Paper Title:The Application of Genetic Programming to General Purpose GPU Computation

Author: Eamon Phelan
Student No: 06165699
Course: 4033 Natural Computing

Paper Abstract: The recent advances in GPU hardware have resulted in the wide availability of programmable commodity hardware suited to massively parallel computation. This paper examines the application of the evolutionary computation approach of Genetic Programming to explore the space of general purpose GPU programs.

1. Introduction

The area of general purpose GPU programming has developed alongside GPU hardware into a field of serious research interest. This paper examines the application of genetic programming to the the problem of developing solutions to problems on this non-traditional platform. The paper is divided into 6 sections:

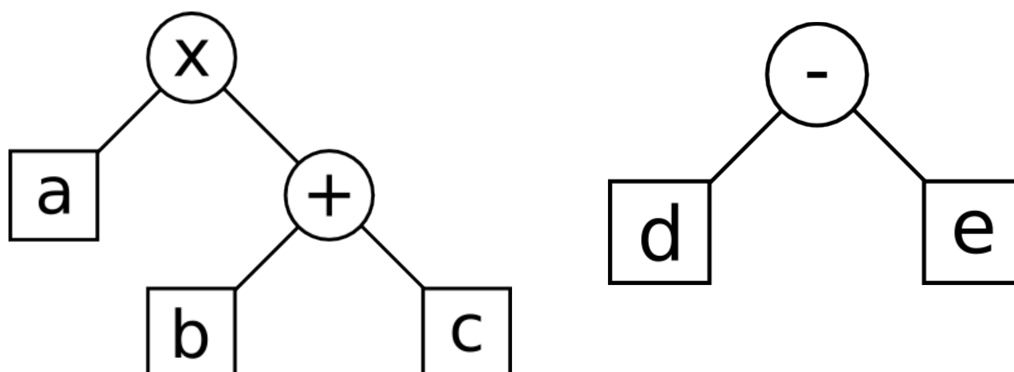
- Introduction where the background subjects of Genetic Programming and GPGPU programming are discussed.
- Related work where previous applications of natural computing to both the general field of graphics and to GPUs specifically is discussed.
- Implemented approach outlines the genetic programming system developed.
- Results presents the results obtained from the approach.
- Discussion outlines possible future applications of genetic programming to GPGPU programming: specific developments of the system presented as well as more general possibilities.
- Conclusion.

1.1 Genetic Programming

Genetic programming is a natural computing approach developed by John Koza [1]. It differs from many earlier approaches in that it has no phenotype/genotype distinction. In the traditional GA for example individuals are represented at the genotype level by a bit vector [2]. The genetic operations of crossover and mutation are performed on this bit vector. The fitness of the individual is determined by interpreting the values of the bit vector in the context of the problem domain – effectively the analogue of the biological phenotype mapping from DNA. In Genetic Programming however the individual is the implementation of the solution in a programming language. The fitness is determined by simply evaluating the individual and rating its suitability.

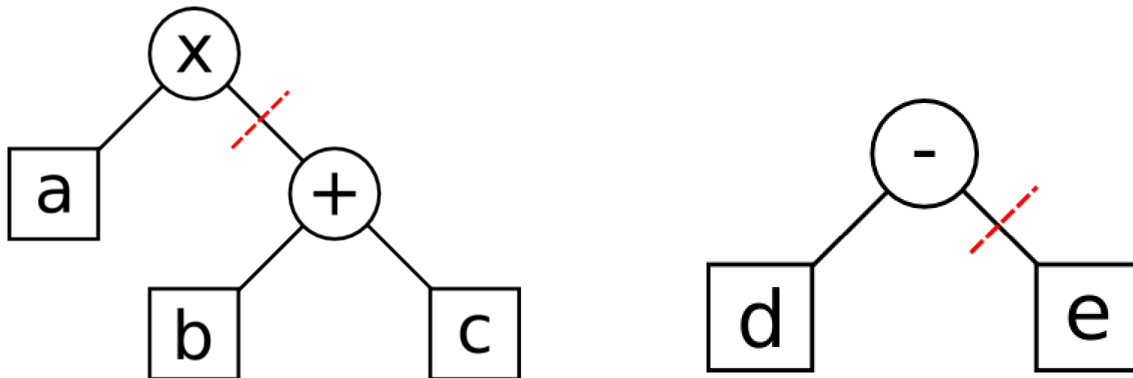
The original genetic programming method produced lisp s-expressions. The operations of crossover and mutation were implemented directly on the s-expressions (effectively a parse tree) however the approach is easily modified for C style procedural languages and the language used in this paper is a graphically-oriented stream language based on C called CG [3].

An example of two lisp s-expressions is shown below. The two expressions correspond to the arithmetic expressions (a * (b + c)) and (d – e).

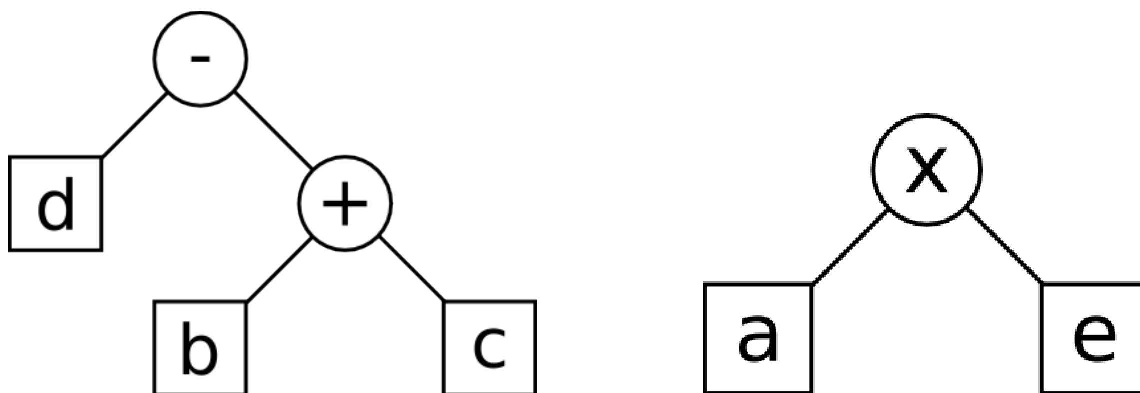


Crossover is implemented directly on these trees as follows: A crossover point is chosen in each parent, (illustrated in the diagram below by a red line), the subtrees below these crossover points are then swapped producing two offspring trees. The resulting trees are shown below.

Parents:



Offspring:



1.2 GPGPU programming

GPUs (Graphics Processing Units) originated in response to the need for more specialised hardware for graphical tasks. The more general purpose PC CPU was unsuitable for the more parallel operations of rendering such as matrix transforms and texturing. Originally the hardware pipeline was fixed function but as GPUs became more capable they became more flexible. This resulted in the creation of GPU shader programs. Originally only assembly programs with limited length and no control flow they have developed into high level C-like languages of unrestricted length with a large built in function set to draw on.

Two types of shader programs exist on most of today's programmable hardware[3]: vertex shaders and fragment(or pixel) shaders corresponding to two different stages in the graphics pipeline. Vertex shaders allow the programmer to specify operations to occur when the geometry is being processed while fragment shaders allow the programmer to specify operations to occur when the actual pixel information is being generated to be displayed. Fragment shaders are the more flexible of the two and this paper focuses on their implementation.

GPUs have massively increased in capability since they were introduced(they developed at a much faster rate than Moore's law would predict). The ever-increasing demands of graphical applications have led to them becoming capable of massively parallel computation, the NVidia GeForce 8800 for instance contains 128 stream processors which work in parallel [4]. This processing capability is ideally suited to certain algorithms, particularly tasks which require matrix operations which the GPUs have implemented directly in hardware. Tasks as varied as fractal image compression [5] and fast sorting [6].

2. Related Work

Previous work in the area of natural computing and programmable GPUs has focused either on the graphical capabilities of the cards or the ability of the cards to basically be a math co-processor for speeding up the evolutionary algorithms.

2.1 Natural Computing and Graphics

Natural computing has long had a connection with graphics, Dawkins' BioMorphs can be seen as an exercise in evolving a specific type of image for instance. More specifically techniques have been developed which use genetic algorithms to explore the search of possible 2D textures with fitness interactively determined by a user as well as automatically determined relative to the desired properties of the final image (similarity to a training texture for instance)[7,8]. A method to evolve vertex and pixel shaders using this interactive fitness determination method has also been developed by Ebner et al [9]. Meyer-Spradow and Loviscach developed a GPU based method to evolve the BRDFs of materials[10] (the BRDF or bidirectional reflectance distribution function is a high dimensional function which describes the reflectance of surfaces in computer graphics), again interactive.

2.2 Natural Computing and GPU processing

Wong et al have implemented two seperate parallel genetic algorithms on GPU hardware reporting a significant speedup in processing time.[11] Their specific aim was to use the GPU as a fast parallel processor for genetic algorithms.

3. Implemented Approach

The implementation of genetic programming on the GPU has two main components, the aspects of the implementation dealing with purely genetic programming were implemented in Java using the ECJ toolkit [12]. The implementation of the GPGPU specific component was completed in C++. A simple GP problem was used to test the system.

3.1 Genetic Programming Component

The ECJ toolkit was used to produce code trees and to implement the evolutionary aspects of the implementation. It used tree based genetic programming and generated trees with nodes corresponding to a subset of the NVidia CG language. When the trees were evaluated the tree was traversed and converted into standard C-style notation. The generated program was then evaluated

for fitness using the C++ GPGPU component. The results of this evaluation were then used to determine the fitness of the individual for selection purposes by the ECJ component.

3.2 GPGPU Component

The GPGPU component took as input the generated CG code segment, it then combined this with a header and footer and executed it on the test data provided by the genetic programming component.

3.3 GP Problem

The quartic polynomial symbolic regression genetic programming problem described by Koza[1] was used to test the implementation. The genetic programming component used the default GP settings of ECJ to test both the accuracy and the performance of the system. The terminal set was defined to be {x} and the function set was {*,+,-}. The terminal and function set was kept deliberately simple to both give a clearer impression of the performance of the system and so that code generated would be portable between CG and standard C.

4. Results

A pre-generated test data set was created which was then used to compare both the pure CPU based genetic programming and GPGPU genetic programming. Two tests were carried out, the first was for accuracy where the results of the CPU and GPGPU evaluation step were compared.

There were no significant differences noticed in the accuracy of the GPGPU approach.

The second was a performance test where the evaluation phase of the GPGPU GP process was repeated 10000 times with a large function taken from the population set generated by ECJ on data sets of various sizes. The results are presented in the table below.

Input Size	***Seconds to Execute***
16	2.094
32	1.859
64	2.047
128	1.86
256	1.875
512	1.719
1024	1.75
4096	1.906
16384	2.359
65536	3.907
262144	8.344
1048576	26.469

5. Discussion

Two possible areas of utility for GPGPU genetic programming exist. The first is as a fast parallel processor to speed the execution of genetic programming runs and the second is as a way to explore the specific search space of GPGPU algorithms for an efficient and novel solution.

5.1 Fast parallel genetic programming

GPUs have already been used to significantly speed the execution of genetic algorithms [11]. If a mapping from the representation used by the genetic programming system to CG exists then a GPU could be used as a fast parallel processor to speed up the search. It may also be possible with unlimited length shaders to build a very limited interpreter into the shader itself to execute the representation. From the results of the experiments undertaken the main difficulty seems to be keeping the GPU fed with programs to evaluate if the programs are short. A possible solution to this would be to have a server containing a GPU based evaluation core and evolutionary islands on standard machines which feed individuals to the server to evaluate.

5.2 Exploring the Search Space of GPU Programs

While GPGPU programming has become more accessible with the advent of high-level shader programming languages the nature of the hardware and its original purpose means that the operations exposed are often unsuited for the implementation of algorithms originally designed for general purpose CPUs. The possible uses of the many linear algebra and graphics originated GPGPU capabilities for standard programming is non-intuitive. Genetic programming could be used to attempt to find suitable GPU solutions so that the capabilities of the hardware could be leveraged.

6 Conclusion

GPU hardware is a resource that is underutilised, while it developed due to the demand for greater graphics capabilities in areas such as gaming its capabilities have developed to the point where they can be used as general purpose computation devices. The challenge lies in mapping the algorithm to the abilities of the hardware. Genetic Programming as presented in this paper may be able to aid in this task.

6 References

[1] J.R. Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, 1992.

[2] D.E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison Wesley, 1989.

[3] Website, CG Toolkit: http://developer.nvidia.com/object/cg_toolkit.html .

[4] Website, NVidia 8800 hardware FAQ: http://www.nvidia.com/page/8800_faq.html

[5] U. Erra. Toward Real Time Fractal Image Compression Using Graphics Hardware. Lecture Notes in Computer Science, Vol. 3804, pp. 723-728, Springer-Verlag, 2005.

[6] Website, GPUSORT: http://gamma.cs.unc.edu/GPUSORT/

[7] A.L. Wiens and B.J. Ross. Gentropy: Evolutionary 2D Texture Generation. Computers and Graphics Journal, 26(1):75-88, February 2002.

[8] K. Sims. Interactive evolution of equations for procedural models. The Visual Computer, 9:466-476, 1993.

[9] M. Ebner, M. Reinhardt and J. Albert. Evolution of Vertex and Pixel Shaders. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert and M. Tomassini (editors): Proceedings of the Eighth European Conference on Genetic Programming (EuroGP 2005), Lausanne, Switzerland, pp. 261-270, © Springer-Verlag, 2005.

[10] J. Meyer-Spradow and J. Loviscach. Evolutionary design of BRDFs. In M. Chover, H. Hagen, and D. Tost, eds., Eurographics 2003 Short Paper Proceedings, pp. 301– 306, 2003.

[11] M. L. Wong, T. T. Wong and K. L. Fok. Parallel Evolutionary Algorithms on Graphics Processing Unit. *Proceedings of IEEE Congress on Evolutionary Computation 2005 (CEC 2005)*, Vol. 3, Edinburgh, UK, September 2005, pp. 2286-2293.

[12] Website, ECJ: http://cs.gmu.edu/~eclab/projects/ecj/