Cartesian Genetic Programming on the GPU

Simon Harding and Julian F. Miller

Abstract Cartesian Genetic Programming is a form of Genetic Programming based on evolving graph structures. It has a fixed genotype length and a genotype–phenotype mapping that introduces neutrality into the representation. It has been used for many applications and was one of the first Genetic Programming techniques to be implemented on the GPU. In this chapter, we describe the representation in detail and discuss various GPU implementations of it. Later in the chapter, we discuss a recent implementation based on the GPU.net framework.

1 Introduction

Cartesian Genetic Programming (CGP) was one of the first Genetic Programming representations to take advantage of the general purpose computing capabilities of modern GPUs [5]. As with other evolutionary algorithms (EAs), CGP maps well to the GPU architecture and is able to exploit the massive parallelism. But because CGP is based on a fixed length graph that allows node reuse, its implementation is distinct from the typical tree-based Genetic Programming (GP).

For those unfamiliar with CGP, an overview of the representation is provided in Sect. 2. In Sect. 3 we provide a review of the previous work of CGP on GPU. We see that GPU implementations on CGP have been used not only for typical machine

S. Harding (✉)
Machine Intelligence Ltd, Exeter, UK, EX4 IEJ
e-mail: simon@machineintelligence.co.uk

J.F. Miller
Department of Electronics, University of York, York Y01 9UD, UK
e-mail: julian.miller@york.ac.uk

learning problems such as regression, classification and image processing but also for the fitness evaluation in complex fluid dynamics problems. Finally, in Sect. 4 a recent implementation using an unusual programming approach is introduced.

2 Cartesian Genetic Programming

We give a brief overview of CGP. A more detailed account is available in the recently published book [24]. In CGP [20, 21], programs are generally represented in the form of directed acyclic graphs. These graphs are often represented as a two-dimensional grid of computational nodes. The genes that make up the genotype in CGP are integers that represent where a node gets its data, what operations the node performs on the data and where the output data required by the user is to be obtained. When the genotype is decoded, some nodes may be ignored. This happens when node outputs are not used in the calculation of output data. When this happens, we refer to the nodes and their genes as "non-coding". We call the program that results from the decoding of a genotype a phenotype. The genotype in CGP has a fixed length. However, the size of the phenotype (in terms of the number of computational nodes) can be anything from zero nodes to the number of nodes defined in the genotype. The types of computational node functions used in CGP are decided by the user and are listed in a function lookup table.

In CGP, each node in the directed graph represents a particular function and is encoded by a number of genes. One gene is the address of the computational node function in the function lookup table. We call this a *function gene*. The remaining node genes say where the node gets its data from. These genes represent addresses in a data structure (typically an array). We call these *connection genes*. Nodes take their inputs in a feed-forward manner either from the output of nodes in a previous column or from a program input. The number of connection genes a node has is chosen to be the maximum number of inputs (often called the arity) of the node functions used.

In CGP program data inputs are given the absolute data addresses 0 to n_i minus 1 where n_i is the number of program inputs. The data outputs of nodes in the genotype are given addresses sequentially, column by column, starting from n_i to $n_\mathrm{i} + L_n - 1$, where L_n is the user-determined upper bound of the number of nodes. The general form of a Cartesian genetic program is shown in Fig. 1. If the problem requires n_o program outputs, then n_o integers are generally added to the end of the genotype. In general, there may be a number of output genes (O_i) which specify where the program outputs are taken from. Each of these is an address of a node where the program output data is taken from. Nodes in columns cannot be connected to each other. In many cases graphs encoded are directed and feed-forward; this means that a node may only have its inputs connected to either input data or the output of a node in a previous column. The structure of the genotype is seen in the schematic in Fig. 1. All node function genes f_i are integer addresses in a lookup table of functions.

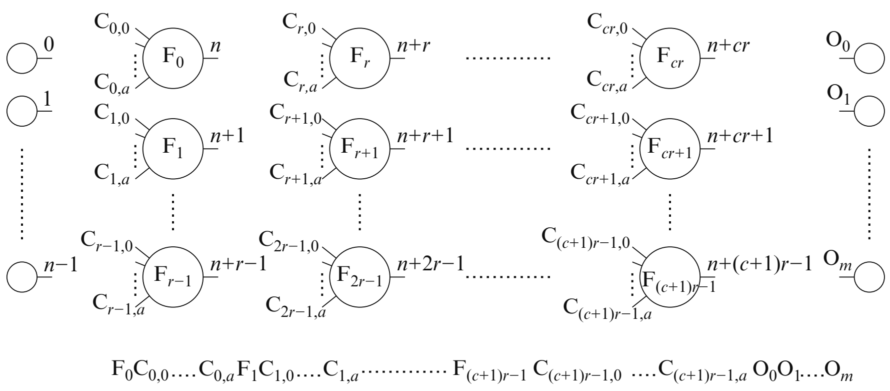

Fig. 1 General form of CGP. It is a grid of nodes whose functions are chosen from a set of primitive functions. The grid has n_c columns and n_r rows. The number of program inputs is n_i and the number of program outputs is n_o. Each node is assumed to take as many inputs as the maximum function arity a. Every data input and node output is labelled consecutively (starting at 0), which gives it a unique data address which specifies where the input data or node output value can be accessed (shown in the figure on the outputs of inputs and nodes)

All connection genes C_{ij} are data addresses and are integers taking values between 0 and the address of the node at the bottom of the previous column of nodes.

In our work here, we have used a one-dimensional geometry (one row of nodes). We have also adopted some of the features used in a developmental form of CGP [12]. We use *relative* connection addresses (rather than absolute), in which a connection gene represents how many nodes back (on the left) the node gets it inputs from. When these addresses point beyond the left end of the graph, zero is returned as a node input. The way we handle inputs and outputs is also different from classical CGP and also follows the method used in [12, 13]. This requires adding additional argument genes to all nodes (so nodes now have function, connection and argument genes), that is, a single positive floating point constant. Inputs are handled via functions: INP, INPP and SKIPINP. These functions ignore their connection genes; they return the input in the array of inputs given by an input pointer variable. After INP (INPP) the pointer is incremented (decremented). In the case of SKIPINP, the pointer is incremented by the argument of SKIPINP and then the mod operation (by the number of inputs) is taken (this ensures a valid input is always obtained). A function OUTPUT allows the input/output interpreter to find which nodes to use as output nodes at run time. The location and number of OUTPUT nodes can change over the run time of a program. When the genotype is decoded, the interpreter starts at the beginning of the encoded graph and iterates over the nodes until it finds the appropriate number of OUTPUT nodes. It then evaluates (recursively) from these nodes. The interpreter has features that allow it to cope when the number of OUTPUT nodes is different from the required number of outputs. If there are more OUTPUT nodes found than are needed, the excess nodes are simply ignored. If there are too few (or none), the interpreter starts using nodes from the end of the graph as outputs. This ensures that programs (of sufficient size) are always “viable”.

3 CGP on GPUs

Since 2007, several different implementations of CGP have been developed using different platforms and targeting different applications.

The first publication that reported an implementation of CGP on GPUs benchmarked the algorithm on regression and Boolean problems [5]. It showed that, compared to a naive, CPU-based $C\sharp$ implementation, the GPU was able to execute evolved programs hundreds of times faster. The paper used the Microsoft Accelerator framework [26], which is a *.Net* library for the GPU. Accelerator is limited to performing vector operations and cannot exploit the rich programming capabilities that recent graphics cards allow. However, it provides a simple method for GPU development. To run CGP using Accelerator, an existing CPU-based implementation was converted to use the Accelerator vector as the data type. The function set contained functions such as `add` and `subtract` that would operate on vectors and perform the chosen operation on the elements of the vectors to produce an output vector. Using this approach, each column of the input dataset was presented as a vector. The output of the program would be a single vector, containing the predicted outputs for each row. The fitness function was also implemented using Accelerator. On the GPU, the difference between the predicted and actual outputs was computed, and then a reduction was performed to find the sum of the differences. In this chapter, the EA and the interpreting of the genotype were done on the CPU, with the fitness evaluation and the execution of the individual done on the GPU.

A follow-up paper investigated the use of Accelerator for artificial developmental systems [6]. In this scenario, an artificial developmental system is defined as 2D cellular automata, where each cell contains a CGP program to define its update rules. For this problem, the GPU was seen as an ideal platform. Cellular automata are inherently parallel systems, with each cell determining the next state based on the state of its neighbours at the current time step. Although such developmental systems had been successfully implemented on the CPU [23], they tended to be small as the computational demands are high. Using the GPU, it was found that it was possible to execute large cellular automata, with complex programs and for more time steps than would have been practical on the CPU.

The update process of the artificial developmental system shares a large number of similarities with morphological image operations. Here, a kernel takes the values of a neighbourhood of pixels and outputs a new value for the centre pixel of that neighbourhood. A simple example is a smoothing operation that outputs a weighted sum of the neighbouring pixels. Although weighted sums are the most common forms of these kernels, it is possible to use complicated expressions, including evolved programs. As expected, it was found that CGP on the GPU was able to rapidly find such programs [4, 7, 8]. Before such GPU implementations were available, most work on evolving image filters was restricted to fitness evaluation with a single 256×256 (or smaller) image. This led to problems of overfitting. However, by implementing the processing on the GPU, many images, each with different properties, could be efficiently tested.

Initially, suitable GPUs for GPGPU were relatively expensive and obscure. However, they quickly became a standard component in recent systems. For example, when a student computer lab was updated, it became possible to test the use of multiple GPUs as part of an ad hoc cluster [11]. Previous versions of CGP on GPUs performed evaluated one individual at a time, with the fitness cases being operated on in parallel. Using a cluster of GPUs, it was possible to evaluate the population in parallel, which in turn increased the speed at which the population could be evaluated.

Evaluating programs with loops and recursion can be extremely computationally expensive, and this is one of the reasons that evolution of such programs is under-represented in the literature. Using GPUs though reduces this computational cost significantly. Although CGP is typically used to evolve feed-forward expressions, by removing the restriction that nodes must connect to previous columns, it is possible to evolve cyclic graphs. It is possible to evaluate cyclic CGP on the GPU very efficiently and process individuals hundreds of times faster than with a single CPU [18].

All of these previous examples looked at more traditional applications of GP, such as regression and classification. However, GP can also be used for other applications—such as shape design [2, 15, 19]. In [10], GPUs were used to evaluate a computational fluid dynamics (CFD) simulation to test an evolved wing design. A form of CGP based on Self-Modifying CGP [12, 13] was used to generate the cross-sectional shape of an airfoil. This was then simulated to determine its lift and drag properties. Using the GPU, the simulation speeds were dramatically improved. To further increase performance, an ad hoc cluster of GPUs was used to evaluate the population in parallel. Care was taken to ensure that the EA could run totally asynchronously, as CFD simulations vary in how long they take to execute depending on factors such as how much turbulence is generated.

4 Example: CGP for Classification

4.1 GPU.NET

As shown in previous sections, GPU programming often requires specialist programming skills. Although the development tools have been considerably improved in recent years, they are still difficult to work with. In the literature, we see that there are many different ways to program GPUs and that EA have been implemented with most of the common types. Most EAs have been written using NVidia's CUDA, with a few more recent examples using OpenCL. It is also possible to write low-level shader kernels (using Cg, 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.

CGP has previously been implemented in both CUDA and Accelerator. However, in this chapter we present an implementation based on a recent commercial, closed-source tool for programming GPUs from TidePowerd called GPU.NET. Like cuda.net and MS Accelerator, GPU.NET is designed to work with Microsoft's .Net Common Language Runtime (CLR). GPU.NET's main feature is that it converts Intermediate Language (IL) in an already compiled .Net assembly to device code (e.g. PTX instructions for NVidia graphics cards). GPU.NET's tool rewrites the assembly converting flagged methods 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 any .Net-managed language, such as $C\sharp$, 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 optimisations have been (or can be) made. It should be noted that the generated PTX is further compiled by the device driver before execution. Compared to a compiled 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 can have an overhead of a few microseconds. Therefore, as with all GPU programming, care has to be taken to ensure that the unit of work given to the GPU is sufficiently large to make the overheads negligible.

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);
}
```

4.2 A GPU-Based Interpreter for CGP

Looking back at previous CGP implementations and other work in the field, we see that 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 of GP on the GPU, one implementation compiled individual programs from generated Cg shader code and then loaded into the GPU for execution [1]. The MS Research Accelerator toolkit generates and compiles a DirectX shader program transparently. It then executes this program. In [9], 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.

Another 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 of this approach is that the GPU program only needs to be compiled and loaded once and that the only thing that changes is the data that represents the programs to be tested for fitness. The interpreter approach has been used with CUDA [17,25] and DirectX shaders [27]. Without the pre-compilation overhead, the interpreters work well for smaller amounts of data and for shorter programs. However, interpreters introduce their own performance issues, as there is now the continuous overhead of parsing the evolved program in addition to performing the computation. Fortunately on GPUs, the branching that is typically required in an interpreter is handled very efficiently.

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 and read a reverse Polish version of the tree and use only two registers to store intermediate results.

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 multilayered 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 32-bit registers. In version 1.2, this was increased to 16,000 and in version 2 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 16 KB of storage on 1.x compute capabilities and 48 KB 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 analysed 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 [22], which has implications for perceived efficiency that we observe later in this chapter. 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(). The genotype size was set to 100,000 nodes. Each gene has an equality probability (0.01) of being mutated. A 1+4 evolutionary strategy (ES) was used [24].

To test using GPU.NET, three different interpreter strategies were considered. All three strategies were implemented to run the same format instruction code (IC). In the next section, the instruction code is discussed. The following section describes how the different strategies worked.

4.3 Generating the Instruction Code

Generating the "interpreted code" (IC) from CGP requires a number of steps, which are all performed on the CPU side. In the first step, the genotype is analysed 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 [12,13]. 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 the same 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 ten 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 is to work forward 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, in the former such connections return zero.

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.

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.

4.4 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. 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 ten vectors to use as working registers (see Sect. 4.5).

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 16 KB. With ten 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 a risk of underutilising 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

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

	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

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.

4.5 Sizing for Interpreter Parameters

To determine the number of registers needed and the maximum amount of storage we needed for the interpreter, we analysed 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 convenience, we chose to use ten 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.

4.6 Fitness Function

For benchmarking, two fitness functions were implemented. Both are based on the KDD Cup Challenge 1999 data [3, 16]. 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 $1/512$ the data). Each thread counts the number of true/false positives/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 redundancy, this is not the case. With the bloat-free evolution of CGP, 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.

4.7 Speed 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 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 (2.6 GHz) processor, running Windows 7, with a Tesla C1060 (240 CUDA cores, 4 GB, driver version 258.96).

4.8 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 (ten registers, each the size of number of rows of the input data). Therefore, these benchmarks are formed 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 an 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.

4.9 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 does not yet provide debugging tools, so we were unable to find the cause. Both types of failure are evident in Figs. 2 and 3, where the

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)];
    }
}
```

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)]; ;
    }
}
```

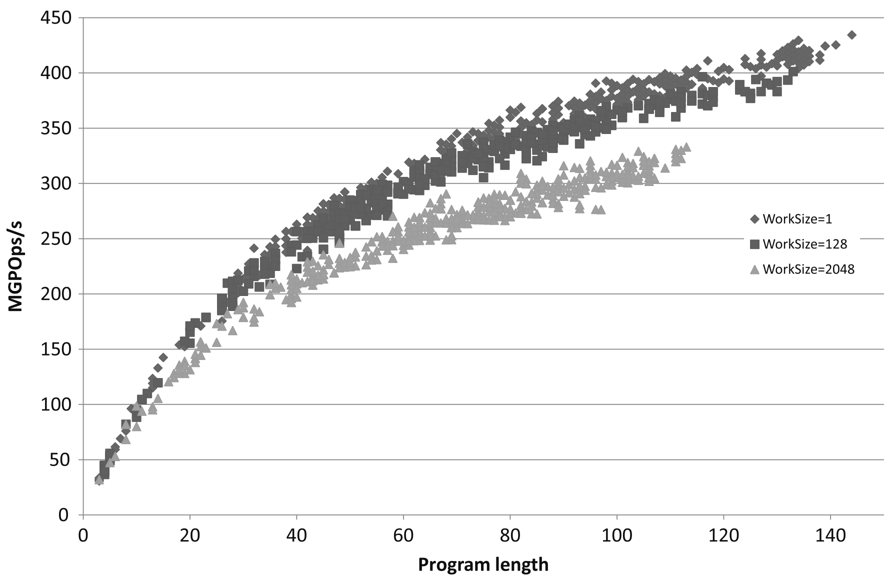

Fig. 2 Graph showing how the speed is dependent on both the length of the evolved program (in operations) and the number of test cases handled per thread (WorkSize). The results here are for the interpreter using global memory. Missing results are due to program instability

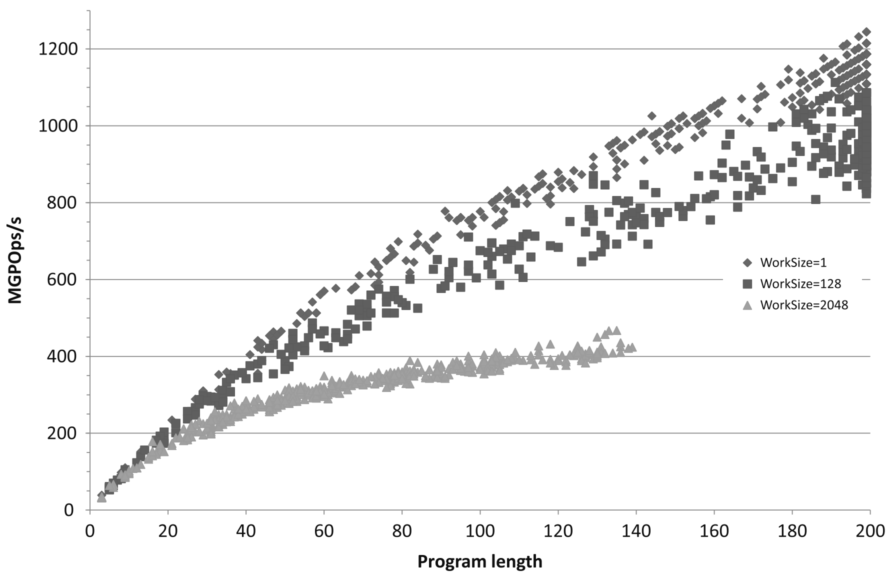

Fig. 3 Graph showing how the speed is dependent on both the length of the evolved program (in operations) and the number of test cases handled per thread (WorkSize). The results here are for the interpreter using shared memory. Using shared memory with GPU.NET is much more stable and faster than using global memory

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

Interpreter	Global memory			Shared memory		
WorkSize	1	128	2,048	1	128	2,048
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

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. Five hundred samples per WorkSize were used. It can be clearly seen that the shared memory version is much faster. It is interesting to see that giving each thread more test cases to work with (and hence fewer threads running) reduced performance. The most efficient approach is to have one test case per thread.

The graphs in Figs. 2 and 3 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 penalised 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 [22], and therefore we would not expect the same high performance as demonstrated here (in Sect. 4.11, 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.

4.10 Fitness Scores

The 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 s to compute. However, the timings showed a large variance (a standard deviation of 0.1 s, median 0.03 s). 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.

4.11 CGP Classification Results

Both to test 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 run using the shared memory approach, with WorkSize = 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 of the area where the GPU produces an advantage. For this task, it may have been more efficient to use the multicore 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 these particulars of this classification task, observed speed-ups in other applications will be different.

Although not the focus of this chapter, it is worth noting that the classification results from CGP appear to be very good. In over 50 experiments, it was found that the average fitness (sensitivity-specificity) was 0.955 (std. dev. 0.034). The maximum classification rate 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 straightforward extension.

5 Conclusions

CGP was one of the first Genetic Programming techniques implemented on the GPU and has been successfully used to solve many different problem types using GPUs. Although this chapter focused on TidePowerd's implementation, CGP has successfully been developed using technologies such as CUDA and MS Accelerator. In all circumstances, significant speed-ups have been reported.

As GPU technology improves, both in terms of hardware features and software development, it is likely that more advanced CGP approaches such as SMCGP and MT-CGP [14] will be implemented. SMCGP requires a very flexible environment to work in, as programs can change dramatically during their run time. MT-CGP operates on multiple data types, and this presents some interesting challenges when developing a high-performance system. However, as GPGPU becomes ever more flexible, we expect that CGP will be able to take advantage of the new capabilities.

Acknowledgements SH acknowledges the 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.

References

1. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: Thierens, D., Beyer, H.G., et al. (eds.) GECCO '07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, vol. 2, pp. 1566–1573. ACM Press, London (2007)
2. Coates, P.: Using Genetic Programming and L-systems to explore 3D design worlds. In: CAADFutures'97. Kluwer Academic, Dordecht (2008)
3. Elkan, C.: Results of the KDD'99 classifier learning contest. http://cseweb.ucsd.edu/~elkan/clresults.html (1999)
4. Harding, S.: Evolution of image filters on graphics processor units using Cartesian genetic programming. In: Wang, J. (ed.) 2008 IEEE World Congress on Computational Intelligence. IEEE Computational Intelligence Society, IEEE Press, Hong Kong. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4631051 (2008)
5. Harding, S., Banzhaf, W.: Fast genetic programming on GPUs. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) Proceedings of the 10th European Conference on Genetic Programming. Lecture Notes in Computer Science, vol. 4445, pp. 90–101. Springer, Valencia (2007). doi:10.1007/978-3-540-71605-1_9. http://www.springerlink.com/index/w57468k30j124410.pdf
6. Harding, S.L., Banzhaf, W.: Fast genetic programming and artificial developmental systems on GPUs. In: 21st International Symposium on High Performance Computing Systems and Applications (HPCS'07), p. 2. IEEE Computer Society, Canada (2007). doi:10.1109/HPCS.2007.17. http://doi.ieeecomputersociety.org/10.1109/HPCS.2007.17
7. Harding, S., Banzhaf, W.: Genetic programming on GPUs for image processing. In: Lanchares, J., Fernandez, F., Risco-Martin, J. (eds.) Proceedings of the First International Workshop on Parallel and Bioinspired Algorithms (WPABA-2008), Toronto, Canada, 2008, pp. 65–72. Complutense University of Madrid Press, Madrid. http://www.inderscience.com/search/index.php?action=record&rec_id=24207&prevQuery=&ps=10&m=or (2008)
8. Harding, S., Banzhaf, W.: Genetic programming on GPUs for image processing. Int. J. High Perform. Syst. Archit. **1**(4), 231–240 (2008). doi:10.1504/IJHPSA.2008.024207. http://www.inderscience.comkern-1pt/search/index.php?action=record&rec_id=24207&prevQuery=&ps=10&m=or
9. Harding, S.L., Banzhaf, W.: Distributed genetic programming on GPUs using CUDA. In: Hidalgo, I., Fernandez, F., Lanchares, J. (eds.) Workshop on Parallel Architectures and Bioinspired Algorithms. Raleigh, USA. http://www.evolutioninmaterio.com/preprints/CudaParallelCompilePP.pdf (2009)
10. Harding, S., Banzhaf, W.: Optimizing shape design with distributed parallel genetic programming on GPUs. In: Fernández de Vega, F., Hidalgo Pérez, J.I., Lanchares, J. (eds.) Parallel Architectures and Bioinspired Algorithms. Studies in Computational Intelligence, vol. 415, pp. 51–75. Springer, Berlin (2012)
11. Harding, S.L., Banzhaf, W.: Distributed genetic programming on GPUs using CUDA. http://www.evolutioninmaterio.com/preprints/Technicalreport (submitted)
12. Harding, S., Miller, J.F., Banzhaf, W.: Developments in Cartesian genetic programming: self-modifying CGP. Genet. Program. Evolvable Mach. **11**(3–4), 397–439 (2010)
13. Harding, S., Miller, J.F., Banzhaf, W.: A survey of self modifying CGP. Genetic Programming Theory and Practice, 2010. http://www.evolutioninmaterio.com/preprints/ (2010)

14. Harding, S., Graziano, V., Leitner, J., Schmidhuber, J.: MT-CGP: Mixed type cartesian genetic programming. In: Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference, pp. 751–758. ACM, New York (2012)
15. Hotz, P.E.: Evolving morphologies of simulated 3D organisms based on differential gene expression. In: Proceedings of the Fourth European Conference on Artificial Life, pp. 205–213. Elsevier Academic, London (1997)
16. KDD Cup 1999 Data: Third international knowledge discovery and data mining tools competition. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html (1999)
17. Langdon, W.B.: A many threaded CUDA interpreter for genetic programming. In: Esparcia-Alcázar, A.I., Ekárt, A., et al. (eds.) Genetic Programming. Lecture Notes in Computer Science, vol. 6021, pp. 146–158. Springer, Berlin (2010)
18. Lewis, T.E., Magoulas, G.D.: Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO '09, pp. 1379–1386. ACM, New York (2009). doi:10.1145/1569901.1570086. http://doi.acm.org/10.1145/1569901.1570086
19. Lohn, J.D., Hornby, G., Linden, D.S.: Human-competitive evolved antennas. AI EDAM **22**(3), 235–247 (2008)
20. Miller, J.F.: An empirical study of the efficiency of learning Boolean functions using a Cartesian genetic programming approach. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 1135–1142. Morgan Kaufmann, Los Altos (1999)
21. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proceedings of European Conference on Genetic Programming. Lecture Notes in Computer Science, vol. 1802, pp. 121–132. Springer, Berlin (2000)
22. Miller, J.: What bloat? Cartesian genetic programming on Boolean problems. In: Goodman, E.D. (ed.) 2001 Genetic and Evolutionary Computation Conference Late Breaking Papers, pp. 295–302. Morgan Kaufmann (2001)
23. Miller, J.F.: Evolving a self-repairing, self-regulating, French flag organism. In: Deb, K., Poli, R., Banzhaf, W., et al. (eds.) GECCO (1). Lecture Notes in Computer Science, vol. 3102, pp. 129–139. Springer, Berlin (2004)
24. Miller, J.F. (ed.): Cartesian Genetic Programming. Natural Computing Series. Springer, Berlin (2011)
25. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Genetic programming on graphics processing units. Genet. Program. Evolvable Mach. **10**(4), 447–471 (2009)
26. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program GPUs for general-purpose uses. In: ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 325–335. ACM, New York (2006). http://doi.acm.org/10.1145/1168857.1168898
27. Wilson, G.C., Banzhaf, W.: Deployment of parallel linear genetic programming using GPUs on PC and video game console platforms. Genet. Program. Evolvable Mach. **11**(2), 147–184 (2010)