Chapter 8
Hardware Acceleration for CGP: Graphics Processing Units

Simon L. Harding and Wolfgang Banzhaf

8.1 Graphics Processing Units

Graphic Processing Units (GPUs) are fast, highly parallel units. In addition to processing 3D graphics, modern GPUs can be programmed for more general-purpose computation. A GPU consists of a large number of 'shader processors', and conceptually operates as a single instruction multiple data (SIMD) or multiple instruction multiple data (MIMD) stream processor. A modern GPU can have several hundred of these stream processors, which, combined with their relatively low cost, makes them an attractive platform for scientific computing. In the last two years, the genetic programming community has begun to exploit GPUs to accelerate the evaluation of individuals in a population [1, 4].

CGP was the first GP technique implemented in a general-purpose fashion on GPUs. By 'general purpose', we mean a technique that can be applied to a number of GP applications and not just a single, specialized task. Implementing CGP on GPUs has resulted in very significant performance increases.

In this chapter, we discuss several of our implementations of CGP on GPUs. To begin with, we start with an overview of the hardware and software used, before discussing applications and the speed-ups obtained.

8.2 The Architecture of Graphics Processing Units

Graphics processors are specialized stream processors used to render graphics. Typically, a GPU is able to perform graphics manipulations much faster than a general-purpose CPU, as graphics processors are specifically designed to handle certain primitive operations. Internally, a GPU contains a number of small processors that are used to perform calculations on 3D vertex information and on textures. These processors operate in parallel with each other, and work on different parts of the problem. First, the vertex processors calculate a 3D view, and then the shader processors paint this model before it is displayed. Programming a GPU is typically done through a virtual-machine interface such as OpenGL or DirectX, which provides a common interface to the diverse GPUs available, thus making development easy. However, DirectX and OpenGL are optimized for graphics processing, and hence other APIs are required to use a GPU as a general purpose device. There are many such APIs, and Sect. 8.3 describes several of the ones that have been used to implement CGP.

For general-purpose computing, we wish here to make use of the parallelism provided by the shader processors (see Fig. 8.1). Each processor can perform multiple floating-point operations per clock cycle, meaning that the performance is determined by the clock speed, the number of pixel shaders and the width of the pixel shaders. Pixel shaders are programmed to perform a given set of instructions on each pixel in a texture. Depending on the GPU, the number of instructions may be limited. In order to use more than this number of operations, a program needs to be broken down into suitably sized units, which may impact performance. Newer GPUs support unlimited instructions, but some older cards support as few as 64 instructions. GPUs typically use floating-point arithmetic, the precision of which is often controllable, as less precise representations are faster to compute with. Again, the maximum precision is manufacturer-specific, but recent cards provide up to 128-bit precision.

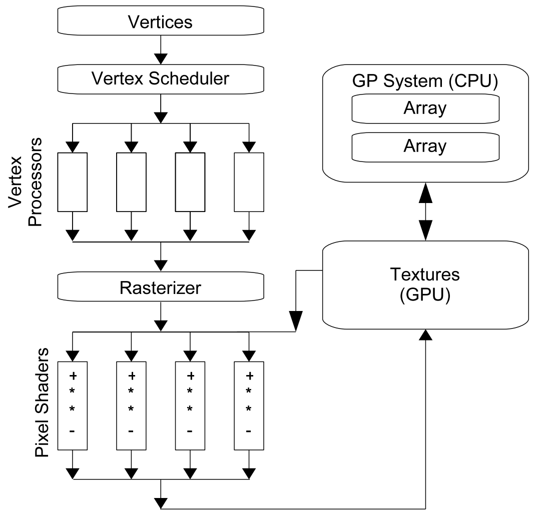

Fig. 8.1 Arrays, representing test cases, are converted to textures. These textures are then manipulated (in parallel) by small programs inside each of the pixel shaders. The result is another texture, which can be converted back to a normal array for CPU-based processing.

A variety of different graphics cards were used in our experiments. In particular, we used three cards from Nvidia: GeForce 7300 Go, GeForce 8600 and GeForce 9800.

8.3 Programming a GPU

We experimented with two different software APIs for programming GPUs: Microsoft's Accelerator and the API from Nvidia.

Accelerator is a .Net assembly from Microsoft Research that provides access to the GPU via the DirectX interface [15]. The system is completely abstracted from the GPU, and presents the end user with only arrays that can be operated on in parallel. Unfortunately, the system is available only for the Windows platform, owing to its reliance on DirectX. However, the assembly can be used from any .Net programming language. Conversion to textures and transfer to the GPU are handled transparently by the API, allowing the developer to concentrate on the implementation of the algorithm. The function set available for operating on parallel arrays is similar to those for other APIs. It includes element wise arithmetic operations, and square root, multiply–add and trigonometric operations. There are also conditional operations and functions for comparing two arrays. The API also provides reduction operators, such as the sum, product, minimum value and maximum value of an array. Further functions perform transformations such as shift and rotate on the elements of an array.

One benefit of this tool is that it uses lazy evaluation. Operations are not performed on the data until the evaluated result is requested. This enables a certain degree of real-time optimization, and reduces the computational load on the GPU. In particular, it enables optimization of common subexpressions, which reduces the creation of temporary shaders and textures. The movement of data to and from the GPU can also be optimized efficiently, which reduces the impact of the relatively slow transfer of data out of the GPU. The compilation to the shader model occurs at run time, and hence can automatically make use of the different features available on the various graphics cards supported.

CUDA is an extension of the C programming language developed by Nvidia. It allows low-level access to the GPU and therefore can make optimal use of the hardware. CUDA can be viewed as a set of language extensions that provide support for parallel programming using massive numbers of threads. The API provides all the tools needed to marshal data to and from the GPU and for the actual algorithm implementation. The programs look very much like C programs.

CUDA programs are typically written as pure C programs. However, many libraries exist that expose the programming interface of the CUDA driver to other programming environments. In the work described here, the CUDA.net library from GASS [3] was used. This library allows any .Net language to communicate with the CUDA driver.

Accessing the CUDA driver allows applications to access the GPU memory, load code libraries (modules) and execute functions. The libraries themselves still need to be implemented in C and compiled using the CUDA compiler. During this compilation process, the CUDA compiler can optimize the generated code.

In addition to the overhead of compiling shader programs, GPU programming also has a number of other bottlenecks. In particular, there is a small cost of transferring data to or from the GPU memory.

8.4 Parallel Implementation of GP

Implementations of GP on GPUs have largely fallen into two distinct evaluation methodologies: population-parallel and fitness-case-parallel. Both methods exploit the highly parallel architecture of the GPU. In the fitness-case-parallel approach, all the fitness cases are executed in parallel, with only one individual being evaluated at a time. This can be considered as a SIMD approach. In the population-parallel approach, multiple individuals are evaluated simultaneously. Both approaches have problems that impact performance.

The fitness-case-parallel approach requires that a different program is uploaded to the GPU for each evaluation. This introduces a large overhead. The programs need to be compiled (on the CPU side) and then loaded into the graphics card before any evaluation can occur. Previous work has shown that for smaller numbers of fitness cases (typically less than 1000), this overhead is larger than the increase in computational speed [2, 7, 6]. In these cases, there is no benefit from executing the program on the GPU. For some problems, there are fewer fitness cases than shader processors. In this scenario, some of the processors are idle. This problem will become worse as the number of shader processors increases with new GPU designs. Unfortunately, a large number of classic benchmark GP problems fit into this category. However, there are problems such as image processing that naturally provide large numbers of fitness cases (as each pixel can be considered as a different fitness case) [5, 8].

GPUs can be considered as processors (or in which each processor handles multiple copies of the same program, using different program counters [14]). When a population of programs is to be evaluated in parallel, the common approach so far has been to implement some form of interpreter on the GPU [14, 13, 16]. The interpreter is able to execute the evolved programs (typically machine-code-like programs represented as textures) in a pseudo-parallel manner. To see why this execution is pseduo-parallel, the implementation of the interpreter needs to be considered.

A population can be represented as a 2D array, with individuals represented in the columns and their instructions in the rows. The interpreter consists of a set of IF statements, where each condition corresponds to a given operator in the GP function set. The interpreter looks at the current instruction of each individual, and sees if it matches the current IF statement condition. If it does, that instruction is executed. This happens for each individual in the population in parallel, and for some individuals the current instruction may not match the current IF statement. These individuals must wait until the interpreter reaches their instruction type. Hence, with this approach, some of the GPU processors are not doing useful computation. If there are four functions in the function set, we can expect that on average at least 3/4 of the shader processors will be ‘idle’. If the function set is larger, then more shaders will be ‘idle’. Despite this idling, population-parallel approaches can still be considerably faster than CPU implementations.

The performance of population-parallel implementations is therefore impacted by the size of the function set used. Conversely, the fitness-case-parallel approaches are impacted by the number of fitness cases. Population-parallel approaches are also largely unsuited for GPU implementation when small population sizes are required.

In our work, we used a fitness-case-parallel version of the algorithm. Both Accelerator and CUDA compile the code before execution. Although this introduces a significant overhead, it does have some advantages, namely that the compiler can optimize the executed code.

With CGP, it is very commonly to observed that large numbers of nodes in the genotype are reused, and hence there are a lot of intermediate results that can be reused during execution. With an interpreter-based implementation, this would be very hard to implement. However, with compilation, we can rely on the compiler (and our own code generator) to spot where such work is redundant and remove it.

When running in a fitness-parallel fashion, the processing can be considered as vector mathematics. Each column in the data set is represented as a vector of values. The programs then operate on these vectors, in an elementwise fashion. Each shader program runs on one index of the vector, and hence each shader program is responsible for one row in the data set. Figure 8.2 illustrates this approach.

In the work demonstrated here, only the fitness evaluation was implemented on the GPU. The evolutionary algorithm (and other parts of the software) were all implemented on the CPU side, as shown in Fig. 8.1.

8.5 Initial GPU Results

In our initial papers [7, 6], we used an Nvidia 7300 Go GPU. The programming used the Accelerator package. At the time of writing, this GPU is still in use, but is a very dated and underpowered platform. The GP parser used here was written in C♯, and compiled using Visual Studio 2005. All benchmark tests were done using the Release build configuration, and were executed on a CLR 2.0 with Windows XP. The GPU was an Nvidia GeForce 7300 Go with 512 Mb video memory. The CPU used was an Intel Centrino T2400 (running at 1.83 Ghz), with 1.5 Gb of system memory.

In these experiments, GP trees were randomly generated with a given number of nodes. The expressions were evaluated on the CPU and then on the GPU, and each evaluation was timed. For each input-size/expression-length pair, 100 different randomly generated expressions were used, and the results were averaged to calcu-

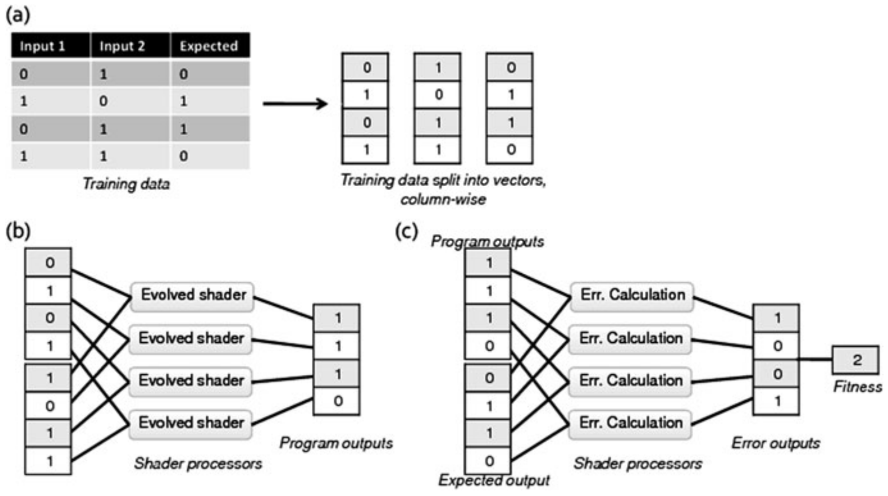

Fig. 8.2 (a) The data set is split into vectors. Each vector contains one column. (b) These vectors are passed to the evolved shader program. Each shader program runs on a single index, so it is able to see only one row of the data set. (c) The output of the evolved program is compared with the expected output, and a fitness value obtained. All these steps exploit the parallelism of the shader processor.

late acceleration factors. Therefore our results showed the average number of times by which the GPU was faster at evaluating a given tree size for a given number of fitness cases. Results less than 1 mean that the CPU was faster at evaluating the expression, and values above 1 indicate that the GPU performed better.

8.5.1 Floating-Point-Based Expressions

In the first experiment, we evaluated random GP trees containing varying numbers of nodes, and exposed them to test cases of varying sizes. The mathematical functions $+$, $-$, $*$ and $/$ were used. The same expression was tested on the CPU and the GPU, and the speed difference was recorded. The results are shown in Table 8.1. For small node counts and fitness cases, the CPU performance is superior because of the overhead of mapping the computation to the GPU. For larger problems, however, there is a massive speed increase for GPU execution.

Table 8.1 Results showing the number of times faster evaluating floating-point-based expressions is on the GPU compared with CPU implementation. An increase of less than 1 shows that the CPU is more efficient

	Test cases					
Expression length	64	256	1024	4096	16384	65536
10	0.04	0.16	0.6	2.39	8.94	28.34
100	0.4	1.38	5.55	23.03	84.23	271.69
500	1.82	7.04	27.84	101.13	407.34	1349.52
1000	3.47	13.78	52.55	204.35	803.28	2694.97
5000	10.02	26.35	87.46	349.73	1736.3	4642.4
10000	13.01	36.5	157.03	442.23	1678.45	7351.06

8.5.2 Binary

The second experiment compared the performance of the GPU at handling Boolean expressions. In the CPU version, we used the C♯ Boolean type – which is convenient, but not necessarily the most efficient representation. For the GPU, we tested two different approaches, one using the Boolean parallel array provided by Accelerator, and the other using floating-point numbers. The performance of these two representations is shown in Table 8.2. It is interesting to note that improvements are not guaranteed. As can be seen in the table, the speed-up can decrease as the expression size increases. We assume that this is due to the way in which large shader programs are handled by either Accelerator or the GPU. For example, the length of the shader program on the Nvidia GPU may be limited, and going beyond this length would require repeated passes of the data. This type of behaviour can be seen in many of the results presented here.

We limited the functions in the expressions to AND, OR and NOT, which are supported by the Boolean array type. Following some sample code provided with Accelerator, we mimicked Boolean behaviour using 0.0f as false and 1.0f as true. For two floats, AND can be viewed as the minimum of the two values. Similarly, OR can be viewed as the maximum of the two values. NOT can be performed as a multiply–add, where the first stage is to multiply by minus 1 then 1 is added.

8.5.3 Regression and Classifcation

Next we investigated the speed-up on both toy and real-world problems, rather than on arbitrary expressions. In the benchmark experiments, the expression lengths were uniform throughout the tests. However, in real GP, the lengths of the expressions vary throughout the run. As the GPU sometimes results in slower performance, we need to verify that on average there is an advantage.

Table 8.2 Results showing the number of times faster evaluating Boolean expressions is on the GPU, compared with CPU implementation. An increase of less than 1 shows that the CPU is more efficient. Booleans were implemented as floating-point numbers and as Booleans. Although representation as floating-point numbers provides faster valuation than that obtained with the CPU faster than the CPU for large input sizes, in general it appears preferential to use the Boolean representation. Using floating-point representation can provide speed increases, but the results are varied

Boolean implementation								
	Test cases							
Expression length	4	16	64	256	1024	4096	16384	65536
10	0.22	1.04	1.05	2.77	7.79	36.53	84.08	556.40
50	0.44	0.57	1.43	3.02	14.75	58.17	228.13	896.33
100	0.39	0.62	1.17	4.36	14.49	51.51	189.57	969.33
500	0.35	0.43	0.75	2.64	14.11	48.01	256.07	1048.16
1000	0.23	0.39	0.86	3.01	10.79	50.39	162.54	408.73
1500	0.40	0.55	1.15	4.19	13.69	53.49	113.43	848.29

Boolean implementation, using floating-point numbers								
	Test cases							
Expression length	4	16	64	256	1024	4096	16384	65536
10	0.024	0.028	0.028	0.072	0.282	0.99	3.92	14.66
50	0.035	0.049	0.115	0.311	1.174	4.56	17.72	70.48
100	0.061	0.088	0.201	0.616	2.020	8.78	34.69	132.84
500	0.002	0.003	0.005	0.017	0.064	0.25	0.99	3.50
1000	0.001	0.001	0.003	0.008	0.030	0.12	0.48	1.49
1500	0.000	0.001	0.002	0.005	0.019	0.07	0.29	1.00

8.5.3.1 Regression

We evolved functions that performed regression with respect to $x^6 - 2x^4 + x^2$ [11]. We measured the difference in the speed of evaluation using a number of test cases. In each instance, the test cases were uniformly distributed between -1 and +1. We also changed the maximum length of the CGP graph. Hence, the expression lengths could range anywhere from one node to the maximum size of the CGP graph. GP was run for 200 generations to allow convergence. The function set comprised $+$, $-$, $*$ and $/$. In C♯, division by zero of a float returns 'Infinity', which is consistent with the result from the Accelerator library.

The fitness was defined as the sum of the absolute errors for each test case and the output of the expression. This could also be calculated using the GPU. Each individual was evaluated with the CPU and then the GPU and the speed difference was recorded. Also, the outputs from the GPU and CPU were compared to ensure that they were evaluating the expression in the same manner. We did not find any instances where the two differed.

Table 8.3 shows results that are consistent with the results for the other tests described above. For smaller input sets and small expressions, it was more efficient to evaluate them on the CPU. However, for the larger test and expression sizes, the performance increase was dramatic.

Table 8.3 Results for the regression experiment. The results show the number of times faster evaluating evolved GP expressions is on the GPU, compared with CPU implementation. The maximum expression length is the number of nodes in the CGP graph

	Test cases			
Max. expressionlLength	10	100	1000	2000
10	0.02	0.08	0.7	1.22
100	0.07	0.33	2.79	5.16
1000	0.42	1.71	15.29	87.02
10000	0.4	1.79	16.25	95.37

8.5.3.2 Classification

In this experiment we attempted to solve the classification problem of distinguishing between two spirals, as described in [11]. This problem has two input values (the x and y coordinates of a point on a spiral) and has a single output indicating which spiral the point is found to be on. In [11], 194 test cases were used. In the present experiments, we extended the number of test cases to 388, 970 and 1940. We also extended the function set to include sine, cosine, $\sqrt{x}$, x^y and a comparator. The comparator looked at the first input value of a node, and if it was less than or equal to zero returned the second input, and 0 otherwise. The relative speed increases can be seen in Table 8.4. Again we see that the GPU is superior for larger numbers of test cases, with larger expression sizes.

Table 8.4 Results for the two-spirals classification experiment. The results show the number of times faster evaluating evolved GP expressions is on the GPU compared with CPU implementation. The maximum expression length is the number of nodes in the CGP graph

	Test cases			
Max. expression length	194	388	970	1940
10	0.15	0.23	0.51	1.01
100	0.38	0.67	1.63	3.01
1000	1.77	3.19	9.21	22.7
10000	1.69	3.21	8.94	22.38

8.5.3.3 Classification in Bioinformatics

In this experiment we investigated the behaviour on another classification problem, this time a protein classifier, as described in [12]. Here the task was to predict the location of a protein in a cell from the amino acids in that particular protein. We used the entire data set as the training set. The set consisted of 2427 entries, with 19 variables each and one output. We investigated the performance gain using several expression lengths, and the results can be seen in Table 8.5. Here, the large number of test cases used results in considerable improvements in evaluation time, even for small expressions.

Table 8.5 Results for the protein classification experiment. The results show the number of times faster evaluating evolved GP expressions is on the GPU compared with CPU implementation. The maximum expression length is the number of nodes in the CGP graph

	Test cases
Expression length	2427
10	3.44
100	6.67
1000	11.84
10000	14.21

8.6 Image Processing on the GPU

In a later set of experiments, we demonstrated the use of the GPU as a platform for the evolution of image filters, such as filters for noise reduction. An image filter is defined as a convolution kernel (or program) that takes a set of inputs (pixels in an image) and maps them to a new pixel value. This program is applied to each pixel in an image to produce another image. Such filters are common in image processing. For example, one commonly used filter to reduce noise in an image is a median filter, where the new state of a pixel is the median value of it and its neighbourhood.

Using CGP it is possible to evolve such filters. Further examples of this and other approaches can be seen in Chap. 6.

Running the filters on a GPU allows us to apply the kernel to every pixel (logically, but not physically) simultaneously. The parallel nature of the GPU allows a number of kernels to be calculated at the same time. This number is dependent on the number of shader processors available. The image filter consists of an evolved program that takes a pixel and its neighbourhood (a total of nine pixels) and computes the new value of the centre pixel. On a traditional processor, you would iterate over each pixel in turn and execute the evolved program each time. Using the parallelism of the GPU, many pixels (in effect all of them) can be operated on simultaneously. Hence, the evolved program is only evaluated once. Although the evolved program actually evaluates the entire image at once, we can break down the problem and consider what is required for each pixel. For each pixel, we need a program that takes it and its neighbourhood, and calculates a new pixel value. For this, the evolved program requires as many inputs as there are pixels in the neighbourhood, and a single output. In the evolved program, each function has two inputs and one output. These inputs are floating-point numbers that correspond to the grey-level values of the pixels. Figure 8.3 illustrates a program that takes a nine-pixel subimage, and computes a new pixel value.

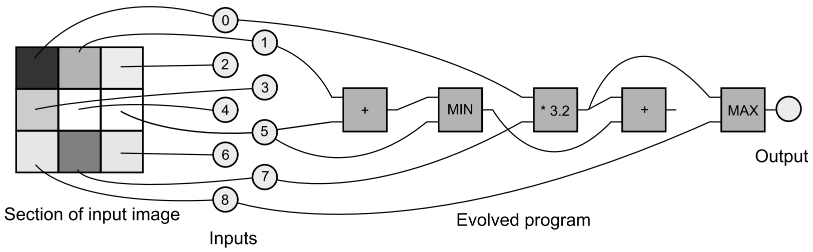

Fig. 8.3 In this example, the evolved program has nine inputs which correspond to a section of an image. The output of the program determines the new colour of the centre pixel. Note that one node has no connections to its output. This means that the node is redundant, and will not be used during the computation.

Mapping the image-filtering problem to the parallel architecture of the GPU is relatively straightforward.

It is important to appreciate that a GPU typically takes two arrays and produces a third by performing a parallel operation on them. The operation is elementwise, in the same way as for matrix operations. To clarify this, consider two arrays, $a = [1,2,3]$ and $b = [4,5,6]$. If we perform addition, we get $c = [5,6,9]$. With the SIMD architecture of the GPU, it is difficult to do an operation such as adding the first element of one array to the second of another. To do such an operation, the second array would need to be shifted to move the element in the second position to the first position.

For image filtering, we need to take a subimage from the main image, and use these pixels as inputs for a program (our convolution kernel), keeping in mind the vectorized operations of the GPU. To do this we take an image (e.g. the top left array in Fig. 8.4) and shift the array by one pixel in all eight possible directions. This produces a total of nine arrays (labelled (a) to (i) in Fig. 8.4).

Taking the same indexed element from each array will return the neighbourhood of a pixel. In Fig. 8.4, the neighbourhood is shaded grey and the dotted line indicates how these elements are aligned. The GPU runs many copies of the evolved program in parallel and, essentially, each program can can act on only one array index. By shifting the arrays in this way, we have lined up the data so that although each program can see only a given array position, by looking at the set of arrays (or, more specifically, a single index in each of the arrays in the set) it can have access to the given pixel and its neighbourhood. For example, if we add array (e) to array (i) the new value of the centre pixel will be 6, as the centre pixel in (e) has a value of 5 and the centre pixel in (i) has a value of 1.

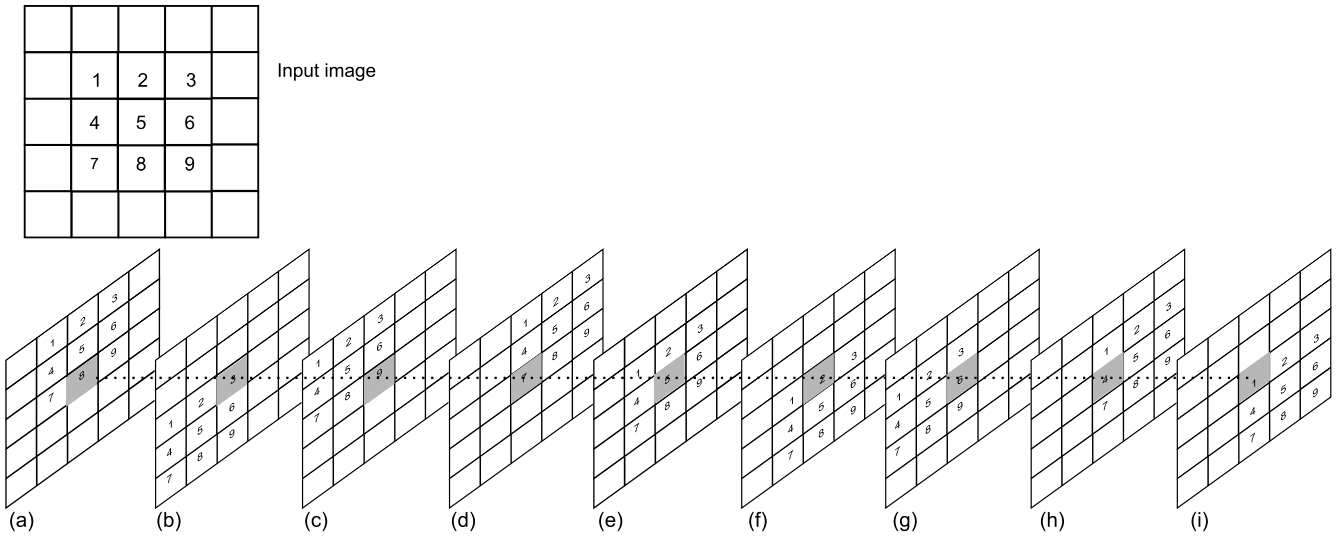

Fig. 8.4 Converting the input image to a set of shifted images allows the elementwise operations of the GPU to access a pixel and its neighbourhood. The evolved program treats each of these images as inputs. For example, should the evolved program want to sum the centre pixel and its top-left neighbour, it would add (e) to (i).

8.6.1 Evolving Image Filters Using Accelerator

In [5, 9], CGP was used with MS Accelerator to evolve a large number of different image filters. Chapter 6 contains more detail about the CGP component of this work, whereas here the focus is on the GPU side of the algorithm. In summary, the original input images (Fig. 8.5) were combined together to form a larger image. A filter was applied, using GIMP. We then used an evolutionary algorithm to find a mapping between the input and output images. The fitness function attempted to minimize the error between the desired output (the GIMP-processed image) and the output from the evolved filters.

GPU acceleration allows us to process images much more rapidly than on a CPU, and enables improvements to be made in the quality of the results collected. In [5] four different images were used for fitness evaluation, and none for validation. In [9], 16 different images (Fig. 8.5) were used; these were largely taken from the USC-SIPI image repository, with 12 used for fitness evaluation and four for validation. This allowed us to be confident that evolved filters would generalize well. As we employed a GPU for acceleration, we were able to test all the images at the same time and obtain both the fitness score and the validation score at the same time.

The fitness score of an individual was the average of the absolute difference (per pixel) between the target image and the image produced using CGP. The lower this error, the closer our evolved solution was to the desired output.

The fitness function was relatively complicated. The first step was to take the output from the evolved program, subtract it from our desired output and then find the absolute value of the result. The whole input image was processed at once (this input image contained 16 subimages). This provided an array containing the difference between the two images. Next, a mask was applied to remove the edges where the subimages meet. This was done because when the images were shifted, there would be overlapping data from different images that could introduce undesirable artefacts into the fitness computation. Edges were removed by multiplying the difference array by an array containing 0s and 1s. The 1s labelled where the fitness function would measure the difference.

To calculate the training error, the difference array was multiplied by another mask. This mask contained 1s for each pixel in the training images, and 0 for pixels that we did not wish to consider. The fitness value was the sum the contents of the array divided by the number of pixels.

The fitness function is illustrated in Fig. 8.6.

8.6.2 Results

Results for the quality of the image filters can be found in Chap. 6. Here, only the GPU acceleration is considered. Table 8.6 shows the performance of this implementation.

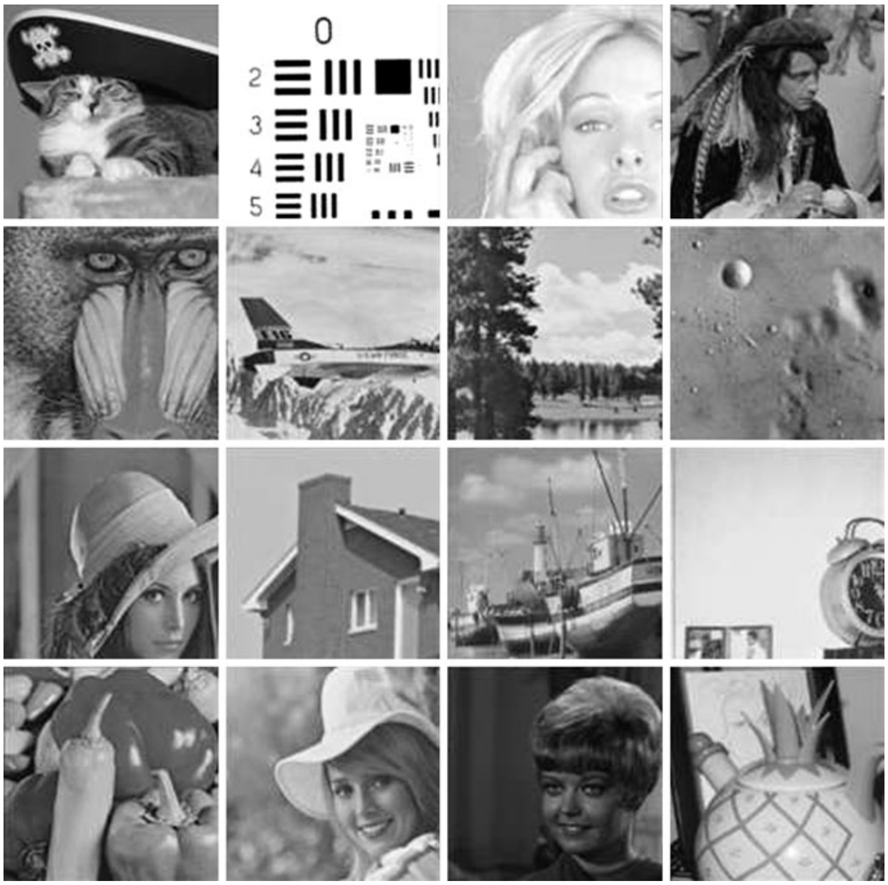

Fig. 8.5 The training and validation image set. All images were presented simultaneously to the GPU. The first column of images was used to compute the validation fitness, and the remaining 12 for the training fitness. Each image is 256 × 256 pixels, with the entire set of images containing 1024 × 1024 pixels.

It was found that using the GPU greatly decreased the evaluation time per individual. On our test system (Nvidia 8800 GTX, AMD Athlon 3500+, Microsoft Accelerator API), the average speed was approximately 145 million genetic programming operations per second (GPOps), and a peak performance of 324 Million GPOps was obtained. The processing rate was dependent on the length of the evolved programs. Some filters benefited more from the GPU implementation than others.

When the evolved programs were executed using the CPU, we obtained only 1.2 million GPOps, i.e. a factor of 100 times slower than for the GPU. However, using the reference driver incurs significant overhead and may not be an accurate reflection of the true speed of the CPU.

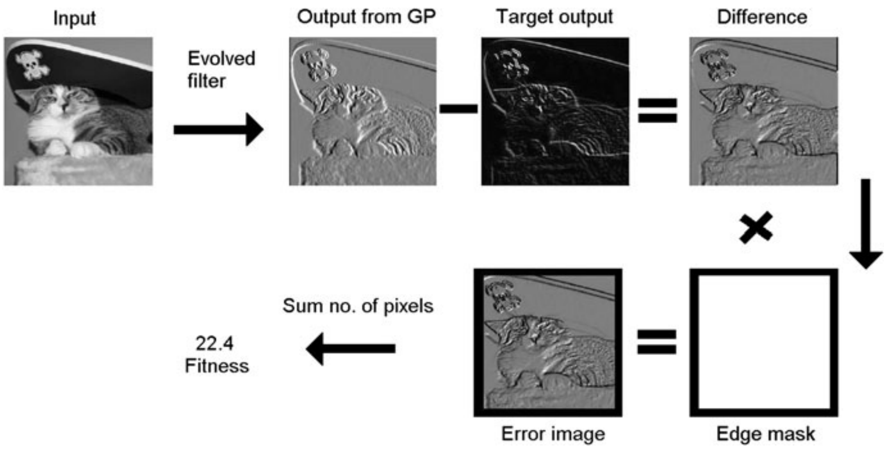

Fig. 8.6 Fitness calculation for images (simplified for a single image). The evolved program is run on the input image, and its per-pixel difference from the target image is calculated. This is then multiplied by an edge mask to remove artefacts created by the multiple subimages touching each other. Finally, the final error is calculated (the average pixel error).

Filter	Peak MGPOps	Avg MGPOps
Dilate	116	62
Dilate2	281	133
Emboss	254	129
Erode	230	79
Erode2	307	195
Motion	280	177
Neon	266	185
Sobel	292	194
Sobel2	280	166
Unsharp	324	139

Table 8.6 Maximum and average mega genetic programming operations per second (MGPOps) observed for each filter type

8.7 CUDA Implementation

The Accelerator implementations discussed so far suffer from many problems. Accelerator is very limiting (Windows only, bugs, too high-level), and does not appear to be under active development – although it is likely to resurface in future versions of DirectX.

The next implementation of CGP for GPUs that we describe is based on Nvidia's CUDA. However, doing this presented many challenges. To use a test-case-parallel approach, individual programs have to be compiled – and this introduces a significant time overhead. In [10], a new approach was introduced which aims to remove this overhead. This section discusses the most recent version of this approach.

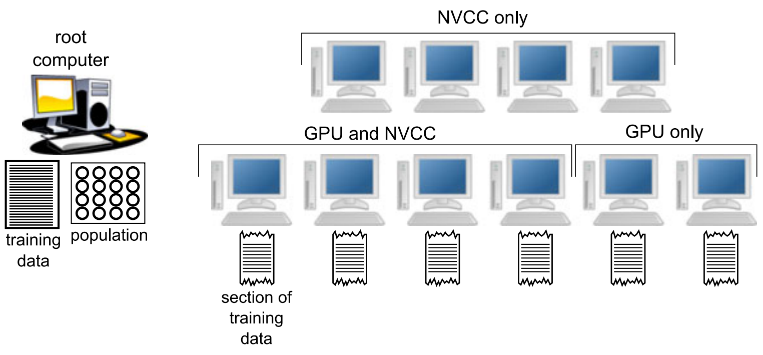

Fig. 8.7 Algorithm overview. The root computer administers the client computers. The root computer is responsible for maintaining the population and distributing sections of the data set to the GPU-equipped client computers. The client computers can be capable of executing programs (i.e. contain a compatible GPU) and/or be capable of compiling CUDA programs using NvCC. Different combinations of compiling and executing computers can be used, depending on the resources available.

8.7.1 Algorithm

The algorithm devised here is an implementation of a parallel evolutionary algorithm, where there is a master controller (called the 'root') and a number of slave computers (or 'clients'), as shown in Fig. 8.7. In summary, the algorithm works in the following manner:

1. A process is loaded onto all computers in the cluster.
2. The root process is started on the master computer.
3. Each computer announces its presence, and the root listens to these broadcasts and compiles a list of available machines.
4. After announcing their presence, each of the client machines starts a server process to listen for commands from the root.
5. After 2 minutes, the root stops listening for broadcasts and connects to each client machine.
6. Upon connecting to each machine, the root asks each client what its capabilities are (in terms of CUDA compiler and device capability).
7. The root distributes sections of the fitness set to each CUDA-capable computer.
8. The root initializes the first population.
9. The root begins the evolutionary loop.
   a. Convert population to CUDA C code.
   b. Parallel-compile population.
   c. Collate compiled code and resend to all compiling clients.
   d. Perform execution of individuals on all GPU-equipped clients.
   e. Collect (partial) fitness values for individuals from GPU-equipped clients.
   f. Generate next population, return to step 9a.

The algorithm here offloads both the compilation and the execution to a cluster of GPU-equipped machines. This significantly reduces the overhead and execution costs.

The cluster consisted of 28 computers, each with an Nvidia GeForce 8200 graphics card. This GPU is a very low end device. Each 8200 GPU has eight stream processors and access to 128 Mb of RAM. As the GPU was shared with the operating system, the free GPU memory on each machine was approximately 80 Mb. The root computer node used here was equipped with an Intel Q6700 quadcore, 4 Gb of RAM and Windows XP. The client computers used Gentoo Linux, AMD 4850e processors and 2 Gb of RAM. The software developed here uses a CUDA wrapper called Cuda.Net [3] and should work on any combination of Linux and Windows platforms.

8.7.2 Compilation and Code Generation

To mitigate the slow compilation step in the algorithm, a parallelized method was employed. Figure 8.8 shows how the compilation speed varies with the number of individuals in a population. The compilation time here includes rendering the genotypes to CUDA C code, saving the code and running the compiler. For a single machine, the compilation time increases quickly with the size of the population. For large populations, the compilation phase quickly becomes a significant bottleneck. Using distributed compilation it is possible to maintain a rapid compilation phase for typical population sizes.

CUDA allows library files (extension '.cubin') called 'modules' to be loaded into the GPU. Modules are libraries of functions that can be loaded, executed on the GPU and then unloaded. Module load times increase linearly with the number of functions (or individuals) they contain. However, there is a constant overhead to this, and for unloading the module, transferring the module across the network and other file-handling operations. Therefore it is preferable to minimize the number of module files that are created.

To compile a module, the user first creates a source file containing only CUDA functions. These should be declared as in the following example:

```
extern "C" __global__
    void Individual0(
         float* ffOutput,
         float* ffInput0,
         int width, int height)
```

When compiled, each of these functions is exposed as a function in the compiled library. Internally, the .cubin files are text files with a short header, followed by a set of code blocks representing the functions. The code blocks contain the binary code for the compiled function, in addition to information about memory and register usage.

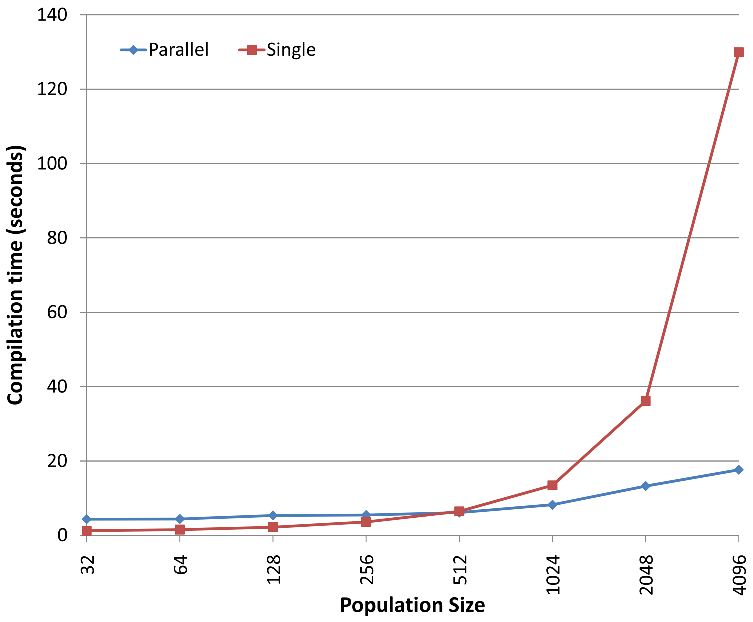

Fig. 8.8 How compilation speed varies with the number of individuals in a population, comparing single-machine compilation with compilation using 14 machines in the cluster.

It is possible to merge the contents of module files together by simply appending .cubin files together. This allows one to compile sections of a library independently of each other and then reduce these sections to a single module. However, only one header is permitted per .cubin file.

In this implementation, the following process was used to convert the genotypes into executable code.

1. Convert the population to CUDA. This process is performed on the root machine. At this stage, the population is divided up into sections, the number of which is equal to the number of machines available for compilation. Each section is then converted to CUDA using the code generator described in Sect. 8.7.2. This stage is performed in parallel across the cores on the root computer, where a separate thread is launched for each section of the population's code generator.
2. Send the uncompiled code to the distributed machines. In parallel, each section of uncompiled code is transmitted to a computer for compilation. The remote computer saves the code to the local disk, before compilation. The remote computer then loads the .cubin file from disk and returns it to the root computer.
3. The host computer merges the returned .cubin files together (in memory) and then sends the merged version back to all the computers in the grid for execu-

tion. Again, this happens in parallel, with the remote computers sending back their compiled files asynchronously.

Figure 8.8 shows the performance benefit obtained with this parallel compilation method. For typical population sizes, the parallel compilation means that each processor compiles only a small number of functions – and for such sizes the compiler is operating at its most efficient ratio of compilation time to function count. However, it should be noted that the compilation time does not appear to scale linearly with the number of compiling clients.

Tables 8.7 and 8.8 show the average times taken to compile populations of various sizes, with various program lengths. The results show a clear performance increase using the parallel compilation method.

Table 8.7 Time (in seconds) to perform the entire compilation process using 28 computers

Population size	Graph length			
	256	512	1024	2048
32	3.08	2.88	2.78	2.82
64	2.43	2.32	2.48	2.65
128	2.84	2.78	3.01	3.01
256	3.29	3.28	3.24	3.39
512	3.73	3.63	4.26	3.97
1024	4.96	5.89	5.41	6.35
2048	8.37	7.98	8.38	9.65

Table 8.8 Time (in seconds) to perform the entire compilation process using a single-threaded compilation method. Owing to memory allocation issues, some results are incomplete

Population size	Graph length			
	256	512	1024	2048
32	1.22	1.22	1.22	1.22
64	1.55	1.55	1.54	1.54
128	2.20	2.20	2.21	2.18
256	3.54	3.73	3.60	3.55
512	6.43	6.44	6.43	6.45
1024	13.33	13.56	13.43	13.45
2048	36.16	36.26	36.06	36.01

Code generation is the process of converting the genotypes in the population to CUDA-compatible code. During development, a number of issues were found that made this process more complicated than expected. One issue was that the CUDA compiler does not like long expressions. In initial tests, the evolved programs were written as a single expression. However, when the length of the expression was increased, the compilation time increased dramatically. This is presumably because the programs were difficult to optimize.

Another issue was that functions with many input variables can cause the compilation to fail, with the compiler complaining that it was unable to allocate sufficient registers. In the initial development, we passed all the inputs in the training set to each individual, regardless of whether the expression used them. This worked well for small numbers of inputs; however the training set that was used to test the system contained 41 columns. The solution to this problem was to pass the function only the inputs that it used. However, this requires each function to be executed with a different parameter configuration. Conveniently, the CUDA.Net interface does allow this, as function calls can be generated dynamically at run time. The other issue here is that all or many inputs may be needed to solve a problem. It is hoped that this is a compiler bug and that it will be resolved in future updates to CUDA.

```
extern "C" __global__ void Individual430(float* ffOutput,
    float* ffInput38, float* ffInput36, float* ffInput7,
    float* ffInput20, float* ffInput33, float* ffInput11,
    float* ffInput19, float* ffInput28)
{
    //set up indexes of where to read from
    unsigned int DataIndex = (blockIdx.x * blockDim.x) + threadIdx.x;
    //
    //Active nodes = 11
    float Temp788 = ((ffInput38[DataIndex]) +
        (ffInput36[DataIndex])) - (ffInput7[DataIndex]);
    float Temp1304 = ((-3395.2410) * ((ffInput20[DataIndex])
     * (ffInput33[DataIndex]))) * ((ffInput11[DataIndex]) *
     ((ffInput19[DataIndex]) - (ffInput28[DataIndex])));
    float Temp2196 = (Temp788) / ((ffInput38[DataIndex]) / (1175.2612));
    float r = ((1714.3846) + (Temp1304)) * (Temp2196);
    if (r<0)
        ffOutput[DataIndex] = 0;
    else
        ffOutput[DataIndex] = 1;
    //Operation Count = 11
    //Input Count = 8
}
```

Fig. 8.9 Example of the CUDA code generated for an individual.

The code in Fig. 8.9 illustrates a typical generated individual. As noted before, long expressions failed to compile within a reasonable time. The workaround used here was to limit the number of nodes that made up a single statement, and then use temporary variables to combine these smaller statements together. It is unclear what the best length is for statements, but it is likely to be a function of how many temporary variables (and hence registers) are used. The names of the temporary variables were related to the node index in the graph at which the statement length limit was introduced.

The final output of the individual was stored in a temporary variable (r). As this is a binary classification problem, it was convenient to add code here to threshold the actual output value to one of the two classes.

During code generation, it was also possible to remove redundant code. When a CGP graph is parsed recursively, unconnected nodes are automatically ignored and so do not need to be processed. It is also possible to detect code which serves no purpose, for example division where the numerator and denominator are the same, or multiplication by 0. This can be seen as another advantage of precompiling programs over simple interpreter-based solutions.

8.7.3 Fitness Function

The task here was to evolve an ‘emboss’ filter on a single image. A large image (3872 $\times$ 2592 pixels) was used. It should be noted that Accelerator could only cope with images of size 2048 $\times$ 2048 pixels.

The image was converted to grey scale, and then converted in a way similar to that described earlier, where each neighbourhood becomes a row in a data set. The desired output image was an ‘embossed’ version of the input. The data set contained 10,023,300 rows and 10 columns, which was approximately 400 Mb of data in memory. As each row was independent of every other row, the image could be split across the client computers in the same way as before.

The fitness was calculated as the sum of the differences between the image outputted by the evolved program and the target (embossed) image.

8.7.4 Results

Table 8.9 Average giga GP operations per second when the ‘emboss’ image filter was evolved using 28 computers

Population size	Graph length			
	256	512	1024	2048
32	3.3	3.6	4.5	5.4
64	5.2	6.8	7.3	8.4
128	7.2	8.4	9.9	13.8
256	11.4	13.2	15.3	18.6
512	14.2	16.6	18.3	21.8
1024	16.4	17.3	21.0	24.1
2048	16.4	18.3	21.6	26.2

Table 8.10 Peak giga GP operations per second when evolving the 'emboss' image filter was evolved using 28 computers

Population size	Graph length			
	256	512	1024	2048
32	7.2	7.5	8.9	10.0
64	8.0	9.9	13.8	16.2
128	12.3	15.7	17.9	22.5
256	19.5	21.7	26.2	29.7
512	22.1	24.5	28.5	32.6
1024	25.0	28.9	30.0	34.7
2048	24.1	27.3	32.2	34.2

Table 8.9 shows the average numbers of giga GP operations per second (GGPOps) and Table 8.10 shows the peak numbers of GGPOps for each of the population/graph size combinations used.

The results are considerably faster than those obtained using Accelerator on a single GPU, which had a peak of 0.25 GGPOps. The peak results here are over 100 times faster. The speed improvement is likely to be due to an increase in the number of shader processors available, although it is difficult to compare the results, given the differences in image size and fitness function.

8.8 Conclusions

GPUs are becoming increasingly common in all forms of scientific computing. For GP practitioners, we find that the nature of the problem of evaluating either individuals in parallel or test cases in parallel maps very well to the GPU architecture. As GPU technology improves, and particularly as the development tools improve, it is likely that more and more GP implementations will exploit this form of parallel processing.

More information about GPUs and genetic programming can be found at www.gpgpgpu.com.

8.9 Acknowledgements

WB and SH gratefully acknowledge funding from Atlantic Canada's HPC network ACENET, from the Canadian Foundation of Innovation, New Opportunities Grant No. 204503, and from NSERC under the Discovery Grant Program RGPIN 283304-07.

References

1. Banzhaf, W., Harding, S.L., Langdon, W.B., Wilson, G.: Accelerating Genetic Programming through Graphics Processing Units. In: R.L. Riolo, T. Soule, B. Worzel (eds.) Genetic Programming Theory and Practice VI, chap. 1, pp. 229–249. Springer (2008)
2. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: D. Thierens, H.G. Beyer, et al. (eds.) Proc. Genetic and Evolutionary Computation Conference, vol. 2, pp. 1566–1573. ACM Press (2007)
3. GASS Ltd.: CUDA.NET. http://www.gass-ltd.co.il/en/products/cuda.net/
4. Harding, S.L.: Genetic Programming on GPU Bibliography. http://www.gpgpgpu.com/
5. Harding, S.L.: Evolution of Image Filters on Graphics Processor Units Using Cartesian Genetic Programming. In: J. Wang (ed.) IEEE World Congress on Computational Intelligence, pp. 1921–1928. IEEE Press (2008)
6. Harding, S.L., Banzhaf, W.: Fast Genetic Programming and Artificial Developmental Systems on GPUs. In: International Symposium on High Performance Computing Systems and Applications, p. 2. IEEE Computer Society (2007)
7. Harding, S.L., Banzhaf, W.: Fast genetic programming on GPUs. In: Proc. European Conference on Genetic Programming, *LNCS*, vol. 4445, pp. 90–101. Springer (2007)
8. Harding, S.L., Banzhaf, W.: Genetic programming on GPUs for image processing. International Journal of High Performance Systems Architecture **1**(4), 231–240 (2008)
9. Harding, S.L., Banzhaf, W.: Genetic Programming on GPUs for Image Processing. In: J. Lanchares, F. Fernandez, J. Risco-Martin (eds.) Proc. International Workshop on Parallel and Bioinspired Algorithms, pp. 65–72. Complutense University of Madrid Press (2008)
10. Harding, S.L., Banzhaf, W.: Distributed Genetic Programming on GPUs using CUDA. In: I. Hidalgo, F. Fernandez, J. Lanchares (eds.) Proc. International Workshop on Parallel Architectures and Bioinspired Algorithms, pp. 1–10 (2009)
11. Koza, J.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT Press (1992)
12. Langdon, W.B., Banzhaf, W.: Repeated Sequences in Linear Genetic Programming Genomes. Complex Systems **15**(4), 285–306 (2005)
13. Langdon, W.B., Banzhaf, W.: A SIMD Interpreter for Genetic Programming on GPU Graphics Cards. In: Proc. European Conference on Genetic Programming, *LNCS*, vol. 4971, pp. 73–85. Springer (2008)
14. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Population Parallel GP on the G80 GPU. In: Proc. European Conference on Genetic Programming, *LNCS*, vol. 4971, pp. 98–109. Springer (2008)
15. Tarditi, D., Puri, S., Oglesby, J.: MSR-TR-2005-184 Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. Tech. rep., Microsoft Research (2006)
16. Wilson, G., Banzhaf, W.: Linear Genetic Programming GPGPU on Microsoft's Xbox 360. In: J. Wang (ed.) IEEE World Congress on Computational Intelligence. IEEE Press (2008)