPU with interpreter (Figure 2). To avoid exceeding the GPU 22-bit limit, the populations is processed in 4 megabyte (`GPU_NP`) units. If the training data is to be changed, `Train.write_data()` etc, must be executed repeatedly to ensure the new data are copied to the GPU.

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) In many cases, calculation of an individuals fitness requires testing its performance on multiple training examples. Mostly each training case could be run independently in parallel. 2) Typically a fitness measure is defined for each member of the population independently. So the fitness of each member of the population could be calculated on independent hardware in parallel. 3) Lastly sometimes experimenters wish to assign statistical confidence to the stochastic element of their results. With GP, and other stochastic search algorithms, this is typically requires multiple independent runs of the GP. Again each run can typically be performed on several workstations or a network of personal computers simultaneously. The, comparative, ease with which EC can exploit parallel architectures has lead to the expression "embarrassingly parallel".

Early work on parallel GP include Ian Turton's use of a Cray super computer by a GP written in Fortran [Turton *et al.*, 1996]. Koza has 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 [Chong and Langdon, 1999] 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 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.

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 (see [Ebner *et al.*, 2005] and the references therein).

Last year [Harding and Banzhaf, 2007, Section 3] detailed the various major high level language tools for programming GPUs (Sh, Brook, PyGPU and Accelerator). In the mean time nVidia has promoted two additional tools: CUDA and Cg (C for graphics) [Fernando and Kilgard, 2003]. CUDA, in particular, is specific to nVidia's GPUs. While "Sh" is still available from SourceForge its development is effectively frozen at Sh 0.8.0 and the first author of [McCool and Du Toit, 2004] 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. RapidMind supports a number of different data types (Boolean, byte, short int, float) and provides automatic translation from these to and the native floating point data type used on the GPU. (And back again.) Another nice feature is RapidMind frees the C++ programmer from the need to learn graphics terminology (e.g. it uses arrays not textures, programs not fragment shaders) and conceals many limitation of the hardware.

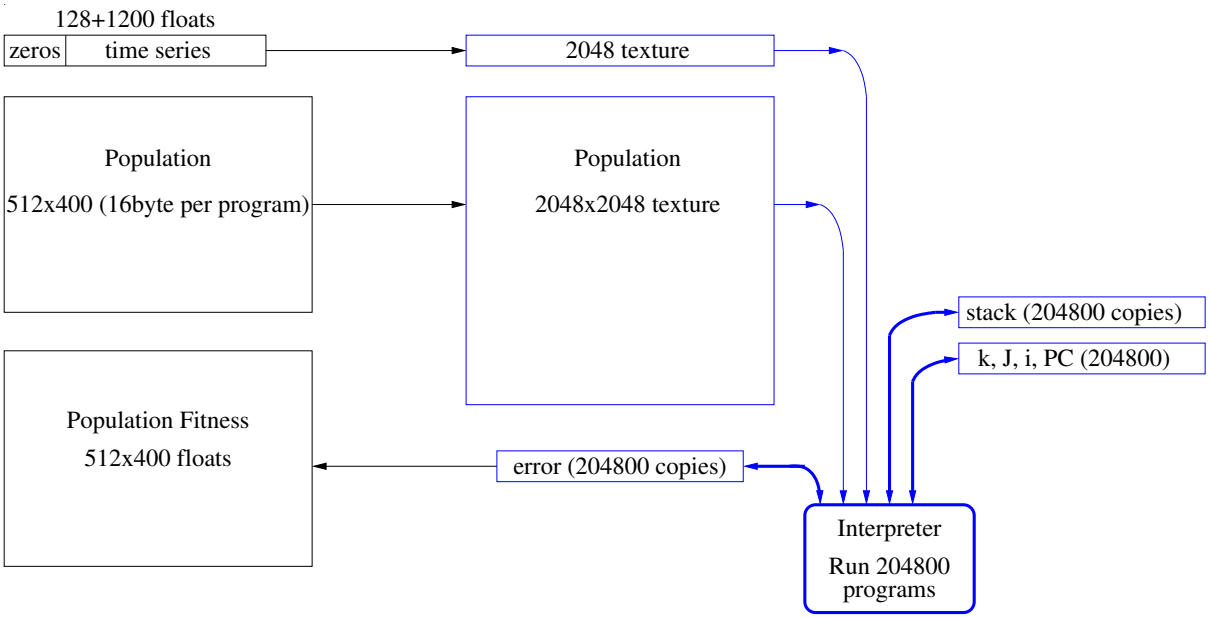

Figure 4: Major data structures for evolving Mackey-Glass. At the start of the run the interpreter is compiled on the CPU (left hand side). It and the training data are loaded onto the GPU (righthand side). Every generation the whole population is transfered to the GPU. Each individual is interpreted using its own stack and local variables and its RMS error is calculated. The error is is used as the programs' fitness. All transfers are made automatically by RapidMind.

4 Mackey-Glass

The Mackey-Glass chaotic time series is described in [Langdon and Banzhaf, 2005b], [Langdon and Banzhaf, 2007] and Table 1. Briefly the GP is presented with 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 upto 128 time steps in the past.

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 5. This population structure both allows a low selection pressure and ready visualisation, cf. Figure 6.

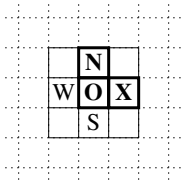

Figure 5: If **N**orth is better than **O**rigin, it is copied over it. But if **O**rigin is is better, **O** is copied over **N**. (No change if equally fit.) After selection, crossover may occur between **O** and **X**. To promote mixing, 50% of crossovers swap roles of the two parents, so a child 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**.

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. 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 (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

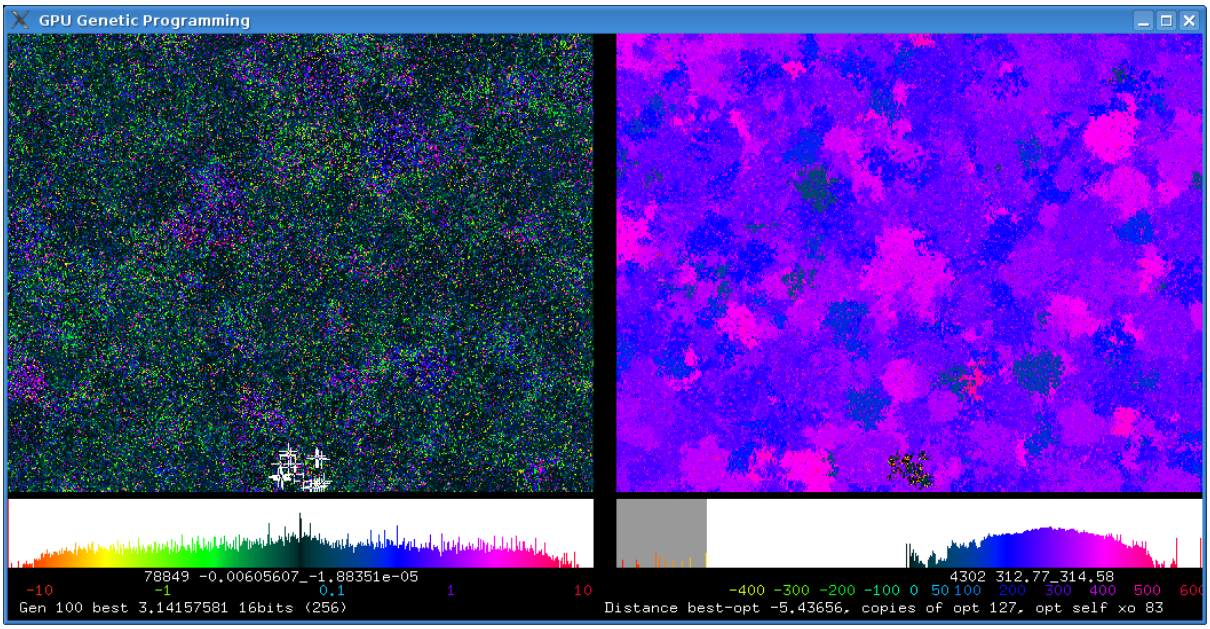

Figure 6: Screen shot of 512×400 GP population evolving under selection, crossover and subtree mutation after 100 generations. Colour hue indicates fitness (left) and syntax (right). Below are two histograms (log scale) showing distribution of population by fitness and genotypic distance from the first optimal solution. The population is just starting to move towards maximal fitness (centre black, highlighted by white cross hairs). Crossover is producing large numbers of unfit leafs (vertical lines at 540 and 600) [Poli *et al.*, 2007]. Nevertheless, particularly when looking at the syntax rather than fitness, local convergence and the production of species, which are promoted by demes, can be readily seen.

Table 2: First row: Best Mackey-Glass prediction error after 50 generations in ten runs. 2nd row: top row multiplied by 128. 3rd Solution size (max 15). Run duration (secs).

										Mean
.036646	.036459	.036646	.036646	.036634	.036646	.036646	.036646	.036646	.036646	.036626
4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69	4.69
9	11	9	9	13	9	9	9	9	9	9.6
167.283	168.062	167.467	167.500	167.337	167.437	167.454	167.466	167.464	167.556	167.464

4.2 Subtree Crossover and Mutation

Koza's [Koza, 1992] crossover was implement for linearised reverse polish notation, except both crossover points are chosen uniformly at random. I.e. 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 either too big or too deep, both are rejected and chosen again. Children produced by crossover are not also mutated prior to fitness evaluation.

If crossover is not used, offspring are produced by mutation. In these experiments, the crossover and mutation rates where chosen so all of the next population are produced by genetic operations. (Ignoring the small chance of crossover or subtree mutation creating children which are genetically identical to their parents.) 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 (0:1) 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 randomly selection as was used in the initial population. Repeated mutations are applied until, the mutated tree is syntactically different from its parent.

In constant 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 solutions that are smaller 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. (To ease comparison the second row of Table 2 gives the RMS error rescaled by the same scaling factor (128) as was used in [Langdon and Banzhaf, 2005a].) 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: GPquick 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) Two more types of mutation (point mutation and "fine tuning constants"). 5) 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, 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. In terms of predictive accuracy on unseen proteins this run produced results significantly better than one technique and the same accuracy but a significantly smaller solution than the other technique [Langdon and Banzhaf, 2007, Table 5].

During this run the average size of the trees was about 56.9 primitives. $1024 \times 1024 \times 1000$ programs were evaluated 200 times each in 6:46:17 (excluding monitoring time). Surprisingly the CPU time has grown from almost nothing to 11%. This suggests an inefficiency which might lead to a small performance improvement. Multiplying by the average program length give us 504 million GP ops per second.

There are several reasons for the decrease. Firstly supporting deeper trees means the interpreter's stack must be bigger. On the GPU this increased the cost of each operation by about 30%. Secondly it is possible that the chunks of work given to the CPU were reduced too much, leading to better interactive response but more data transfer overhead. Also dynamically choosing the training examples may also have reduced performance.

6 Performance of SIMD Interpreter

6.1 How Fast is Interpreter?

The interpreter's performance is summarised in Table 4. The C++ Mackey-Glass GP interpreter was compiled with GCC (version 3.4.6[1], optimisation level -O2) and RapidMind 2.0 (for Fedora Core 5 x86, default optimisation 2). On an nVidia GeForce 8800 GTX, the interpreter was run on 204 800 randomly generated programs of various lengths (mean length 11). (All programs were padded with noops to the fixed 15 statement limit). Each program was run on each of the first 1200 times steps of the IEEE bench-mark series. Averaging over 25 trials this took

[1]RapidMind suggests GCC 4 under Linux.

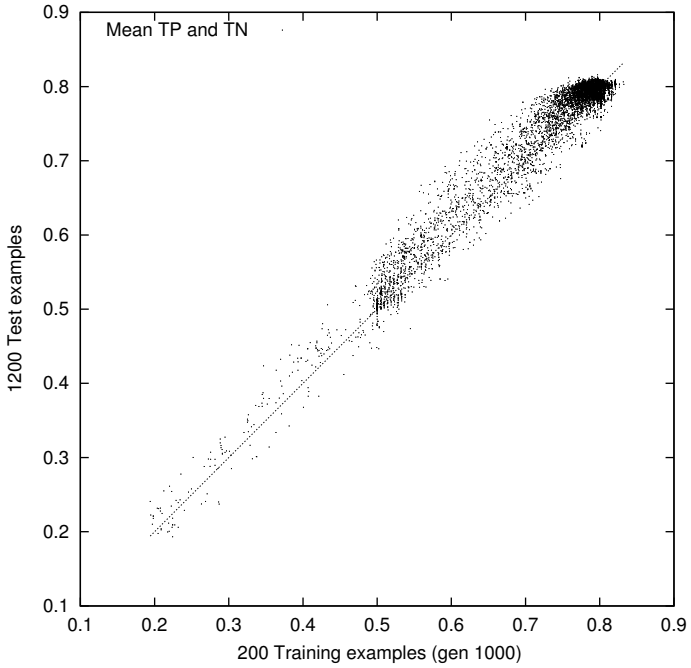

Figure 7: AUROC on 200 randomly chosen training cases in generation 1000 versus AUROC on unseen proteins (First 10 000 of 1 048 576 programs). The strong correlations shows GP has learnt for random samples and (better yet) GP models have avoided over fitting and generalise well.

Table 4: Speed (millions GP operations per second) of SIMD GPU Interpreter

Experiment	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{F}\rvert$	Population	program size	test cases	Speed
Mackey-Glass	8+128	4	204800	11.0	1200	895
Mackey-Glass	8+128	4	204800	13.0	1200	1056
Protein	20+128	4	1048576	56.9	200	504
Laser	3+128	4	18225	55.4	151360	656
Laser$^{9-8}$	9+128	8	5000	49.6	376640	190

3.025 seconds. (We exclude the first execution which incurs an additional one off compilation overhead of about 1.38 seconds and about 0.36 seconds to create the random programs). The GPU interpreter ran $11 \times 204800 \times 1200$ ($2.70336\ 10^9$) GP primitives. An average of 895 million GP operations per second. This excludes a small CPU overhead ($\approx 0.1\%$) associated with loading the program and checking its results.

6.2 Overhead of Opcode Selection

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 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 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.

On an nVidia GeForce 8800 GTX, the interpreter ran 204 800 copies of the same 13 statement program (padded with noops to the fixed 15 statement limit) through the first 1200 time steps

of the IEEE bench-mark series in an average of 3.0258 seconds. (We exclude the first execution which incurs an additional one off compilation overhead of about 1.46 seconds). That is the GPU interpreter ran $13 \times 204800 \times 1200$ ($3.19488\ 10^9$) GP primitives. An average of 1.056 **billion** GP operations per second. This includes a small CPU overhead (less than 1%) associated with loading the program and checking its results.

Next an equivalent interpreter was created specific to the benchmark program which avoided the overhead of dispatching all possible instructions and only used the 13 (plus 2 noops) actually in the benchmark. On average the same $13 \times 204800 \times 1200$ GP primitives took 1.0489 seconds. (Still more than 99% of the elapse time is spent on the GPU.) Thus the SIMD overhead of the GPU GP interpreter is only 2.89 rather than the factor of five anticipated.

One has to be cautious since RapidMind provides an optimising compiler and so observed differences will depend to some extent on how well the optimiser deals with the various cases. Assuming the compilation is equally good in the two cases the overhead of 2.89 compared to the expected 5 can be explained. 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, which is 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 expected. Thus we could expand the function set to include trigonometry operations, log, exponentiation and other mathematical operations which are directly implemented by the GPU and take the same time as our existing functions (such as multiply). While this would not be free, the additional overhead will be proportionately less.

6.3 Expected Speed of Interpreter

The GeForce 8800 GTX GPU card has 128 1.35GHz processors (The nVidia GeForce series no longer makes a distinction between vertex and shader processors. The 128 processors are identical and can be used either as vertex or shader processors.) Each processor is capable of performing a floating point operation (such as multiply) in 4 clock cycles. Thus it has a theoretical rating of $128 \times 1.35/4 = 43.2$GFlop. However this is infeasible since it relies on all the data being immediately accessible all the time. In practice there is a delay on reading data from the GPU's memory.

We estimated each GP function takes about 23 clock cycles and each leaf 42 clock cycles. Therefore each SIMD interpreter loop will take $42 + 23 \times 4 = 134$ processor clock ticks. In contrast an observed rating of 895 million GPops per second corresponds to $895\ 10^6/128 =$ 7 million SIMD loops per second (i.e. one per processor). 1350 MHz/7 $10^6 = 193$ clock cycles for each SIMD loop. This compares well to our weak estimate of 134. This suggests the RapidMind framework is seldom stalling the GPU stream processors for lack of data.

6.4 Comparison with Traditional GP

The Mackey-Glass interpreter (of Section 6.1) was recoded with minimum changes to run in C++ without RapidMind. (The CPU and GPU versions give the same results except for slight differences associated with floating point accuracy). Again using the first 1200 time steps of the Mackey-Glass benchmark the 2211 MHz AMD Athlon 64 Processor 3500+ CPU evolved 50 generations of a population of 204 800 trees in 1129.59 seconds (excluding performance monitoring). (The GPU version took 167.283 seconds). In 50 generation the CPU version evaluated 66 846 484 primitives each 1200 times. (The average program length was 6.4) I.e. an average of 71 million GP ops per second.

There are very few published benchmarks available for tree based genetic programming. However Appendix D [Langdon, 1998] gives figures for Andy Singleton's GPQuick on Gath-

ercole's MAX problem [Langdon and Poli, 1997] for two computers: A 400 MHz Digital Alphastation 500/400 and a SUN SPARCstation 2 fitted with an 80MHz CPU Weitek PowerUP CPU and floating point processor giving a performance of about 1.6 times that of a standard SPARCstation 2. The Alpha gave 1.25M Ops and the Sparc 160K Ops. The MAX problem has only 3 primitives and so GP should be fast. However the MAX problem has a single training case. Table D.2 also provides estimates (by assuming many training cases) of the best speed of GPQuick at 24MOPs for the Alpha and 3.3MOPs for the Sparc2.

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. The program size and number of training examples per fitness evaluation appeared to be a critical factor. 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. In cases where the compiled GP program is run many times, the compiler overhead is spread over many training cases and for large programs, Harding's results mostly show a big performance gain from using the GPU. However if run few times the cost of the compiler and transference 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 smaller than the peak value potentially obtained by compiling but to be more consistent. We obtain a speed up of more than an order of magnitude for very small programs.

Rather than return the results of each programs' execution to the CPU, typically the output is compared with the ideal answer for that fitness case and a partial fitness computed. When all the fitness cases have been processed, the complete fitness (e.g. RMS error) is returned to the CPU. This reduces the volume of data generated by the GPU and returned to the CPU and so may give a modest efficiency gain.

7.1 Implementation Issues

The GPU is designed to operate on graphic images and imposes a limit that arrays be no more than 2k by 2k (i.e. no more than 2^{22} elements). Similarly RapidMind imposes the same limit. Therefore to run megapopulations, we partitioned the population into convenient chunks and executed these. Even though the GeForce 8800 contains ample memory and could store the whole population.

RapidMind imposes a FOR loop limit of 256 iterations. However a given loop can be run many more times by suitable nesting of RapidMind For and C++ for loops. (See column 6 of Table 4.)

Neither FOR loop nor array size limits are automatically checked. If exceeded, the RapidMind code may compile and run but not do what was anticipated. In later versions of the code, C `assert` statements where added to check that critical limits were not inadvertently exceeded.

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. With larger lumps, the interactive response becomes too slow to make it feasible to use the PC for anything else when running a GP. Also if the GPU was active for more than about 16 seconds, the Linux PC became locked and had to be

recovered by rebooting. Conversely if the GPU operates in tiny units, it did not appear to give peak performance.

The complete population is transfered to the GPU and a vector holding the fitness of each individual tree is returned. Therefor the data transfered via the PCs PCI express bus may be several megabytes. While beforehand there was much concern about the cost of data transfers to and from the GPU, our experience seemed to suggest that if the GPU is given reasonably large units of work the cost of transferring the population and the training data into the GPU at the start and fitness vector out at the end are not too onerous.

RapidMind on a unix platform uses the GNU C++ compiler GCC and GDB debugger. These worked but GCC's error reporting can be hard to interpret since RapidMind (like Sh) makes heavy use templates. Our interpreter is written in RapidMind C++ and is automatically compiled and loaded into the GPU by RapidMind once at the start of the GP run. RapidMind uses a cross compiler for each GPU. This worked seamlessly.

7.2 No protected division: Closure

Since a given operation will be performed on many data items simultaneously they are designed not to fail. The GPU could not cope with a case where one division out of 128 failed because it required a divide by zero but the other 127 were ok. All 128 cases must be treated the same. Therefore all operations on GPUs are intended not to fail. Special cases, like divide by zero, are handled by special data values *nan* (not a number) and *inf* (infinity). So our interpreter does not check for divide by zero and does not provide closure [Koza, 1992]. Undoubtedly this makes it faster. In effect, the GPU's floating point hardware supplies closure for us.

We do not get away scott free. Since the GPU can return *nan* and *inf* from a calculation, we need to consider how selection and fitness calculation will deal with them. This is a relatively simple C++ coding issue, c.f. Figure 3. However whatever our solution 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 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 split between the 128 processors and so partly responsible for the GPUs speed. However it does make it difficult to implement a stack, which is the natural way to build a reverse polish notation expression interpreter. Instead it was necessary to simulate a stack using joins, c.f. Figure 2. This works fine for small stacks. Indeed with a stack depth of 4 the interpreter flew at more than a billion GP primitives per second. For the Mackey-Glass and protein prediction experiments the depth was doubled to 8. This 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 all the original GP experiments [Koza, 1992] where conducted in Lisp with a depth limit of 17.

7.4 Non-Tree GP

[Harding and Banzhaf, 2007] has already demonstrated a system based on compiling cartesian GP programs and running the compiled code on GPUs. However where the compilation overhead is too severe, this reverse polish interpreter approach could be readily applied to cartesian GP and also to linear GP. Usually such approaches act directly upon 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 mudelled situations. We could imagine an evolutionary system in which the program did not wait for exactly the next 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. However this form of coevolution is unlikely to yield immediate speed ups on today's problems, but might be a route to meta evolution on more interesting problems in future.

7.5 Possible extensions

Whilst I have stress the use of the interpreter for genetic programming, the idea is general and perhaps could be a convenient way to run programs created by more conventional techniques on under used hardware included in many modern computers.

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, I expect GP crossover could also be performed by the GPU. Whilst [Fok *et al.*, 2007] ran almost all the key steps in an evolutionary computation system (for a bit string GA) on their GPU, the GPU was not independent from the CPU. They still needed synchronised communication between the two. (E.g. they ran their pseudo random number generator on the CPU, even though PRNG can be run on GPUs.) This might be part of the reason why they report some parts of EC running more slowly on the GPU than on the CPU.

With the current state of the art GPU, in practise GP runs are still dominated by fitness calculation. Meaning the remainder of the GP algorithm (selection, mutation, crossover, etc.) cost almost nothing in comparison. Therefore the speed up to be obtained by implementing on the GPU is negligible. However as GPUs continue to improve relative to CPUs, it may not be long before this becomes worthwhile and complete GPU systems will run on each personal computer's ancillary graphics hardware.

8 Conclusions

We have demonstrated execution of traditional tree genetic programming with mega populations actually on the GPU. Parallel operation can yield a speed up of more than 12. Whilst performance varies a little, 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 about £400.

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 readily extended it to other types of GP.

C++ code is available in `ftp://cs.ucl.ac.uk/genetic/gp-code/gpu_gp_1.tar.gz`

Acknowledgements

I would like to thank Simon Harding, Nolan White, Paul Price (MUN) and Joanna Armbruster and Nick Holby of RapidMind. These experiments were performed in the department of Computer Science, Memorial University.

References

[Andre and Koza, 1996] David Andre and John R. Koza. A parallel implementation of genetic programming that achieves super-linear performance. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume III, pages 1163–1174, Sunnyvale, 9-11 August 1996. CSREA.

[Bennett III *et al.*, 1999] Forrest H Bennett III, John R. Koza, James Shipman, and Oscar Stiffelman. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1484–1490, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[Chong and Langdon, 1999] Fuey Sian Chong and W. B. Langdon. Java based distributed genetic programming on the internet. In Banzhaf *et al.*, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1229, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann. Full text in technical report CSRP-99-7.

[Ebner *et al.*, 2005] Marc Ebner, Markus Reinhardt, and Jürgen Albert. Evolution of vertex and pixel shaders. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *LNCS*, pages 261–270, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[Eklund, 2003] Sven E Eklund. Time series forecasting using massively parallel genetic programming. In *Proceedings of Parallel and Distributed Processing International Symposium*, pages 143–147, 22-26 April 2003.

[Fernando and Kilgard, 2003] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. Addison-Wesley, nVidia, 2003.

[Fok *et al.*, 2007] Ka-Ling Fok, Tien-Tsin Wong, and Man-Leung Wong. Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, 22(2):69–78, March-April 2007.

[Harding and Banzhaf, 2007] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on GPUs. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *LNCS*, pages 90–101, Valencia, Spain, 11 - 13 April 2007. Springer.

[Juille and Pollack, 1996] Hugues Juille and Jordan B. Pollack. Massively parallel genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 17, pages 339–358. MIT Press, 1996.

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

[Langdon and Banzhaf, 2005a] William B. Langdon and Wolfgang Banzhaf. Repeated patterns in tree genetic programming. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *LNCS*, pages 190–202, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

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

[Langdon and Banzhaf, 2007] W. B. Langdon and W. Banzhaf. Repeated patterns in genetic programming. *Natural Computation*, 2007? Online.

[Langdon and Barrett, 2004] W. B. Langdon and S. J. Barrett. Genetic programming in data mining for drug discovery. In Ashish Ghosh and Lakhmi C. Jain, editors, *Evolutionary Computing in Data Mining*, volume 163 of *Studies in Fuzziness and Soft Computing*, chapter 10, pages 211–235. Springer, 2004.

[Langdon and Poli, 1997] W. B. Langdon and R. Poli. An analysis of the MAX problem in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 222–230, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Langdon, 1998] William B. Langdon. William B. Langdon. *Genetic Programming and Data Structures*, Kluwer.

[Langdon, 2004] W. B. Langdon. Global distributed evolution of L-systems fractals. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming, Proceedings of EuroGP'2004*, volume 3003 of *LNCS*, pages 349–358, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag.

[McCool and Du Toit, 2004] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. AK Peters, 2004.

[Page *et al.*, 1999] J. Page, R. Poli, and W. B. Langdon. Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 39–49, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[Poli *et al.*, 2007] Riccardo Poli, William B. Langdon, and Stephen Dignum. On the limiting distribution of program sizes in tree-based genetic programming. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *LNCS*, pages 193–204, Valencia, Spain, 11 - 13 April 2007. Springer.

[Turton *et al.*, 1996] Turton, Openshaw, and Diplock. Some geographic applications of genetic programming on the cray T3D supercomputer. In Jesshope and Shafarenko, editors, *UK Parallel'96*. Springer, 1996.