Evolution of Image Filters on Graphics Processor Units Using Cartesian Genetic Programming

Simon Harding

Abstract— **Graphics processor units are fast, inexpensive parallel computing devices. Recently there has been great interest in harnessing this power for various types of scientific computation, including genetic programming. In previous work, we have shown that using the graphics processor provides dramatic speed improvements over a standard CPU in the context of fitness evaluation. In this work, we use Cartesian Genetic Programming to generate shader programs that implement image filter operations. Using the GPU, we can rapidly apply these programs to each pixel in an image and evaluate the performance of a given filter. We show that we can successfully evolve noise removal filters that produce better image quality than a standard median filter.**

I. Introduction

In recent months the first use of graphics processing units (GPUs) for genetic programming have appeared. Modern GPUs are extremely cost effective at performing parallel mathematical operations [1], and it is possible to exploit this for work with genetic programming.

Langdon's work [2] shows the possibility of using a single GPU to evaluate a population of many different individuals simultaneously. Work by Harding and Banzhaf[3], [4] and by Chitty[5] has shown that a single, low-end GPU is able to speed up the evaluation of a GP tree by 20 to 30 times.

These early results indicate that there is great utility in developing evolutionary algorithms that are able to exploit the parallelism of the GPU.

Until recently it was cumbersome to use this resource for general purpose computing. However, several new application programming interfaces (APIs) now exist that ease the programming challenge and allow for rapid development with minimal knowledge of the underlying hardware. These APIs have allowed many algorithms to be ported to GPU hardware. For a general survey on algorithms implemented on GPUs the reader is referred to [6]. For example, discrete wavelet transformations [7], the solution of dense linear systems [8], physics simulations for games, fluid simulators [9], have been shown to run faster on GPUs.

In this paper we demonstrate another application, that of evolving image filters. Using the GPU we are able to rapidly apply an evolved program to each pixel in an image, thereby increasing the evaluation speed on an individual. We demonstrate these image filters on standard noise removal problems, and we expect that the idea is also applicable to other image processing tasks such as segmentation and enhancement.

Simon Harding is with the Department Of Computer Science, Memorial University, Newfoundland, Canada, A1B 3X5 ; email: simonh@cs.mun.ca

II. Programming GPUs

Graphics processors are specialized stream processors used to render graphics. Typically, the GPU is able to perform graphics manipulations much faster than a general purpose CPU, as the graphics processor is specifically designed to handle certain primitive operations. Internally, the 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 the 3D view, then the shader processors paint this model before it is displayed. Programming the GPU is typically done through a virtual machine interface such as OpenGL or DirectX which provide a common interface to the diverse GPUs available thus making development easy. However, DirectX and OpenGL are optimized for graphics processing, hence other APIs are required to use the GPU as a general purpose device.

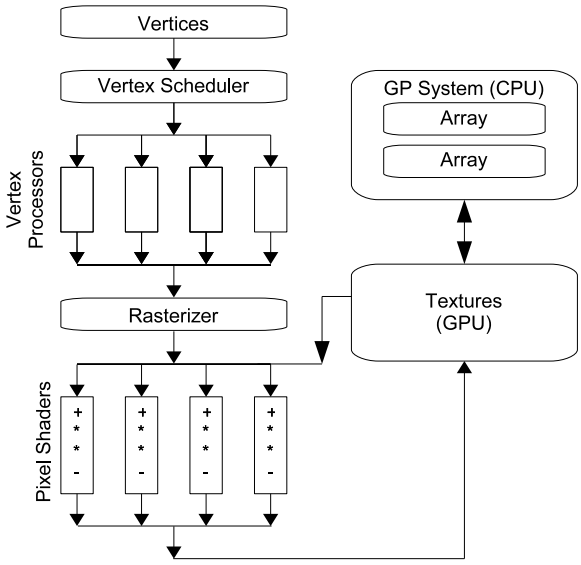

Fig. 1. Arrays, representing the test cases, are converted to textures. These textures are then manipulated (in parallel) by small programs inside each of the pixel shaders. The result is another texture, which can be converted back to a normal array for CPU based processing.

For general purpose computing, we here wish to make use of the parallelism provided by the shader processors, see Figure 1. Each processor can perform multiple floating

978-1-4244-1823-7/08/$25.00

point operations per clock cycle, meaning that performance is determined by the clock speed and 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, this may reduce 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.

We use C♯ and the Microsoft Accelerator package[10]. As with other GPU programming interfaces, such as RapidMind and Brook, Accelerator is based on arrays implemented as textures. The API allows one to perform parallel operations on the arrays, largely in matrix type operations. Conversion to textures, and transfer to the GPU is handled transparently by the API, allowing the developer to concentrate on the implementation of the algorithm. The available function set for operating on parallel arrays is similar to the other APIs. It includes element-wise arithmetic operations, 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 or maximum value in the array. Further functions perform transformations, such as shift and rotate on the elements of the array.

III. Evolution of Image Filters

We show the use of the GPU as a platform for the evolution of image 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, a common 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 it's neighbourhood. Examples of the behaviour of this filter are shown later in the results section as a basis for comparison with the evolved filters.

The evolution of noise-reducing image filters has been previously tackled[11], [12], including on parallel hardware architectures. For example, Vasicek and Sekanina used an FPGA based approach[13]. Here, Cartesian Genetic Programming (CGP, see section IV) was used to evolve the configuration for logic blocks inside the FPGA. This limited the functions to digital operations such as OR, AND, XOR and shifting. The entire algorithm was implemented on the FPGA and its associated PowerPC processor. They conclude that the FPGA evaluates individuals 22 times faster than a PC with a Celeron 2.4GHz CPU. Similary, Kumar et al. also evolved FPGA configurations [14] for noise removal, although in this case the exact performance in terms of speed up compared to a traditional CPU is unclear.

The FPGA based approaches are limited to binary operations, using a GPU we are able to work using floating point numbers - which makes direct comparison difficult. FPGA based approaches also suffer from the need for specialist hardware and software skills.

IV. Cartesian Genetic Programming

Cartesian Genetic Programming was originally developed by Miller and Thomson [15] for the purpose of evolving digital circuits and represents a program as a directed graph. One of the benefits of this type of representation is the implicit re-use of nodes in the directed graph. Originally CGP used a program topology defined by a rectangular grid of nodes with a user defined number of rows and columns. However, later work on CGP always chose the number of rows to be one, thus giving a one-dimensional topology, as used in this paper. In CGP, the genotype is a fixed-length representation and consists of a list of integers which encode the function and connections of each node in the directed graph.

CGP uses a genotype-phenotype mapping that does not require all of the nodes to be connected to each other, resulting in a bounded variable length phenotype. This allows areas of the genotype to be inactive and have no influence on the phenotype, leading to a neutral effect on genotype fitness called neutrality. This unique type of neutrality has been investigated in detail [15], [16], [17] and found to be extremely beneficial to the evolutionary process on the problems studied.

Each node in the directed graph represents a particular function and is encoded by a number of genes. The first gene encodes the function the node is representing, and the remaining genes encode the location where the node takes its inputs from, plus one parameter that is used as a constant. Hence each node is specified by 4 genes. The genes that specify the connections do so in a relative form, where the gene specifies how many nodes back to connect. If this address is negative, a node connects to an input. Modulo arithmetic is used to handle conditions where the index goes beyond the number of inputs.

The graph is executed by recursion, starting from the output nodes down through the functions, to the input nodes. In this way, nodes that are unconnected are not processed and do not effect the behavior of the graph at that stage. For efficiency, nodes are only evaluated once with the result cached, even if they are connected to multiple times.

To clarify, figure 2 shows an example CGP program. The genotype for such a graph would be:

```
ADD 2 6 4.35
MIN 1 7 2.3
MULT 3 8 3.2
ADD 1 2 -54
MAX 2 13 1.23
```

A. GPU Implementation

Running the filters on the GPU will allow us to apply the kernel to every pixel (logically, but not physically) simultane-

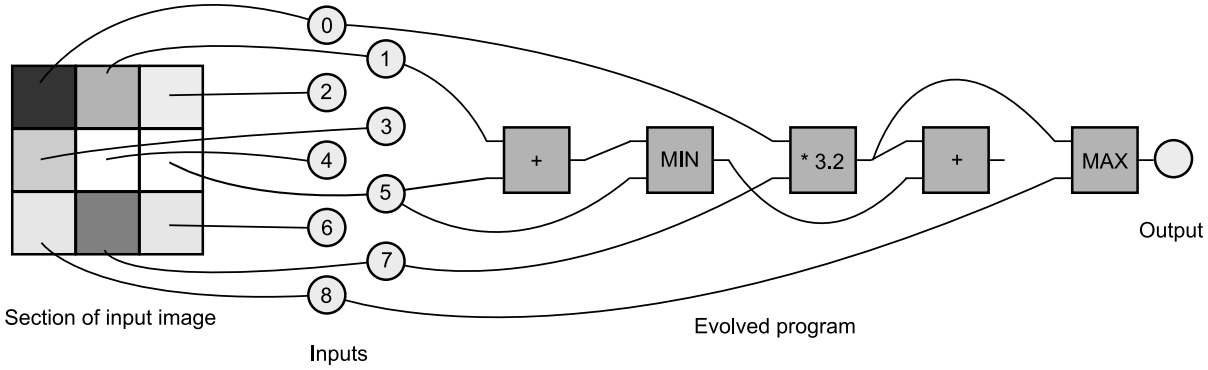

Fig. 2. In this example, the evolved program has 9 inputs - that correspond to a section of an image. The output of the program determines the new colour of the centre pixel. Note that one node has no connections to it's output. This means the node is redundant, and will not be used during the computation.

ously. The parallel nature of the GPU will allow for multiple kernels to be calculated at the same time. This number will be dependent on the number of shader processors available. Using the Microsoft Accelerator architecture, it will appear to be completely parallel. Although internally, the task will be broken down into chunks suitable for the GPU.

The image filter is made of an evolved program that takes a pixel and it's neighbourhood (a total of 9 pixels) and compute the new value of that 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 it's 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 2 illustrates a program that takes a 9 pixel sub image, and computes a new pixel value.

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

It is important to appreciate that the GPU typically takes 2 arrays and produces a 3rd by performing a parallel operation on them. The operation is element-wise, in the same way as matrix operations. To clarify, consider 2 arrays: $a = [1, 2, 3]\, b = [4, 5, 6]$. 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 add the first element of one array to the second of other. To do such an operation, the second array would need to be shifted to move the element in the second position to the first.

For the image filtering, we need to take a sub image from the main image, and use these pixels as inputs for a program (our convolution kernel) - keeping in mind the matrix like operations of the GPU.

To do this we take an image (e.g. the top left array in figure 3) and shift the array one pixel in all 8 possible directions. This produces a total of 9 arrays (labeled (a) to (i) in figure 3).

Taking the same indexed element from each array will return the neighbourhood of a pixel. In figure 3, the neighbourhood is shaded grey and a dotted line indicates how these elements are aligned. The GPU runs many copies of the evolved program in parallel, and essentially each program can only can act on one array index. By shifting the arrays in this way, we have lined up the data so that although each program can only see 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 a given pixel and it's neighbourhood.

These arrays become the inputs to our evolved program, so each program has access to a pixel and it's 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 value 5 and the centre pixel in i has value 1.

It is important to note that the evolutionary algorithm itself remains on the CPU, and only the fitness function is run on the GPU.

V. Experiments

A. Noise Removal

We chose two different types of noise to demonstrate both the ability of CGP as a genetic programming algorithm, but also to demonstrate the GPU platform. We use both salt and pepper and random valued noise.

Salt and pepper noise is the addition of random black or white pixels to the image. In these experiment we change a pixel to either black or white with a probability of 0.05.

For the random valued noise, we set the value of a pixel to a random grey value with a probability of 0.05.

As the GPU greatly increases the speed of evaluation (see section VI-B), we are able to test a number of images in parallel. We use a number of input images of 256x256 pixels,

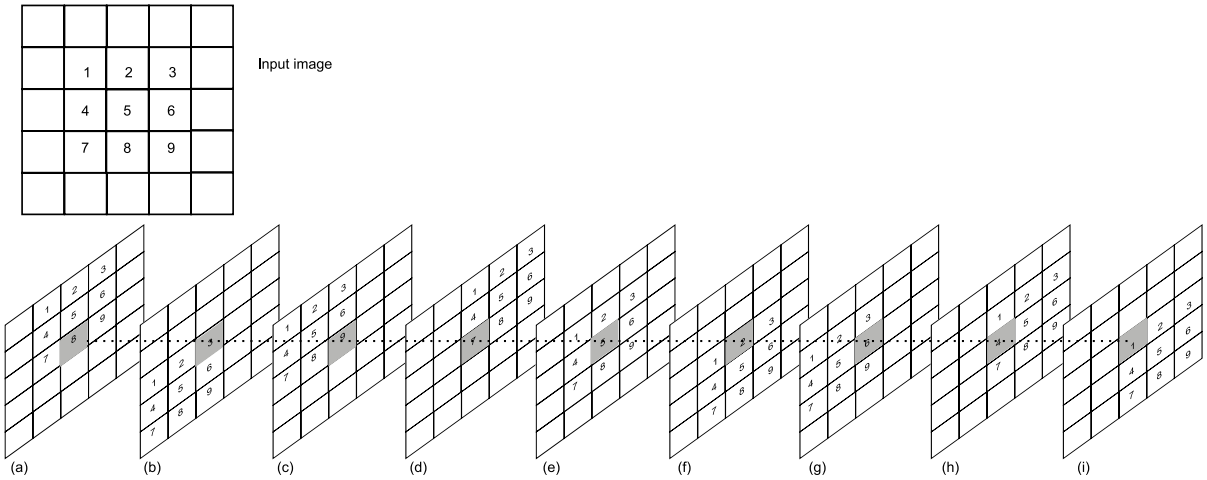

Fig. 3. Converting the input image to a set of shifted images allows the element wise operations of the GPU access a pixel and it's neighbourhood. The evolved program treats each of these images as inputs. For example, should the evolved program want to sum the centre pixel and it's top-left neighbour, it would add (e) to (i).

and tile them to form a larger image of either 512x512 or 1024x1024 pixels. In effect, this allows us to test 4 or 16 test cases at the same time. We compare the difference in behaviour of evolvability and quality of solution depending on the number of different test cases.

For the fitness function, we take a noisy image and then apply the evolved filter. The fitness score is the average error on each pixel. Where the error is the absolute difference between a pixel in the original image and the pixel. Hence, a lower fitness score is better. For large images, it is easy for a measure, such as the sum squared error, to exceed the largest value that can be stored in a float. We use a scaling of 0 to 255 to represent the pixel grey values.

As there are multiple images in the data, we have to be careful of the edges where one image touches another. For the fitness function here, we ignore an area 4 pixels wide around the edge of each image. We do this by generating a mask, which we apply to the image before computing the fitness. The mask consists of a white image, with black pixels representing pixels to ignore.

In summary the fitness function operates in the following manner:

1) Run evolved program on image to compute the output image, O.
2) Find the difference between O and the target image T. i.e. $\|O - T\|$
3) Apply the mask, M, so that edge pixels will be ignored $O = min(O, T)$
4) Divide O by the number of pixels in the image
5) Sum the pixels in O to find the average error, and return as the fitness.

Each experiment was repeated 25 times.

B. Genetic Algorithm and Parameters

The algorithm used here is a simple evolutionary algorithm. We have a population of size 50. The mutation rate is set to be 5%, i.e. each gene in the genotype will be mutated with probablity 0.05. We do not use crossover.

Selection is done using a tournament selection of size 5. The 5 best individuals are promoted to the next generation without modification. The CGP graph is intialised to contain 50 nodes (it is important to note that not all nodes will be used in the generated program).

Evolution was allowed to run for 50,000 evaluations.

Table I shows the available functions.

VI. Results

We were able to successfully evolve noise filters for both types of noise. The third column in Figure 4 shows quality of the filter when removing random grey values. Similarly, the third column in Figure 5 demonstrates the successful removal of salt and pepper noise from an image. Both these examples were typical of the evolved filters.

A. Evolution Results

Table II shows the achieved fitness results at the end of evolution. The use of more test images gave a slightly better average fitness (and hence better image quality) than when using only 4 images in the fitness evaluation. Surprisingly, this difference is not statistically significant. This indicates that, at least for this problem, there is no significant advantage in presenting more test cases. We were unable to try larger number of test images due to memory limitations on the test hardware.

The images in figures 4 and 5 show the performance of the best evolved individuals. By comparing the output image (third column) to the original image (first column), we can see that the noise is removed and that the image

Function	Description
ADD	Add the two inputs
SUB	Subtract the second input from the first
MULT	Multiply the two inputs
DIV	Divide the first input by the second
ADD CONST	Adds a constant (the node's parameter) to the first input
MULT CONST	Multiplies the first input by a constant (the node's parameter)
SUB CONST	Subtracts a constant (the node's parameter) to the first input
DIV CONST	Divides the first input by a constant (the node's parameter)
SQRT	Returns the square root of the first input
POW	Raises the first input to the power of the second input
COS	Returns the cosine of the first input
SIN	Returns the sin of the first input
NOP	No operation - returns the first input
CONST	Returns a constant (the node's parameter)
ABS	Returns the absolute value of the first input
MIN	Returns the smaller of the two inputs
MAX	Returns the larger of the two inputs
CEILING	Rounds up the first input
FLOOR	Rounds down the first input
FRACTION	Returns the fractional part of dividing the first input by the second
LOG2	Log (base 2) of the first input
RECIPRICAL	Returns $1/firstinput$
RSQRT	Returns $1/\sqrt{firstinput}$

TABLE I
CGP FUNCTION SET

quality is retained. The fourth column shows the application of the median filter (using the Paint.net software). It shows significant degradation in image quality compared to the evolved image.

The grey value of each pixel in the input image (second column in 4) for the noise removal problem has an average error of 3.77 compared to the original image. The average error on the median filtered image is 3.54. Our best evolved solution has an average of 0.53, which is significantly better. The image retains its sharpness, and features are preserved - compare the top-left hand section of the cat image for a marked difference.

We can extract the evolved program to see how it works. For the noise filter, we get the following expression:

$OUTPUT = MAX(MIN(P_2, MAX(P_5, P_8, P_3 \times 1.202, P_1)), MIN(FLOOR(MAX(P_5, P_8)), -P_2) \times -0.280)$

Where P_N refers to a pixel from the sub-image. P_5 would be the centre pixel, P_0 the top left and P_8 the bottom right. It is hard to see how the function operates. However, we can make some observations. First the program is relatively short, and uses only a small number of functions from the available function set. Secondly, we can see that not all the pixels are used. In total this program used 17 nodes out of the 50 nodes available in the genotype. However, as the values from many nodes were reused, the expression can be simplified. This also has implications for API that converts the evolved program into a shader program to run on the GPU. Previous experience suggests that short programs do not benefit from execution on the GPU because of the overhead of compilation and transfer of data[3].

We can perform the same analysis for the salt and pepper noise removal problem. Here, the noisey image has an average error of 6.30. Apply the median filter reduces this error to 3.62. However, from the fourth column in figure 5 shows that the median filter removes the noise at the expense of softening the image and reducing the overall visual quality. In comparison, the evolved filter removes the noise and keeps a good amount of detail. Again, this is most evident in the top-left hand corner of the cat image.

The evolved function used 14 nodes, with the output of several nodes reused. The entire function is:

$OUTPUT = MAX(MIN(P_2,$
$ADD(RSQRT(MIN(MAX(P_6, P_1, P_5), P_8)),$
$SUB(MAX(MAX(P_6, P_1), IN5), SUB(P_2 + 67.9, P_8)))),$
$MIN(MIN(MAX(P_6, P_1, P_5), P_8),$
$ADD(RECP(P_2), P_2 + 67.9)))$

Again, the function is complicated and it is difficult to see how it functions. Similar to the previous example, this expression uses only a small number of the available functions.

Table III shows the number of evaluations required to reach the best fitness. We see that in terms of average

Fig. 4. Results of evolution for the noise problem. Each tile shows a crop (128x128 pixels) of the input images. The first column of images shows the original image. The second column shows the image with added noise. The third column shows the result of noise removal using an evolved filter. The final column shows the effect of apply a median filter to the noisey image.

Noise Type	No. of test images	Average Fitness	Best Fitness
Random	4	1.30	0.55
Random	16	1.06	0.53
Salt and Pepper	4	0.76	0.39
Salt and Pepper	16	1.04	0.30

TABLE II

BEST FITNESS ACHIEVED FOR EACH IMAGE TYPE AND SIZE. FITNESS IS THE AVERAGE ERROR PER PIXEL OF THE OUTPUT IMAGE, COMPARED TO THE TARGET IMAGE.

Noise Type	No. of test images	Avg. evals.	Min evals.
Random	4	42539	14712
Random	16	43296	252
Salt and Pepper	4	42852	8732
Salt and Pepper	16	39795	19403

TABLE III

NUMBER OF EVALUATIONS REQUIRED TO REACH THE BEST FITNESS. THE HIGH NUMBER OF EVALUATIONS RELATIVE TO THE TOTAL ALLOWED EVALUATIONS SUGGESTS THAT EVOLUTION WAS STILL ONGOING, AND THAT THE OPTIMUM SOLUTION HAD YET TO BE REACHED.

number of evaluations, that the number of test cases did not have an impact. However when considering the noise problem, for the larger number of test cases we found that 1 solution converged to a rather disappointing 2.66 after only 252 evaluations. This result is an outlier, and removing this we find that the minimum number of evaluations is is 20,996. Which is very similar to the smaller number of test cases.

The high number of evaluations relative to the total allowed evaluations suggests that the filters had not yet finished evolving. However, the results are still of good quality.

B. Performance Results

We are able to find the number of floating point operations per second (FLOPS) by collecting some statistics from the execution of the evolved program. We assume that each API call on Microsoft Accelerator generates one shader operation - but this may not be the case, and may explain why the calculated results are not as expected. The average FLOPS on our test machine (nVidia GeForce 7300, Intel 6400, Windows XP) to be 300 MFLOPS when processing a 512 by 512 pixel image, and 620 MFLOPS for a 1024 by 1024 pixel image. This is a lot less than we expect even from such a low end card. This suggests that the MS Accelerator API is

Fig. 5. Results of evolution for the salt and pepper noise removal problem,. Each tile shows a crop (128x128 pixels) of the input images. The first column of images shows the original image. The second column shows the image with added noise. The third column shows the result of noise removal using an evolved filter. The final column shows the effect of apply a median filter to the noisey image.

not particularly efficient, or that our method of calculating the FLOPS is not calculating the true speed but rather a 'MS Accelerator Operations Per Second'.

The performance gain when moving up to the larger image size also demonstrates there is a large overhead from compilation to the shader program. However, there is no clear way to time this process. Unfortunately Microsoft has removed the CPU implementation from their API, so we were not able to benchmark directly against the CPU. They do provide a CPU-bound reference driver as part of the DirectX SDK, which achieved an average of 1.82 MLOPS for the larger image. However, it is unlikely that this is a true indication of the speed of the CPU.

For future work we will most likely move to another API.

VII. Conclusions

This paper has demonstrated two things. The first is that is possible to evolve image filters (based on floating point arithmetic) using Cartesian Genetic Programming. We believe this is the first demonstration of CGP on this problem. The results, as evidenced by the sample images, show that the technique works well.

Secondly, we demonstrate that it is possible to implement such systems on programmable graphics hardware. Again, we understand this is the first example of evolved image filtering on a GPU. GPU based image filtering is in relatively common usage, and we have shown that it is possible to rapidly evolve filters that can perform better than a more traditional filter (i.e. median). In future we would like to investigate the ability to evolve for speed in addition to quality. We are confident we can produce both faster and higher quality filters than conventional design.

The GPU appears to be an efficient platform for implementing such evolvable systems. However, we are concerned that the MS Accelerator API is not as efficient as it could be. We will therefore likely move to other platforms, such as RapidMind, in the future.

Acknowledgments

We would like to thank Julian Miller, Peter Duerr and Julien Hubert for proof reading this paper and providing early feedback.

REFERENCES

[1] C. Thompson, S. Hahn, and M. Oskin, "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis," in *Proceedings of the 35th International Symposium on Microarchitecture, Istanbul*. IEEE Computer Society Press, 2002, pp. 306 – 317.

[2] W. B. Langdon, "A SIMD interpreter for genetic programming on GPU graphics cards," Department of Computer Science, University of Essex, Colchester, UK, Tech. Rep. CSM-470, 3 July 2007.

[3] S. Harding and W. Banzhaf, "Fast genetic programming on GPUs," in *Proc. 10th Europ. Conference on Genetic Programming, Valencia, Spain*, ser. LNCS, M. Ebner, M. O'Neill, A. Ekart, L. Vanneschi, and A. Esparcia-Alcazar, Eds., vol. 4445. Springer, April 2007, pp. 90 – 101.

[4] ——, "Fast genetic programming and artificial developmental systems on GPUs," in *HPCS '07: Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications*. IEEE Computer Society, 2007, p. 2.

[5] D. M. Chitty, "A data parallel approach to genetic programming using programmable graphics hardware," in *GECCO 07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1566–1573.

[6] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," *Eurographics 2005, State of the Art Reports*, pp. 21–51, 2005.

[7] J. Wang, P. A. H. T. T. Wong, and C. S. Leung, "Discrete wavelet transform on GPUs," in *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors*, 2004, pp. C–41.

[8] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware," *Proceedings of the ACM/IEEE Supercomputing 2005 Conference*, pp. 3– 3, 2005.

[9] T. R. Hagen, J. M. Hjelmervik, K.-A. Lie, J. R. Natvig, and M. O. Henriksen, "Visual simulation of shallow-water waves," *Simulation Modelling Practice and Theory*, vol. 13, pp. 716–726, 2005.

[10] D. Tarditi, S. Puri, and J. Oglesby, "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*. ACM, 2006, pp. 325–335.

[11] S. L. Smith, S. Leggett, and A. M. Tyrrell, "An implicit context representation for evolving image processing filters," in *Applications of Evolutionary Computing, EvoWorkshops2005*, ser. LNCS, F. Rothlauf, J. Branke, S. Cagnoni, D. W. Corne, R. Drechsler, Y. Jin, P. Machado, E. Marchiori, J. Romero, G. D. Smith, and G. Squillero, Eds., vol. 3449. Lausanne, Switzerland: Springer Verlag, 30 Mar.-1 Apr. 2005, pp. 407–416.

[12] K. Slan and L. Sekanina, "Fitness landscape analysis and image filter evolution using functional-level CGP," *Lecture Notes in Computer Science*, vol. 2007, no. 4445, pp. 311–320, 2007.

[13] Z. Vacek and L. Sekanina, "Evaluation of a new platform for image filter evolution," in *Proc. of the 2007 NASA/ESA Conference on Adaptive Hardware and Systems*. IEEE Computer Society, 2007, pp. 577–584.

[14] P. N. Kumar, S. Suresh, and J. R. P. Perinbam, "Digital image filter design using evolvable hardware," in *ICIS '05: Proceedings of the Fourth Annual ACIS International Conference on Computer and Information Science (ICIS'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 483–488.

[15] J. F. Miller and P. Thomson, "Cartesian Genetic Programming," in *Proceedings of EuroGP 2000*, ser. LNCS, R. Poli, W. Banzhaf, and et al., Eds., vol. 1802. Springer-Verlag, 2000, pp. 121–132.

[16] V. K. Vassilev and J. F. Miller, "The advantages of landscape neutrality in digital circuit evolution," in *Proc. of ICES*. Springer-Verlag, 2000, vol. 1801, pp. 252–263.

[17] T. Yu and J. Miller, "Neutrality and the evolvability of boolean function landscape," in *Proc. of EuroGP 2001*, ser. LNCS, J. F. Miller and M. Tomassini, Eds., vol. 2038. Springer-Verlag, 2001, pp. 204–217.