# Implementing Cartesian Genetic Programming Classifiers on Graphics Processing Units using GPU.NET

Simon Harding
IDSIA, Switzerland and
Memorial University, Canada
slh@evolutioninmaterio.com

Wolfgang Banzhaf
Department Of Computer Science
Memorial University
Newfoundland, Canada
banzhaf@cs.mun.ca

## ABSTRACT

This paper investigates the use of a new Graphics Processing Unit (GPU) programming tool called 'GPU.NET' for implementing a Genetic Programming fitness evaluator. We find that the tool is able to help write software that accelerates fitness evaluation. For the first time, Cartesian Genetic Programming (CGP) was used with a GPU-based interpreter. With its code reuse and compact representation, implementing CGP efficiently on the GPU required several innovations. Further, we tested the system on a very large data set, and showed that CGP is also suitable for use as a classifier.

## Categories and Subject Descriptors

I.2.2 [**ARTIFICIAL INTELLIGENCE**]: Automatic Programming; D.1.2 [**Software**]: Automatic Programming

## General Terms

Algorithms

## Keywords

Genetic programming, GPU, Parallel Processing

## 1. INTRODUCTION

Graphics Processor Units (GPUs) are parallel computing devices that can have many hundreds of processors. Each processor can execute a small program, called kernels or threads. The GPU can run many threads in parallel, and typically GPU programs are implemented to work with many thousands of such thread. GPUs make an obvious platform for implementing Genetic Programming(GP) as each thread can work on executing an individual or evaluating a fitness case. Evaluating multiple individuals in parallel, or multiple fitness cases in parallel can produce excellent speed-ups over CPU versions.

GPU programming requires specialist programming skills. Although the development tools have been considerable improved in recent years, they are still difficult to work with. There are many different ways to program GPUs. NVidia's CUDA, ATI's APP and OpenCL are C-language extensions that are designed for general purpose programming of GPUs. Another option for programming GPUs involve is writing shader kernels (using OpenGL or DirectX), however these are targeted for graphics programming and the language semantics make implementing 'general purpose programs' trickier. Tools such as MS Accelerator can generate, or use, shader programs - but have the complexity abstracted away from the developer. CUDA is probably the most common GPU development system currently in use.

A popular way of using CUDA is to call the kernels from a higher level language, for example from Python (using PyCuda), *C#* (e.g. using Cuda.Net) or Java (e.g. using JCuda). Although this approach simplifies many parts of the development (for example the parts of the program that do not run on the GPU), it introduces the complexities of calling CUDA kernels from inside the host language. These tools also remove the ability to easily debug the GPU code.

TidePowerd's GPU.NET is a new commercial, closed-source tool for programming GPUs. Like cuda.net and MS Accelerator, GPU.NET is designed to work with Microsoft's .Net Common Language Runtime (CLR). It should also work with Mono. GPU.NET's main feature is that converts Intermediate Language (IL) in a compiled .Net assembly to device code (e.g. PTX instructions for NVidia graphics cards). The supplied tool rewrites the assembly converting flagged methods in the IL into kernels that can run on a GPU, it also automatically adds in new functionality to handle transferring data and launching the kernels. Kernels can be written in a managed language, such as *C♯*, and use the same threading and memory model as CUDA. An example kernel is shown in Listing 1.

Writing GPU code in this way has some benefits, and also some drawbacks. Currently there are no supplied GPU debugging tools. However, it is very easy to execute the code on the CPU, and use the CPU debugging tools. Further there is difficulty in determining how the code has been translated, and what optimizations have (or can) be made. The converter tool seems very fast, and only has to be ran once on compiled code. It should be noted that the generated PTX is further compiled by the device driver before execution. Compared to a pure-native application, there is a small overhead in using .Net to call unmanaged code (i.e. the functions in the device driver). Each function call to native code has an overhead of a few microseconds.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*GECCO'11,* July 12–16, 2011, Dublin, Ireland.
Copyright 2011 ACM 978-1-4503-0690-4/11/07 ...$10.00.

**Listing 1: Example showing the kernel code to element-wise add two vectors of numbers together.**

```
    [Kernel]
    private static void Add(float[] input1, float[] input2, float[] result) {
        int ThreadId =  BlockDimension.X * BlockIndex.X + ThreadIndex.X;
        int TotalThreads = BlockDimension.X * GridDimension.X;

        for (int i = ThreadId; i < input1.Length; i += TotalThreads)
            result[i] =  input1[i] + input2[i];
    }

    public static void AddGPU(float[] input1, float[] input2, float[] result) {
        Launcher.SetBlockSize(128);
        Launcher.SetGridSize(input1.Length / (128*128));
        Add(input1, input2, result);
    }
}
```

## 2. GP ON GPUS

There are two methodologies for implementing GP on GPUs. One is to compile native programs for executions on the GPU, where candidate individuals are converted to some form of source code and then compiled before fitness testing. In an early example, individuals are emitted as Cg shader code, compiled and then loaded into the GPU for execution [1]. Using MS Research Accelerator, the toolkit generated and compiles a DirectX shader program transparently. It then executes this program. In [6], CUDA C programs were generated from the GP individual, compiled to a GPU 'cubin' PTX library and then executed on the GPU. With CUDA, this PTX code needs further just-in-time compiling by the graphics driver.

This process of pre-compilation leads to a significant time overhead, and in general means that this approach is most suited where there is a large amount of data to be processed and the evolved programs are sufficiently complicated.

The other methodology involves writing an interpreter that runs on the GPU as a kernel, which executes a set of operations on the data. The interpreter runs the same evolved program over every element in the dataset, and does so in parallel. Typically in the interpreters, each thread only considers one fitness case.

The benefit here is that the GPU program only needs to be compiled (and loaded) once, and that the only thing that changes are the data that represents the programs to be tested for fitness. The interpreter approach has been used with CUDA [7, 10] and DirectX shaders [11]. Without the pre-compilation overhead, the interpreters work well for smaller amounts of data and for shorter programs.

Interpreters introduce their own performance issues, as there is now the continuous overhead of parsing the evolved program. Fortunately on GPUs, the branching that is typically required in an interpreter is handled very efficiently by GPUs. In this work here, we use an interpreter based approach.

## 3. CGP ON THE GPU

Previously CGP has been used on the GPU with implementations using CUDA and MS Accelerator [3, 5]. These implementations have used the pre-compilation approach, which therefore suffer from the drawbacks of this approach.

In this work here, we present the first interpreter based approach for CGP.

With a GP tree, the interpreter for the tree can be a very straightforward implementation. GP trees are typically binary trees, and there is no obvious redundancy in the encoding. This means that interpreters can use read a reverse-polish version of the tree and use only two registers to store intermediate results. In CGP, programs are encoded in a partly connected, feed forward graph. The genotype encodes this graph, and within each node in the graph there are genes that represent the node function and and connections to either other nodes. A full description of 'classical' CGP can be found in [9]. However, the version used here is more similar to that described in [4], just without the developmental features enabled.

The representation for CGP introduces a number of features that mean the implementation is slightly more complicated. In CGP nodes in the graph are often reused, therefore the result for that computation (and all proceeding computations in that node's subtree) can be cached to avoid re-computing their value. With the normal CPU implementations, this is not a problem as there is typically a lot of working memory available and the performance penalties for what memory is used are hidden by the compiler. With a CUDA GPU, there is a multi-layered memory hierarchy that can be extremely limited, depending on the 'compute capabilities' of the hardware.

For versions 1.0 and 1.1, each multiprocessor has only 8,000 32bit registers. In version 1.2 this was increased to 16,000 and in version 2 (the current latest) this is 32,000. The registers are the fastest memory to work with on the GPU. The next level up in the hierarchy is the shared memory. This is also fast memory, but again is very limited with 16KB of storage on 1.x compute capabilities, and 48KB on 2.x compatible hardware. The top level in the hierarchy is the global memory that can be very large, but can have significant performance penalties in use, as there may be conflicts and caching issues.

With the CGP interpreter, we have to be very aware of how these memory constraints will impact the node-reuse.

CGP genotypes typically include a lot of redundancy in the form of neutral, unconnected nodes. In a good CPU implementation, the nodes in the graph will be analyzed before execution to determine which nodes are involved in the computation. This means that not all nodes in the graph need to be executed. In a GP tree, all nodes are active in the computation (unless there are simplifications that can be determined before execution). CGP genotypes are fixed length, so the maximum number of program operations is limited, further the length of the program stored in the genotype does not bloat over time [8], which has implications for perceived efficiency that we observe later in this paper.

Further, CGP also allows for multiple program outputs to be considered. This also impacts the implementation, as each of these outputs needs to be held in memory.

The function set used here consists of the mathematical operations $+, -, \div, \times$, log, exp, $\sqrt{()}$ and sin(). This version of CGP also requires special functions INP (to return the next available input), INPP (to return the previous input), SKIPP (to move the input pointer, and return that input) and OUTPUT (which flags that node to be used as an output for the program). These special operators are described in more detail in [4].

For CGP, the genotype size was set to 100,000 nodes wide, by 1 node high. Each gene has an equality probability (0.01) of being mutated. A 1+5 evolutionary strategy was used.

# 4. IMPLEMENTATION

To test using GPU.NET, three different interpreter strategies were considered. All three strategies were implemented to run the same format instruction code (IC).

## 4.1 Generating the instruction code

Generating the IC from CGP requires a number of steps, which are all performed on the CPU side. In the first step, the genotype is analyzed to determine which nodes are to be used as outputs. This is done by reading through the nodes to find special 'OUTPUT' functions. (More information on this approach can be found in [4]. Once the output nodes are determined, the active computational nodes can be found. Essentially this is done by recursing backwards through the connections from each of the outputs, but in reality this can be most efficiently implemented in a linear way. At this time the number of times a node is reused is also counted.

From this the IC itself can be generated. In this interpreter model, each line in the IC contains a function, the addresses of two source registers and the address of a destination register. The interpreter here allows for 10 working registers. When the interpreter runs a program it performs the function on the values stored in the source registers, and stores the output value in the destination register.

Hence the next step in the process it to work forwards through the active nodes to write out each step of the IC. For each node, a register is selected (from a stack of currently available registers) to store the output result in. This register address is then stored in a dictionary alongside the node index. The source registers for this node are computed using previous entries in this dictionary.

There are two circumstances in which there will be no source register in the dictionary: nodes that return inputs and nodes that have connections whose relative address is beyond the edges of the graph. For functions that return a program input, the interpreter has a special instruction that can load (i.e. copy) a value from the input dataset to a destination register. Nodes that do not connect to other nodes within the graph will get that input set to a value of 0.

Using the counter stored in the initial parsing and by keeping count of how many times the register was read from, the converter knows when a value in a register will no longer be used. When this happens that memory location is released to be used by further operations. To release the address, the address is just pushed back into the stack of available registers.

In the work here, we consider only one output per program. This value is copied into shared memory so that it can be returned from the GPU. In future work, this can easily be expanded to copy the output to shared memory whenever an output node is computed.

## 4.2 The interpreters

Three different interpreters were written.

The first places the interpreter loop on the host (CPU) side, with the program instructions implemented as linear-algebra style vector operations. In this setup, the interpreter's loop runs on the host, and calls a GPU function to operate on a vector of data. For example, the ADD operation will element wise add two vectors together, and store the results in a third vector. Each vector represents one column of data in the dataset, with the data being aligned so that all elements at a given index are from the same row. Having the interpreter loop on the CPU removes the need for the GPU to implement a branching structure. In future systems, it would also simplify the implementation of operations that operate across rows (such as 'min' or 'max'). The implementation of these vector operations is also convenient as the memory access does not run into any bottlenecks, as the memory access patterns are optimal for the GPU. The interpreter uses the register pattern described above to hold intermediate results, with each register being a vector in global memory. As the length of these vectors is the same as the number of rows in the data, there may be issues with memory. The GPU needs enough global memory to hold the dataset, as well as the output vectors and 10 vectors to use as working registers (see Section 4.3).

Tidepowerd does not allow for the creation of arrays that are local to a kernel. So the next interpreter method uses the fast shared memory to store the working register values. However, as mentioned previously, there is a lack of available shared memory on the GPU. This means that the number of concurrent threads (i.e. the size of the thread block) has to be relatively small. On the compute capability 1.3 devices used here, the shared memory is 16KB. With 10 working registers using 4 bytes per floating point number, there is (theoretically) a maximum of 409 threads per block. However, we used a thread block size of 384, which is the next smallest power-of-2. This, in principle should allow for full occupancy of the GPU.

As the number of threads per block is very limited, and fewer than the number of processors available, there may be risk of under-utilizing the GPU. In the third implementation, the registers were moved from the fast shared memory to slower global memory. This then meant that more threads could execute per block - and hence more processors could be used in parallel. However, now the register memory is a slower resource.

In previous work using interpreters, each thread dealt with one fitness case at a time. In this work, we also investigated the possibility of testing multiple fitness cases per thread.

## 4.3 Sizing for interpreter parameters

To determine the number of registers needed and the maximum amount of storage we needed for the interpreter, we analyzed the requirements of randomly generated genotypes. Although this does not provide the true requirements of evolved programs (which can grow within the bounds of the genotype), it gives an indication of the requirements.

Table 1 shows the behaviour of the program length and maximum number of working registers required for various program lengths. For the working registers (the more

constrained parameter) we found that 8.5 registers should be sufficient 95% of the time. For convience, we chose to use 10 registers (which covers 97% of the randomly generated programs). The maximum program length that can be used with GPU.NET is more flexible. The results indicated that 50 operations should be sufficient, but as this resource is not as limited we use a maximum length of 200 operations.

| | Program length | | Peak Registers used | |
|---|---|---|---|---|
| Genotype width | Avg. | Std. Dev. | Avg. | Std. Dev. |
| 16 | 3.6 | 2.4 | 2.6 | 0.8 |
| 32 | 4.6 | 3.3 | 2.8 | 1.0 |
| 64 | 5.2 | 4.2 | 3.0 | 1.2 |
| 128 | 6.0 | 5.4 | 3.1 | 1.4 |
| 256 | 6.6 | 6.3 | 3.2 | 1.6 |
| 512 | 8.0 | 8.3 | 3.5 | 2.0 |
| 1,024 | 9.0 | 9.2 | 3.6 | 2.0 |
| 2,048 | 9.2 | 10.1 | 3.6 | 2.1 |
| 4,096 | 10.5 | 12.4 | 3.8 | 2.5 |
| 8,192 | 12.4 | 14.9 | 4.0 | 2.7 |
| 16,384 | 13.6 | 16.6 | 4.2 | 3.0 |
| 32,768 | 14.8 | 18.4 | 4.3 | 3.3 |
| 65,536 | 17.3 | 21.8 | 4.7 | 3.8 |
| 131,072 | 16.8 | 23.2 | 4.5 | 3.8 |
| 262,144 | 19.7 | 25.3 | 5.0 | 4.3 |

**Table 1: For the interpreter, program length and the number of working registers required are important parameters. These results show how they vary depending on the size of the genotype.**

# 5. FITNESS FUNCTION

For benchmarking, two fitness functions were implemented. Both are based on the KDD Cup Challenge 1999 data [2]. One fitness function was a typical classification problem where, the fitness of an individual was a measure of the accuracy in detecting normal or abnormal network traffic (in the original problem the type of abnormal traffic is the classifier output). The fitness itself is also calculated using a kernel on the GPU. To calculate the score, 512 threads were launched (each to operate on 1/512th the data). Each thread counts the number of True/False Positive/Negatives within a section of all the program outputs, which are then combined host-side to find a confusion matrix. From this the sensitivity-specificity was calculated.

For the second fitness function, the fitness was the number of instructions in a program. Informal testing, and previous experience with GPUs, showed that longer programs are more efficient - and a higher speed up can be achieved. Where programs are generated directly from trees, linear genetic programming, etc., then all operations in the genotype are executed. In CGP, because of the neutrality, this is not the case. With the bloat-free evolution of CGP, programs we would also expect programs to be more compact. Therefore this second fitness function gives a better idea of the maximum capabilities of the system. The first fitness function however gives a more realistic impression of how CGP on GPU behaves with a typical problem.

# 6. GPU RESULTS

As with previous papers on GP on GPUs, we measure the speed by counting the number of Genetic Programming Operations Per Second (GPOps/s) that can be performed. This measure includes all the overheads (e.g. transferring data, programs, etc. as well as executing the interpreter), and so it is expected to be significantly less than the theoretical Floating Point Operations Per Second (FLOPS) that is often quoted when measuring GPU speed. Unless stated otherwise here, the GPOps/s are reported just for program interpreter stage and not the fitness evaluation stage. It should also be noted that MGPOps/s and GGPOps/s are used to indicate mega-GPOps/s and giga-GPOps/s respectively. To measure the speed, multiple evolutionary runs were performed, with each individual evaluation benchmarked. These were then re-sampled by picking a number of evaluation results at random.

The computer has an AMD 9950 (2.6Ghz) processor, running Windows 7, with a Tesla C1060 (240 CUDA cores, 4Gb, driver version 258.96).

## 6.1 Vector

We first present the results for the 'vector' approach. Listing 1 shows an example $C\sharp$ kernel for adding two vectors together. TidePowerd's converter requires the kernels be private-static written inside an internal-static class. The kernel code itself is also flagged with the [Kernel] attribute. A public-static method (which runs host side) is then used to call this method. When the converter tool is used, it rewrites the class file inside the assembly to add in code so that the host side method can call the GPU kernel. The methods flagged with [Kernel] have their IL converted to device code to run on the GPU. As the listing shows, the kernel code is very similar to CUDA. However, the language conventions are all $C\sharp$.

In the interpreter, the main loop iterates over the instructions, and calls methods (such as the Add function) on the vector data. Since this methodology is also most suited for CPU execution, we report the timing information for a CPU version to provide a comparison. The CPU version uses only one core. Although the data type is float, .Net only provides double-precision versions of the non-primitive mathematical operators.

For the CPU version, we found that the GPOps/s were independent of the length of the program and (largely) of the number of elements in the vector. On average, the CPU version was capable of 64 MGPOps/s (standard deviation 13.2, 1,000 results) with a minimum 18.3 and maximum of 139 MGPOps/s.

We were unable to successfully run the GPU version on the complete dataset as there was insufficient memory to hold the input data and the working registers (10 registers, each the size of number of rows of the input data). Therefore, these benchmarks are from using 2,000,000 rows (half the data). GPU.NET is able to work with multiple GPUs, so it would be possible to implement the software to span the data over multiple devices.

On the reduced dataset, the GPU vector interpreter was able to perform am average of 144 MGPOps/s (std. dev. 12.8, 1,000 results sampled), and a minimum of 64 and a maximum of 199 MGPOps/s. The speed again was independent of the length of the evolved program.

## 6.2 Global and Shared Memory

Listing 2 shows a section of the kernel source code for interpreting an evolved program on the GPU. The arrays Ops, Src0, Src1 and Dest encode the evolved program's operations. Data is a pointer to the input data to the programs. Outputs are placed into the Output array. The Regs array is the working registers, and is held in global memory. The other kernel parameters specify the length of the program, dimensions of the input data, number of registers available and the number of test cases to process per kernel. Listing 3 shows a similar kernel, but with the working registers now held in the faster shared memory.

Both kernels have stability issues when working when the WorkSize (the number of test cases per thread) was increased, and the program would crash. It is unclear why this occurs, as the CPU version of these kernels appears to function correctly. Unfortunately TidePowerd do not yet provide debugging tools, so we were unable to find the cause. Both types of failure are evident in Figures 1 and 2, where the sampled results do not have the same coverage. However, the shared memory interpreter when used with WorkSize=1 appeared to consistently work.

As there is a limited amount of shared memory, the block size was set to 384 threads. For the global memory kernel, a block size of 512 threads was used.

Table 2 shows statistical results from both interpreter types. 500 samples per WorkSize were used. It can be clearly seen that the shared memory version, is much faster. It it interesting to see that giving each thread more test cases to work with (and hence fewer threads running) reduced performance. The most efficient approach being to have one test case per thread.

The graphs in Figures 1 and 2 show how the interpreters performance is dependent on both the length of the evolved program and the number of test cases each thread handles. The longer the program length, the more efficient the interpreter becomes. However, in this scenario the fitness function was to find long programs. In real world situations, long programs may not be desirable (and may even be penalized by the fitness function) or may not occur due to the GP representation or operators used. With CGP, we do not expect programs to grow over time [8], and therefore we would not expect the same high performance as demonstrated here (in Section 7, this effect is investigated). With other forms of GP, such as basic versions of tree-based GP, we would expect the program length to increase over time. Since this performance-to-program length relationship appears in other GPU papers, it would be interesting to know how the apparent efficiency of the implementations would be effected by forcing the evolution to reduce program length.

| Interpreter | Global memory | | | Shared Memory | | |
|---|---|---|---|---|---|---|
| WorkSize | 1 | 128 | 2048 | 1 | 128 | 2048 |
| Minimum | 31 | 37 | 32 | 38 | 53 | 32 |
| Maximum | 434 | 401 | 333 | 1,244 | 1,113 | 467 |
| Average | 317 | 286 | 255 | 1,009 | 756 | 307 |
| Std. Dev. | 84 | 84 | 53 | 249 | 279 | 85 |

**Table 2: Speed of the two different interpreters, in MGPOps/s.**

## 6.3 Fitness Scores

Time required to compute the confusion matrix for the fitness scores is, in principle, only dependent upon the number of test cases. It was found that this took on average 0.04 seconds to compute. However, the timings showed a large variance (a standard deviation of 0.1s, median 0.03s). It is unclear why this should occur, and perhaps when better debugging tools are available, the reason will become apparent. Fitness evaluation time was also hampered by the need to do large memory transfers to move the predicted and expected outputs to the GPU. This is a current limitation with GPU.NET where the memory management cannot be hand-optimised.

# 7. CGP RESULTS

To test both CGP as a classifier, and to investigate the GPU implementation under a more real world environment, longer experiments (a maximum of 10,000 evaluations) were ran using the shared memory approach, with a work size of 1. Fitness here was the sensitivity-specificity metric discussed previously.

Looking at performance, we found that the average speed was 210 MGPOps/s (std. dev 126, maximum 701) for executing the evolved programs. When the fitness evaluation itself was taken into consideration, the average performance was 192 MGPOps/s (std. dev 118, maximum 657). As discussed previously, the average CPU speed was 64 MGPOps/s, which means that with a real fitness function, this method produces approximately a three times speed up on average, and a peak of just over ten times speed up. The relatively limited speed up is largely due to the short programs found by CGP. The average program contained only 20 operations (std. dev 14, min. 3 and maximum 84). These are therefore on the borderline to where the GPU produces an advantage. For this task, it may have been more efficient to use the multi-core CPU to perform the evaluations. It would have also been possible to implement this application to use multiple GPUs, and this would have also led to a performance increase. As these efficiencies are based on this particulars of this classification task, observed speed ups in other applications will be different.

Although not the focus of this paper, it is worth noting that the classification results from CGP appear to be very good. Over 50 experiments, it was found that the average fitness (sensitivity-specificity) was 0.955 (std. dev. 0.034). The maximum classification rates found was 99.6%, however without a validation test we cannot exclude overfitting. Modifying the fitness function to collect both training and validation fitness scores would be a straight forward extension.

# 8. CONCLUSIONS

Despite some stability issues, we found that GPU.NET is potentially a good platform to work with. Without debugging tools, it was impossible to fix any software issues in our code, or confidently identify bugs in GPU.NET. However, this situation was common with many of the first generation GPU development tools. For example, only recently has CUDA benefited from a comprehensive set of debugging tools. The speed ups obtained in the classification task, although modest, could be improved upon by using

multiple GPUs and using more recent GPUs that have more shared memory (which would allow for more threads to be run concurrently).

GPU.NET seems an attractive choice for those wishing to ease development of GPU applications. But without debugging support, it can only be currently used for simple applications. GPU.NET could further abstract away from CUDA style programming, and this would make it an even more obvious choice. It will be interesting to see how the product matures.

CGP proved itself amenable to use with a GPU interpreter. In future work we may explore the use of dynamic IL code generation (and then conversion with GPU.NET) as the execution method. Comparing the benchmark results found here to other GP on GPU papers is difficult, given the differences in efficient execution strategies of GP and CGP that come from the node reuse.

Finally, the classification results obtained by CGP are very promising. Although not the focus of the paper, it does appear that CGP can work well with such high-dimensional classification problems, and further investigation into this approach is warranted.

## 9. ACKNOWLEDGMENTS

WB acknowledges funding from Atlantic Canada's HPC network ACENET and by NSERC under the Discovery Grant Program RGPIN 283304-07. SH would like to thank Innervision Medical Ltd and TidePowerd for their technical assistance.

## 10. REFERENCES

[1] D. M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In D. Thierens, H.-G. Beyer, et al., 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.

[2] C. Elkan. Results of the kdd'99 classifier learning contest. `http://cseweb.ucsd.edu/~elkan/clresults.html`, 1999.

[3] S. Harding and W. Banzhaf. Fast genetic programming on GPUs. In M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, and A. I. Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 90–101. Springer, Valencia, Spain, 11-13 Apr. 2007.

[4] S. Harding, J. Miller, and W. Banzhaf. SMCGP2: Self modifying cartesian genetic programming in two dimensions. In *GECCO 2011 (Accepted for publication)*, 2011.

[5] S. L. Harding and W. Banzhaf. Fast genetic programming and artificial developmental systems on GPUs. In *21st International Symposium on High Performance Computing Systems and Applications (HPCS'07)*, page 2, Canada, 2007. IEEE Computer Society.

[6] S. L. Harding and W. Banzhaf. Distributed genetic programming on GPUs using CUDA. In I. Hidalgo, F. Fernandez, and J. Lanchares, editors, *Workshop on Parallel Architectures and Bioinspired Algorithms*, Raleigh, USA, Sept. 13 2009.

[7] W. Langdon. A many threaded cuda interpreter for genetic programming. In A. I. Esparcia-Alcázar, A. Ekárt, et al., editors, *Genetic Programming*, volume 6021 of *Lecture Notes in Computer Science*, pages 146–158. Springer Berlin, Heidelberg, 2010.

[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, 2001.

[9] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, et al., editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[10] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines*, 10(4):447–471, 2009.

[11] G. C. Wilson and W. Banzhaf. Deployment of parallel linear genetic programming using gpus on pc and video game console platforms. *Genetic Programming and Evolvable Machines*, 11(2):147–184, 2010.

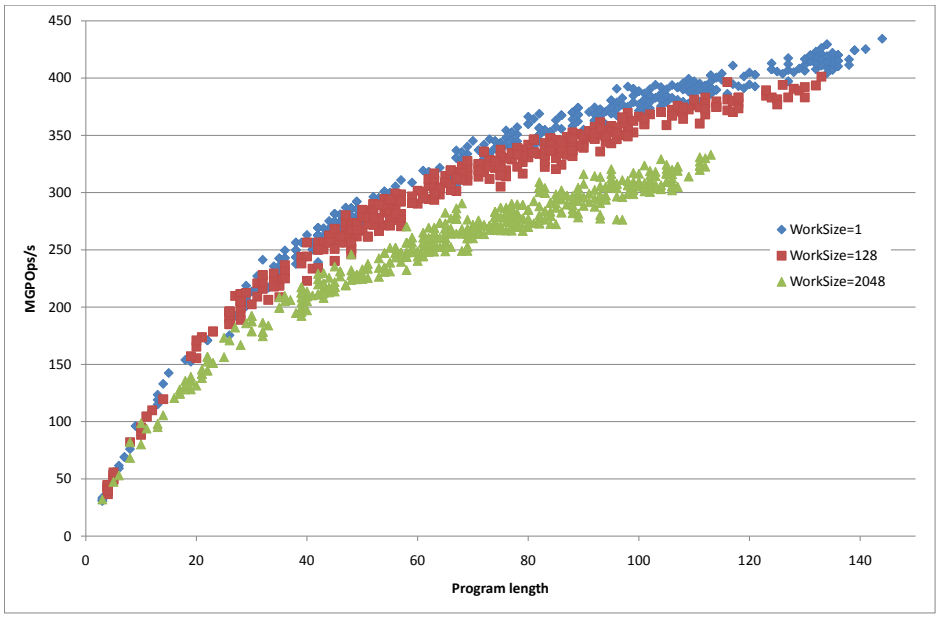


**Figure 1: Graph showing how the speed is dependent on both the length of the evolved program (in operations) and the number of test cases handled per thread (WorkSize). The results here are for the interpreter using global memory. Missing results are due to program instability.**

**Listing 2: Kernel for interpreting a program on the GPU, using global memory to store register results for intermediate workings.**

```
[Kernel]
private static void RunProgGM(int[] Ops, int[] Src0, int[] Src1,
int[] Dest, float[] Data, float[] Output, float[] Regs,
int ProgLength, int DataSize, int DataWidth, int RegCount, int WorkSize)  {
    int ThreadID = BlockDimension.X * BlockIndex.X + ThreadIndex.X;
    int TotalThreads = BlockDimension.X * GridDimension.X;
    int RegOffset = ThreadIndex.X * RegCount;
    int Op = 0; float vSrc0 = 0; float  vSrc1 = 0;

    //Clear registers (omitted for space)

    for (int v = 0; v < WorkSize; v++)  {
        int RegOffset2 = (ThreadIndex.X * RegCount);

        for (int p = 0; p < ProgLength; p++) {
            Op = Ops[p];

            if (Op == -1) break;
            if (Op == 100) {    //load
                    Regs[RegOffset2 + Dest[p]] =
                            Data[(WorkSize * ThreadID * DataWidth) + v + Src0[p]];
                    continue;
                }
            vSrc0 = Regs[RegOffset2 + Src0[p]]; vSrc1 = Regs[RegOffset2 + Src1[p]];
            if (Op == 1) //add
                Regs[RegOffset2 + Dest[p]] = vSrc0 + vSrc1;
            ...  //abridged for space
            else if (Op == 8) //sqrt
                Regs[RegOffset2 + Dest[p]] =
                TidePowerd.DeviceMethods.DeviceMath.Sqrt(vSrc0);
        }
        Output[(ThreadID * WorkSize) + v] = Regs[RegOffset2 + (RegCount - 1)];
    }
}
```

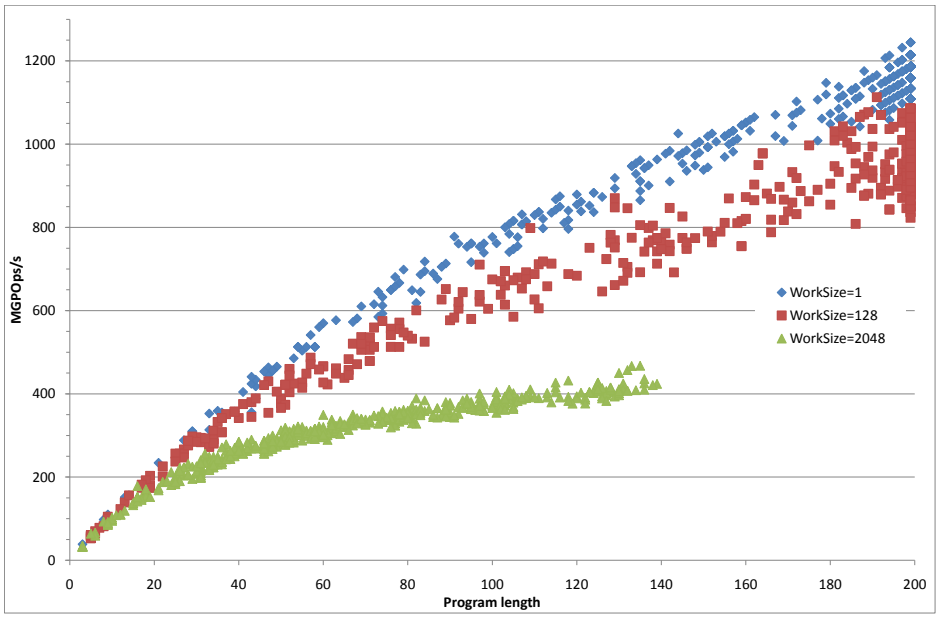


**Figure 2: Graph showing how the speed is dependent on both the length of the evolved program (in operations) and the number of test cases handled per thread (WorkSize). The results here are for the interpreter using shared memory. Using shared memory with GPU.NET is much more stable and faster than using global memory.**

**Listing 3: Kernel for interpreting a program on the GPU, using shared memory to store register results for intermediate workings.**

```
[SharedMemory(10 * 384)]  //10 working registers, 384 concurrent threads.
private static readonly float[] Regs = null;
[Kernel]
private static void RunProgSM(int[] Ops, int[] Src0, int[] Src1,
int[] Dest, float[] Data, float[] Output,
 int ProgLength, int DataSize, int DataWidth, int RegCount, int WorkSize)  {
    int ThreadID = BlockDimension.X * BlockIndex.X + ThreadIndex.X;
    int TotalThreads = BlockDimension.X * GridDimension.X;
    int RegOffset = ThreadIndex.X * RegCount;
    int Op = 0;  float vSrc0 = 0; float vSrc1 = 0;

    //Clear registers

    for (int v = 0; v < WorkSize; v++)  {
        int RegOffset2 = (ThreadIndex.X * RegCount);

        for (int p = 0; p < ProgLength; p++) {
            Op = Ops[p];

            if (Op == -1) break;
            if (Op == 100) { //load
                Regs[RegOffset2+ Dest[p]] =
                     Data[(WorkSize * ThreadID * DataWidth) + v + Src0[p]];
                continue;
            }

            vSrc0 = Regs[RegOffset2 + Src0[p]]; vSrc1 = Regs[RegOffset2 + Src1[p]];
            if (Op == 1) //add
                Regs[RegOffset2+ Dest[p]] = vSrc0 + vSrc1;
             ...   //abridged for space
                }

        Output[(ThreadID * WorkSize) + v] =
            Regs[RegOffset2 + (RegCount - 1)]; ;
    }
}
```