Strategies to Minimise the Total Run Time of Cyclic Graph Based Genetic Programming with GPUs

Track: Parallel Evolutionary Systems

Tony E Lewis
Computer Science and Information Systems
Birkbeck, University of London
London, UK
tony@dcs.bbk.ac.uk

George D Magoulas
Computer Science and Information Systems
Birkbeck, University of London
London, UK
gmagoulas@dcs.bbk.ac.uk

ABSTRACT

In this paper, we describe our work to investigate how much cyclic graph based Genetic Programming (GP) can be accelerated on one machine using currently available mid-range Graphics Processing Units (GPUs).

Cyclic graphs pose different problems for evaluation than do trees and we describe how our CUDA based, "population parallel" evaluator tackles these problems.

Previous similar work has focused on the evaluation alone. Unfortunately large reductions in the evaluation time do not necessarily translate to similar reductions in the total run time because the time spent on other tasks becomes more significant. We show that this problem can be tackled by having the GPU execute in parallel with the Central Processing Unit (CPU) and with memory transfers. We also demonstrate that it is possible to use a second graphics card to further improve the acceleration of one machine.

These additional techniques are able to reduce the total run time of the GPU system by up to 2.83 times. The combined architecture completes a full cyclic GP run 434.61 times faster than the single–core CPU equivalent. This involves evaluating at an average rate of 3.85 billion GP operations per second over the course of the whole run.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*

General Terms

Performance

Keywords

Genetic Programming, Cyclic, Cartesian Genetic Programming, Graph Based, Graphics Card, Graphics Processing Unit, CUDA

1. INTRODUCTION

Evolutionary Computation (EC) techniques are well suited to parallel computation because the task of evaluation is often easily broken up into independent evaluations for different individuals, test cases, population subgroups (or demes) and runs. Tasks which can be divided so easily are often referred to as "embarrasingly parallel" [1].

Genetic Programming (GP), a computationally intensive form of EC, has reaped the benefits of parallel computing through the use of the cluster [1], the Field Programmable Gate Array (FPGA) [13], the Xbox [14] and the Graphics Processing Unit (GPU).

1.1 Genetic Programming on Graphics Processing Units

Of the parallel processors that are currently widely available, perhaps the most interesting for EC researchers is the GPU. Current GPU architecture is essentially Single Instruction, Multiple Data (SIMD) which means that it involves many threads executing the same instructions on different data.

At present, tackling problems using the GPU requires dividing the problem appropriately and writing a suitable GPU program or "kernel" to tackle the sub-problems. In practice this can be an intricate process. However the rewards are significant, with some applications having seen speed improvements of two or more orders of magnitude [6]. Furthermore, the power of the technology appears to have been improving more quickly than for the standard Central Processing Unit (CPU) [2].

GP is particularly well suited to a GPU implementation because it often uses floating point numbers as the basic type in its evaluations and GPUs are particularly effective at floating point computation. Tackling GP with GPU technologies has been referred to as General Purpose Genetic Programming on Graphics Processing Units (GPGPGPU) and a website (`http://www.gpgpgpu.com`) is maintained by Simon Harding.

Previous work has demonstrated the power of the "data parallel" approach [2], [6]. This intuitive method uses the GPU's parallel threads to evaluate the different test cases. A separate GPU kernel is compiled for each individual and this is then used to evaluate the complete data set. For very large data sets, the time taken to compile the kernel is a small fraction of the total evaluation time and remarkable reductions in evaluation time have been observed. For example, a data parallel GPU implementation was 7351.06 times faster than a CPU equivalent at evaluating an expression of length 10000 over 65536 test cases [6].

Unfortunately many GP problems do not have particularly large data sets and so are not as well disposed to data parallel approaches. For these problems, "population parallel" approaches have been used [7], [11] which use the GPU's parallel threads to evaluate the different individuals in the population (and potentially the different test cases too). The advantage of these methods is that they don't require a new kernel to be compiled and launched for each new individual. The difficulty is that each individual may have different behaviour. This is solved by using a GPU interpreter kernel that handles the different individuals. This makes population parallel methods more complicated to implement.

Direct comparisons between the results of different approaches are difficult (as discussed in Section 3.1). However it appears that population parallel approaches are more effective for smaller data sets but are unable to yield speed improvements as impressive as those from data parallel approaches on very large data sets. One population parallel evaluator achieved its best result on a regression problem with 1024 test cases and a population of 2500 individuals; the evaluation stage ran nearly 80 times faster which translated to the full run being over 40 times faster [11]. Another was described reducing run times by around 7 times compared to the CPU and achieving evaluation rates of up to 1056 million GP operations per second [7].

GPUs have provided useful speed improvements for GP but there are significant variations in the susceptibility of different GP systems to these methods. To achieve the best GPU accelerations, researchers might focus their attention on the sorts of GP setups that have previously reaped the best rewards. At present, this means tackling data-rich problems. For example, the evolution of image filters has been tackled with a GP system accelerated using a GPU implementation [5].

It would be useful to know of any other GP systems that will respond well to GPU implementation. Systems that may have previously been prohibitively computationally expensive may find new life if they are highly appropriate for parallel computation. Indeed, intuition might suggest that a computationally intensive system would benefit most from the GPU's computing power for each transfer to and from the graphics card. The candidate system examined in this work is cyclic genetic programming.

1.2 Cyclic Genetic Programming

Cyclic genetic programming is a form of graph-based programming in which individuals are allowed to contain cycles and the evaluation is performed in an iterated flip-flop fashion. The papers that introduced Cartesian Genetic Programming (CGP) [9] and Parallel Distributed Genetic Programming (PDGP) [10] specifically noted these representations' abilities to handle cycles. Neural Programming (NP) [12] is an inherently cyclic representation.

These representations bear some resemblance to neural networks. Research on using these powerful structures in GP could connect with research on the evolution of neural networks and might allow new problems to be tackled. However little research has been done on cyclic graph-based GP because the evaluations are time consuming. The evaluations require the whole individual to be evaluated over multiple iterations. Furthermore the memory requirements are significant, as explained in Section 2.4, which might mean more slow accesses to memory off the processor's cache. In order for cyclic GP to become more widely accepted, it must become faster.

1.3 The Steps of the Investigation

This paper describes work to investigate how much a cyclic graph-based GP run can be accelerated on one machine with the use of GPU technology. This is performed in three incremental steps:

1. Investigate the ability of a population parallel GPU implementation to accelerate cyclic graph evaluation.
2. Investigate accelerating the run further by carrying out GPU execution, CPU execution and data transfer in parallel.
3. Investigate accelerating the run further by using a second GPU and a second CPU core.

Work in this area has often concentrated on the evaluation stage because this typically accounts for the majority of the execution time of a full GP run. However, once the evaluation is accelerated, the time spent on the other parts of the run become more significant. For this reason, very impressive reductions in evaluation time may lead to much less impressive reductions in overall run time.

It is possible to execute GPU kernels in parallel with CPU threads. This parallelism has previously been exploited to further accelerate the evaluation of an EC system by having the CPU and GPU simultaneously evaluating separate demes [4]. Here it is used to reduce the overall run time beyond what is achieved by tackling the evaluation stage alone. This is achieved by allowing the GPU to evaluate one deme at the same time as the CPU is performing other tasks on another.

The argument for this tactic can be elucidated by analogy with project management. The GPU and the CPU are believed to be the critical resources so they should work in parallel to minimise the time that they spend waiting for new work.

In the third step of the investigation a second GPU resource is also added to improve efficiency further.

2. ARCHITECTURE

2.1 Cyclic Cartesian Genetic Programming

This investigation of cyclic genetic programming uses Cartesian Genetic Programming (CGP). It is important to distinguish between these two types of GP as a GP system may be Cartesian, cyclic, both or neither. The acronym CGP will only be used to refer to Cartesian Genetic Programming.

CGP is a graph based form of GP introduced by Miller and Thomson [9]. The standard form of CGP is constrained to be acyclic, however the paper that introduced the representation explicitly stated the possibility of adjusting a parameter to allow cyclic individuals.

CGP was chosen for this work because it has been the subject of numerous papers and tutorials in recent years [5], [8], [9]. Similar representations such as cyclic Parallel Distributed Genetic Programming (PDGP) [10] and Neural Programming (NP) [12] were not tested.

CGP was originally inspired by electronic circuit design and it approached the problem of graph crossover with an elegant mapping from a genotype consisting of a string of integers to a graph based phenotype.

In a CGP phenotype, the nodes are laid out in a two dimensional (Cartesian) grid with the input nodes at the left and the output nodes at the right. The connections are constrained such that nodes' inputs are connected to nodes in the previous l columns where l is an adjustable parameter called "levels back". In a CGP individual, each node input has exactly one connection but nodes' outputs may be connected to no nodes, one node or many nodes. Recent CGP work has tended to use one row and a parameter setting that allows node inputs to be connected to any previous node in the row [5], [8]. This leaves few constraints remaining. To achieve the cyclic graphs for the experiments, further constraints were removed by allowing function nodes' inputs to be connected to the outputs of any other input node or function node (including their own outputs).

The experiments also followed much of the CGP research in not using crossover [5], [8]. Acyclic CGP systems may use very large individuals because few of the nodes typically get connected to the output and most of the nodes can be disregarded during evaluation. Unfortunately cyclic CGP individuals appear to be more prone to use a high proportion of their nodes, so smaller individuals of 30 nodes were used.

The evaluation of a normal CGP individual is much like that of any normal GP individual: each node connected to an input of another node is evaluated before it. However the evaluation of a cyclic individual is trickier because the cycles make it unclear how to order the node evaluations and this ordering can have a big effect on the result. The standard approach to this is to perform multiple iterations. For the first iteration, all of the nodes (except the input nodes) output a value of 0. For each iteration after that, the nodes' results from the previous iteration are used as their outputs. In this way, the order of node evaluation in each iteration does not affect the result. The number of iterations per evaluation was varied as part of the experiments.

The parameters for the CGP runs and the problem tackled are summarised in Table 1.

Objective	Match $x^2 + x + 1$
Test cases	Evenly spaced points from -4 to 4
Number of test cases	Varied in experiments
Function set	+, -, *, % (protected division)
Terminal set	x
Evaluation type	32-bit floating point numbers
Fitness	Sum of absolute errors
Selection	Tournament selection (size 30)
Initialisation	Standard CGP initialisation
Population structure	4 demes of 100 individuals
Deme transfer rate	Once every 30 generations
Deme transfer type	Deme's best replaces a neighbour
Mutation	All genes with probability 0.05
Crossover	None
Termination	100 generations
Inputs per individual	1
Arity of functions	2
Nodes per row	30
Nodes per column	1
CGP levels back	∞
CGP levels forward	∞
CGP self loops allowed	True
Iterations in evaluation	Varied in experiments

Table 1: A table summarising the parameters of the symbolic regression GP runs.

2.2 Technology

The architecture described in this work was built on Compute Unified Device Architecture (CUDA), a software environment provided for free by nVidia to allow general purpose programming of many of their GPUs. CUDA kernels are written in an extended version of C and are compiled with a special compiler. The use of CUDA is expanding rapidly, with the nVidia website already listing many university courses that are teaching the use of the technology. CUDA has previously been used to implement a population parallel GP [11]. CUDA was chosen for this work because it is powerful, relatively simple to use and well supported. CUDA is described as Single Instruction, Multiple Thread (SIMT) which means that the threads are allowed to have divergent behaviour, although they suffer a performance penalty for doing so.

2.3 Code

The CPU evaluator and the GPU evaluator were written as subclasses of a common abstract base class. This architecture made it easy to compare the speeds and cross validate the results. Validation is complicated by the fact that the CPU and GPU have slightly different floating point implementations. Fortunately CUDA has a device emulation mode which attempts to emulate the GPU using CPU threads and this can be used for direct comparison with the CPU evaluator. The authors advocate the approach of using equivalent Evaluator classes because it facilitates adding support for new parallel technologies in the future.

2.4 Memory Requirements

Memory requirements tend to be greater when evaluating cyclic graph-based GP individuals than when evaluating tree-based GP individuals. This is because partial results are not reused in trees and so can be discarded after first use. A full tree of depth d and made up of function nodes with arity a will contain $\frac{a^{d-1}-1}{a-1}$ nodes but a stack based tree evaluator may only require $(d-1)*(a-1)+1$ memory slots to evaluate it. To illustrate the significance of the difference, a full tree with $a = 4$ and $d = 10$ would have 87381 nodes but could be evaluated with only 28 memory slots. Evaluating 87381 nodes in a cyclic graph-based individual might require up to 87381 memory slots.

This large memory requirement makes it more difficult to design a system in which the processor accesses memory as efficiently. It provides a new challenge in designing an appropriate CUDA evaluator. It also means that there may be greater improvements to be had over a standard CPU implementation that faces the same problems.

CUDA arranges its threads into blocks and provides each block with 16384 bytes of fast, shared memory which is ideal for storing the values of nodes during evaluation. Each combination of a node and a test case requires two floating point numbers, or eight bytes, of this memory for the iterated flip-flop evaluation. This means that each block is restricted to evaluating 2048 node–test case combinations at once. Furthermore, each individual must be evaluated within one thread block in order to use this shared memory. These issues affected the design of the CUDA thread layout.

2.5 CUDA Thread Layout

One of the biggest design issues for a GPGPGPU system using CUDA is the division of the individuals, their nodes and the test cases over the CUDA threads. CUDA is most efficient when the divergence of behaviour within each group of 32 neighbouring threads (or warp) is minimal. This has previously been achieved by having the contiguous threads in the warp evaluating the same nodes on contiguous test cases [11]. This approach also allows for efficient memory access and was adopted here.

When there are fewer than 32 test cases, the test cases are padded out so that their number is a power of two in an attempt to minimise unnecessary warp divergence. Similarly, when there are more than 32 test cases, the test cases are padded out so that their number is a multiple of 32 for the same reason. Each warp performs the complete evaluation of each group of 32 test cases before moving on to the next group. As each block can only evaluate a maximum of 2048 node–test case pairs at once as discussed in Section 2.4, particularly large individuals may force some thread blocks to use fewer than 32 test cases per group.

The handling of test cases described above combined with the limit on the number of node–test case pairs means that very few individuals can typically be evaluated per block. A simple strategy might assign a whole program to each thread. However this often leads to a fairly small number of threads and CUDA is most efficient when there are many threads per block. Hence the nodes of each individual are split up into consecutive groups and the CUDA function `__syncthreads()` is used by the kernel to keep the different parts of the evaluation synchronised. Nodes are padded out as required.

Figure 1 shows an example of the division of work in a block of threads. The architecture uses 256 threads per block or fewer to avoid practical problems caused by a CUDA limit of 8192 registers per block. For the experiments in this work, this entailed launching a grid of 400 thread blocks for each deme evaluation.

2.6 Parallel GPU Execution, CPU Execution and Memory Transfer

Figure 2(a) depicts a timeline associated with a standard CPU implementation and Figure 2(b) depicts a timeline for a standard GPU implementation. The extent of the acceleration means that the scale used for the other timelines only allows Figure 2(a) to partially depict the first CPU evaluation which took 1.70 seconds (and the full run which took 59.16 seconds).

Section 1.3 explains that accelerating the evaluation in this way only tackles part of the problem because the other tasks of the run become more significant in the total run time. The same section also explains that synchronous access to the GPU and to the data transfer mechanism only utilises part of the available parallel computing power. The architecture used in this work was designed to use the remaining part of the power to help attack the remaining part of the problem.

This is possible by dividing the population into demes. When one deme is sent for evaluation by the GPU, execution

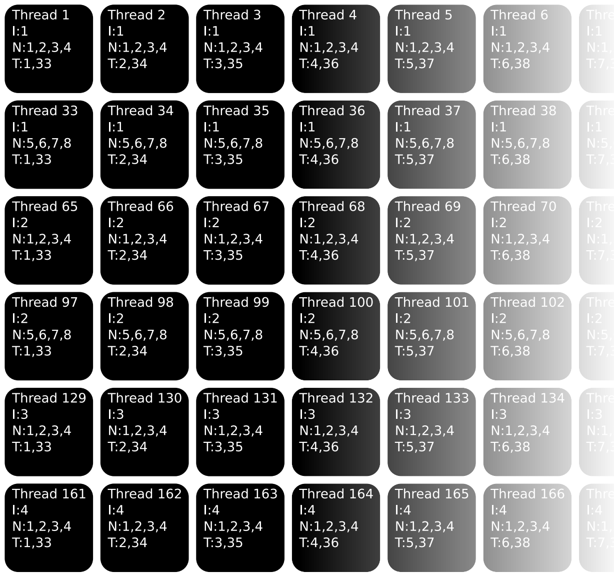

Thread 1 I:1 N:1,2,3,4 T:1,33
Thread 2 I:1 N:1,2,3,4 T:2,34
Thread 3 I:1 N:1,2,3,4 T:3,35
Thread 4 I:1 N:1,2,3,4 T:4,36
Thread 5 I:1 N:1,2,3,4 T:5,37
Thread 6 I:1 N:1,2,3,4 T:6,38
Thread 33 I:1 N:5,6,7,8 T:1,33
Thread 34 I:1 N:5,6,7,8 T:2,34
Thread 35 I:1 N:5,6,7,8 T:3,35
Thread 36 I:1 N:5,6,7,8 T:4,36
Thread 37 I:1 N:5,6,7,8 T:5,37
Thread 38 I:1 N:5,6,7,8 T:6,38
Thread 65 I:2 N:1,2,3,4 T:1,33
Thread 66 I:2 N:1,2,3,4 T:2,34
Thread 67 I:2 N:1,2,3,4 T:3,35
Thread 68 I:2 N:1,2,3,4 T:4,36
Thread 69 I:2 N:1,2,3,4 T:5,37
Thread 70 I:2 N:1,2,3,4 T:6,38
Thread 97 I:2 N:5,6,7,8 T:1,33
Thread 98 I:2 N:5,6,7,8 T:2,34
Thread 99 I:2 N:5,6,7,8 T:3,35
Thread 100 I:2 N:5,6,7,8 T:4,36
Thread 101 I:2 N:5,6,7,8 T:5,37
Thread 102 I:2 N:5,6,7,8 T:6,38
Thread 129 I:3 N:1,2,3,4 T:1,33
Thread 130 I:3 N:1,2,3,4 T:2,34
Thread 131 I:3 N:1,2,3,4 T:3,35
Thread 132 I:3 N:1,2,3,4 T:4,36
Thread 133 I:3 N:1,2,3,4 T:5,37
Thread 134 I:3 N:1,2,3,4 T:6,38
Thread 161 I:4 N:1,2,3,4 T:1,33
Thread 162 I:4 N:1,2,3,4 T:2,34
Thread 163 I:4 N:1,2,3,4 T:3,35
Thread 164 I:4 N:1,2,3,4 T:4,36
Thread 165 I:4 N:1,2,3,4 T:5,37
Thread 166 I:4 N:1,2,3,4 T:6,38

Figure 1: An example of the division of work in a block of 192 CUDA threads. The letter I precedes the individual's number, N precedes the nodes' numbers and T precedes the test cases' numbers. Not all of the nodes are displayed as indicated by the fade to white.

returns to the CPU which can then simultaneously process another deme. This work involves the CPU updating this other deme in preparation for its next evaluation, based on the results of its last evaluation. An example timeline depicting this approach is shown in Figure 2(c). For most generations, the demes do not interact so they can be handled independently. Every 30 generations, this extra parallelism is halted so that evaluation results can be collected for the whole population and used to conduct migration between the demes. The extra parallel procedure is then restarted.

The architecture goes further by also performing data transfers to and from the graphics card in parallel with the GPU computation and the CPU computation. This is achieved by associating each deme with a CUDA stream and then forming the queue of requested GPU data transfers and GPU executions in the appropriate stream. This information can then be used by CUDA to determine which data transfers and kernel executions must be run sequentially and which can be run in parallel. Before the CPU code uses any results from the GPU, it calls a CUDA function to ensure that all preceding operations in the relevant stream are complete. This technique requires the memory on the host side of the transfer to be page locked.

The effects of dividing up the population are not discussed here. The literature contains several studies of this technique and there are indications that it can have significant positive effects on the evolutionary process [3].

2.7 A Second GPU

The final addition to the architecture was a second GPU. The system of demes already described in Section 2.6, makes it simple to divide the work of evaluation between the GPUs. This approach is depicted in Figure 2(d).

The architecture is complicated by the fact that CUDA requires a different thread to access each GPU. There isn't much communication required between the threads because the demes are largely independent and because each thread has its own CUDA resources. However, the use of additional threads does add complication in areas such as the code which handles the occasional deme transfers. The use of a second thread makes the architecture more powerful because it exploits another of the CPU's cores.

3. EXPERIMENTAL SETUP

The aim of the experiments was to investigate the acceleration of the described architecture. It was verified that the GPU evaluator and CPU evaluator produced equivalent results but the experiments were not concerned with the behaviour of the GP runs. The experiments did not use additional optimisations (such as reusing results for identical individuals, removing nodes not connected to the output and terminating the iterated evaluation early on convergence) because these might have distorted the results.

3.1 Assessment of Performance

It is difficult to directly compare between the results from pieces of research in this area and it is not clear what measurement is most useful. An ideal measurement would allow direct comparison between different pieces of research and would give other EC researchers an indication of the potential benefits.

Perhaps the most obvious measurement is the speed improvement over a CPU implementation. Unfortunately different systems will have different configurations in many areas such as the CPU, the GPU, the compiler and the compiler options. A test indicated that simply turning the compiler optimisations off makes the CPU evaluator used here run 3.32 times slower. Worse, each researcher may focus on a different type of EC system and so may have their own CPU implementation with a different level of efficiency. Those researchers who craft the best CPU implementations will then face the harshest comparison.

An alternative comparison might be sought by using the device emulation mode of the GPU technology. However, this may fail to discriminate the quality of the GPU code as any inefficiencies will be present in both parts of the comparison. This mode is also likely to be much less efficient than a good CPU implementation so it may give an artificially positive impression. It was not used here as a test showed the device emulation runs to be 9.60 times slower than the CPU evaluator runs.

Another possibility is to measure the rate of evaluation of GP primitives. This has the advantage of being meaningful without a comparison to the CPU performance. The disadvantage is that it may be unfair on attempts to tackle GP systems that are inherently difficult to run efficiently (on either the CPU or the GPU). The greater memory requirements of cyclic GP discussed in Section 2.4 give reason to suspect that cyclic GP may be just such a system.

A further issue is that it is important to distinguish whether the results are just measured over the period of evaluation or over the whole run. Work in this area often reports results measured over the evaluation to indicate the full scale of the improvement. However only measuring during the evaluation may give an artificially positive impression of the speed improvements to be gained with GPUs.

The second and third steps in this work aim to reduce the overall run time by performing other tasks in parallel with the evaluation, so it is important that measurements over the whole run are included.

CPU	Intel Core2 Quad Q6600, 2.40GHz
Graphics card 1	Inno3D GeForce 8800 GT overclocked
Graphics card 2	EVGA GeForce 8800 GT superclocked
Cores per card	112
Operating system	Ubuntu 8.04.1 (Linux 2.6.24-23-generic)
Compiler	GCC v4.2.4
GCC options	-O3 (optimisation level 3)
CUDA	v2.0 (NVCC v0.2.1221)
nVidia driver	v180.06
Fitness caching	None between generations
Cyclic evaluation	Iterated (no early convergence checks)

Table 2: A table summarising the technical details of the system used for the experiments.

The details of the system setup used in this work are provided in Table 2. Each result was averaged over three runs. The times measured over the whole run did not include all of the time spent creating and destroying resources at the start and end of the run. Figure 2 does not show that the runs used to generate the timelines each had an earlier warm-up generation which was adopted because the first generation is often slower, presumably due to resource allocation.

4. RESULTS

4.1 Step One: Cyclic CGP on a GPU

The aim of the first step was to investigate the ability of a population parallel GPU implementation to accelerate cyclic graph evaluation. Table 3 show the results of runs using the CPU architecture and Table 4 shows the equivalents for the GPU architecture.

They show that the GPU evaluator has achieved remarkable performance improvements over the CPU evaluator. The most impressive acceleration occurred with 512 test cases and 40 iterations: the GPU implementation ran 1259.57 times faster than the CPU implementation over the evaluation. This is the order of magnitude of the best results achieved by data parallel methods [6]. This acceleration translated to running 259.98 times faster over the full run.

With only 30 nodes, 32 test cases, 10 iterations, 100 generations and 100 individuals per deme, the GPU evaluator ran 36.30 times faster than the CPU evaluator. This compares favourably with similar work [7], [11]. The rates of evaluation of GP operations demonstrated by the GPU evaluator also compare favourably with those quoted in similar work [7].

The rates for the CPU evaluator seem rather low despite it being coded as a set of tightly nested loops. This might support the hypothesis (indicated in Section 1.2 and Section 2.4) that the memory requirements of cyclic GP make it hard to implement efficiently without specifically designing the architecture to allow efficient memory access.

4.2 Step Two: GPU and CPU in Parallel

The aim of the second step was to investigate further accelerating the run by carrying out GPU execution, CPU execution and data transfer in parallel.

(a) CPU implementation measured over the full run

Test cases	10 Iterations	40 Iterations	160 Iterations
32	44.33 secs 8.66 mo/s	168.80 secs 9.10 mo/s	719.72 secs 8.54 mo/s
128	166.56 secs 9.22 mo/s	653.76 secs 9.40 mo/s	2787.50 secs 8.82 mo/s
512	655.72 secs 9.37 mo/s	2771.46 secs 8.87 mo/s	11045.90 secs 8.90 mo/s

(b) CPU implementation measured over the evaluation

Test cases	10 Iterations	40 Iterations	160 Iterations
32	40.64 secs 9.45 mo/s	165.04 secs 9.31 mo/s	715.79 secs 8.58 mo/s
128	162.27 secs 9.47 mo/s	649.39 secs 9.46 mo/s	2783.16 secs 8.83 mo/s
512	649.16 secs 9.46 mo/s	2764.59 secs 8.89 mo/s	11039.30 secs 8.90 mo/s

Table 3: The results of using a CPU implementation for different numbers of test cases and iterations. Each result is presented with the duration in seconds and the millions of GP operations evaluated per second (denoted mo/s). Results are taken over the full run in Table (a) and over the evaluation only in Table (b).

(a) GPU implementation measured over the full run

Test cases	10 Iterations	40 Iterations	160 Iterations
32	5.44 secs 70.55 mo/s 8.14x	5.30 secs 289.67 mo/s 31.83x	11.32 secs 542.95 mo/s 63.60x
128	7.74 secs 198.39 mo/s 21.51x	6.46 secs 950.63 mo/s 101.15x	30.61 secs 802.86 mo/s 91.06x
512	16.48 secs 372.90 mo/s 39.80x	10.66 secs 2305.36 mo/s 259.98x	107.15 secs 917.45 mo/s 103.09x

(b) GPU implementation measured over the evaluation

Test cases	10 Iterations	40 Iterations	160 Iterations
32	1.12 secs 342.93 mo/s 36.30x	0.80 secs 1910.28 mo/s 205.26x	6.87 secs 894.53 mo/s 104.21x
128	2.44 secs 628.98 mo/s 66.45x	1.11 secs 5559.51 mo/s 587.62x	25.23 secs 973.92 mo/s 110.29x
512	7.58 secs 811.03 mo/s 85.69x	2.19 secs 11197.00 mo/s 1259.57x	98.50 secs 998.00 mo/s 112.07x

Table 4: The results of using a GPU implementation (step one). The other details are as for Table 3 but each entry now also contains the speedup (denoted x) over the equivalent CPU implementation result.

Test cases	10 Iterations	40 Iterations	160 Iterations
32	5.73 secs 67.07 mo/s 7.74x	5.63 secs 272.60 mo/s 29.96x	6.83 secs 899.15 mo/s 105.33x
128	6.48 secs 236.87 mo/s 25.69x	6.30 secs 974.79 mo/s 103.72x	25.04 secs 981.58 mo/s 111.33x
512	9.89 secs 621.45 mo/s 66.32x	9.85 secs 2494.96 mo/s 281.36x	97.90 secs 1004.15 mo/s 112.83x

Table 5: The results of using a parallel GPU–CPU implementation (step two) measured over the full run. The other details are as for Table 4.

Test cases	10 Iterations	40 Iterations	160 Iterations
32	3.95 secs 97.27 mo/s 11.23x	3.87 secs 396.88 mo/s 43.61x	3.99 secs 1538.68 mo/s 180.25x
128	4.38 secs 350.76 mo/s 38.04x	4.44 secs 1383.48 mo/s 147.21x	12.77 secs 1924.36 mo/s 218.27x
512	6.51 secs 944.08 mo/s 100.76x	6.38 secs 3853.97 mo/s 434.61x	49.80 secs 1974.05 mo/s 221.81x

Table 6: The results of using a two GPU, two CPU core implementation (step three) measured over the full run. The other details are as for Table 4.

The results in Tables 3 and 4 show that excellent evaluator results become less impressive when measured over the full run, which is what really matters to EC practitioners. Table 5 shows the results for the parallel GPU–CPU architecture over the full run and compares them to the equivalent CPU results. The best improvements over the results in step one occurred with 512 test cases and 10 iterations: the total run time was reduced by a further 1.67 times.

Figure 2(c) illustrates how this increased parallelism works using times recorded from a real run. The figure shows that data transfer is a very small factor in the total run time. This is not surprising as cyclic GP performs multiple iterations on each individual that is transferred.

The best results are expected to occur when the CPU and GPU take similar amounts of time on each deme. Through optimisation of whichever step is taking the longest, it should then be possible to almost halve the run time.

It may appear inconsistent that some of the evaluation rate results in Table 5 are better than their equivalents in Table 4(b). The explanation for this relies upon the fact that in order to make the evaluation comparisons fair, the GPU evaluation timings (such as those in Table 4(b)) include time taken by the CPU on GPU related tasks. When using the CPU and GPU in parallel, this extra CPU code can be run in parallel with kernels.

4.3 Step Three: Two GPUs and Two CPU Cores

The aim of the third step was to investigate accelerating the run further by using a second GPU and a second CPU core. Figure 2(d) illustrates how this increased parallelism works using times recorded from a real run. The results of this change are shown in Table 6 along with comparisons to the equivalent CPU results.

The best improvement over the results in step two occurred with 512 test cases and 160 iterations: the total run time was reduced by a further 1.97 times. The best improvement over the results in step one occurred with 32 test cases and 160 iterations: the total run time was reduced by a further 2.83 times. Perhaps most importantly, the best improvement over the CPU results occurred with 512 test cases and 40 iterations: the total run time was reduced by 434.61 times and the evaluation rate over the full run was 3853.97 million GP operations per second. This compares extremely well with similar work [7], [11].

It should be noted that the results for this step are for an architecture that uses two CPU cores but the CPU results are for an architecture with only one. Hence, the comparison should be seen as illustrating how powerful the whole architecture can make one machine rather than how much better GPUs are than CPUs (for which this comparison might be considered unfair).

5. CONCLUSIONS AND FUTURE WORK

The most successful previous attempts to accelerate GP with a GPU have used a data parallel approach on very large data sets but they have proved less effective on more modest configurations.

This work proposed cyclic genetic programming as another candidate system for acceleration. A cyclic CGP system was accelerated using a population parallel evaluator. On a realistic configuration (30 nodes per individual, 400 individuals, 512 test cases and 40 iterations per evaluation) the GPU architecture ran 1259.57 times faster than the single–core CPU equivalent when measured over the evaluation. This translated to being 259.98 times faster when measured over the whole run.

The total run time was reduced further by carrying out the GPU execution, the CPU execution and the data transfer in parallel and by using a second GPU and a second CPU core. This lead to an architecture that completed the full run 434.61 times faster than the single–core CPU equivalent and that evaluated 3853.97 million GP operations per second over the full run.

In some circumstances, these additional techniques were able to reduce the total run time of the GPU system by up to 2.83 times.

Execution time has previously been a major obstacle to research on cyclic GP but this work demonstrated that this no longer need be the case.

The accelerations demonstrated in this work go further than that which can be explained by the increased number of cores alone. Three other factors are likely to have played a significant part in this result. Firstly, the GPU is particularly powerful at manipulating floating point numbers, making each core very efficient at this sort of task. Secondly, the GPU is highly optimised for parallel processing and care was taken when designing the thread layout to follow the guidelines on how to allow these optimisations to be most effective. Thirdly, as described in Section 2.4, cyclic GP has extra memory constraints which are likely to have caused access inefficiencies in the CPU implementation but which should have been tackled more effectively in the GPU implementation by exploiting the various types of on-chip memory. It should be noted that these suggestions are only speculation and it is extremely difficult to be certain of the precise contributions of these factors.

Motherboards that accept three graphics cards and graphics cards with multiple GPUs are both now widely available. Limits on resources available for this work meant that it was not possible to use more than two GPUs. The CPU code has been observed running slower when running in multiple threads and investigations suggest that this is not accounted for by mutex locking overheads.

The evaluator can also process directed acyclic graphs and trees but is not expected to be as efficient as GPU evaluators specifically written to handle those structures. This is not a priority as cyclic graphs require much more computational resources.

The architecture described cannot handle individuals of more than 2048 nodes (and in practice might begin to encounter problems at approximately 1500 nodes due to other use of the memory). This can be alleviated with code that strips out any nodes which cannot affect the output.

It is planned that this architecture will be used to facilitate the study of cyclic genetic programming. It is expected that as parallel computing develops, other evaluators for different technologies may be added to the current repertoire. Perhaps this will become a common practice amongst EC researchers.

6. ACKNOWLEDGEMENTS

The authors wish to thank the EPSRC for funding this research through a DTA award and the staff at Birkbeck for their help and support.

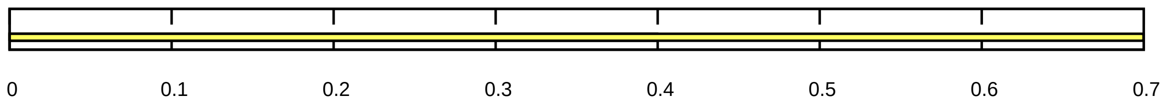
7. REFERENCES

[1] D. Andre and J. R. Koza. A parallel implementation of genetic programming that achieves super-linear performance. *Information Sciences*, 106(3-4):201–218, 1998.

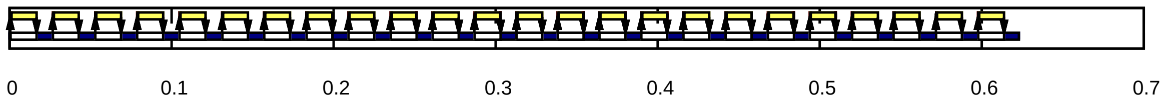
[2] D. M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In D. Thierens, H.-G. Beyer, J. Bongard, J. Branke, J. A. Clark, D. Cliff, C. B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K. O. Stanley, T. Stutzle, R. A. Watson, and I. Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1566–1573, London, 7-11 July 2007. ACM Press.

[3] F. Fernandez, M. Tomassini, and L. Vanneschi. An empirical study of multipopulation genetic programming. *Genetic Programming and Evolvable Machines*, 4(1):21–51, Mar. 2003.

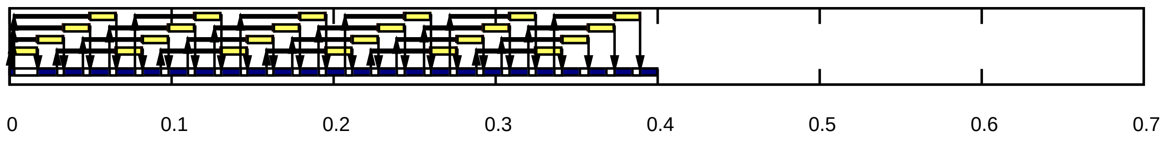
[4] O. Garnica, J. L. Risco-Martin, J. Hidalgo, and J. Lanchares. Speeding-up resolution of deceptive problems on a parallel gpu-cpu architecture. In *Parallel Architectures and Bioinspired Algorithms (at PACT)*, 2008.



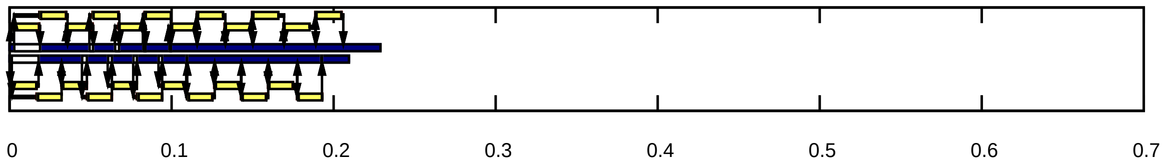
(a) Timeline for standard CPU implementation.



(b) Timeline for GPU accelerated implementation (step one).



(c) Timeline for GPU–CPU parallel implementation (step two).



(d) Timeline for GPU–CPU implementation using two GPUs and two CPU cores (step three).

Figure 2: Timelines (with time in seconds on the x-axis) for different implementations (6 generations, 32 test cases, 160 iterations). Light yellow bars mean evaluation, dark blue bars mean preparation, white sections between bars mean idle time, thin black bars indicate a deme waiting to be evaluated or gathered and arrows mean submission or retrieval.

[5] S. Harding. Evolution of image filters on graphics processor units using cartesian genetic programming. In J. Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[6] S. Harding and W. Banzhaf. Fast genetic programming and artificial developmental systems on gpus. In *HPCS '07: Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[7] W. B. Langdon and W. Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In *EuroGP*, LNCS, Naples, 26-28 Mar. 2008. Springer. Forthcoming.

[8] J. Miller. What bloat? cartesian genetic programming on boolean problems. In E. D. Goodman, editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 295–302, San Francisco, California, USA, 9-11 July 2001.

[9] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[10] R. Poli. Parallel distributed genetic programming. Technical Report CSRP-96-15, School of Computer Science, University of Birmingham, B15 2TT, UK, Sept. 1996.

[11] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Population parallel GP on the G80 GPU. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 98–109, Naples, 26-28 Mar. 2008. Springer.

[12] A. Teller. *Algorithm Evolution with Internal Reinforcement for Signal Understanding*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 5 Dec. 1998.

[13] Z. Vasicek and L. Sekanina. Hardware accelerators for cartesian genetic programming. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 230–241, Naples, 26-28 Mar. 2008. Springer.

[14] G. Wilson and W. Banzhaf. Linear genetic programming GPGPU on microsoft's xbox 360. In J. Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.