Evolving a CUDA Kernel from an nVidia Template*

W. B. Langdon and M. Harman

Technical Report TR-10-07

Department of Computer Science, King's College, London,
Strand, London, WC2R 2LS, UK

6 July 2010

Abstract

We automatically generate an nVidia parallel CUDA graphics card kernel with identical functionality to existing highly optimised ancient sequential C code. Essentially generic GPGPU C++ code supplied by the hardware manufacturer is converted into a BNF grammar. Strongly typed genetic programming uses the BNF to generate compilable and runnable graphics card kernels, which terminate. Their fitness is given by running the population on a GPU against a test suite used to test the original sequential code.

I will briefly introduce genetic programming (GP) but try to assume most of the audience are familiar with the essential idea of using Darwin's idea of species evolution by randomly selecting better individuals from a population of individuals (in our case programs). Each new population is created by randomly breeding from the better individuals in the previous generation. In genetic programming [Poli *et al.*, 2008] we need to start by creating the initial population of random programs. We usually also need an automated way of selecting better programs. This is usually done by testing. There are a number of ways of mutating better programs and a number of ways to combine two better parent programs to create children programs. Many mutants are very poor. Some children are like their parents and a few are better than their parents. After generations of repeatedly selecting the better children and breeding from them, useful solutions can sometimes be automatically created.

The bulk of the talk will be about using GP to automatically create small pieces of source code. GP is usually used to generate solutions: designs, models, data mining, etc. but seldom source code itself. However recently [Weimer *et al.*, 2010] has shown GP can automatically generate bug fixes to C code. I will describe a "proof of concept" experiment in which GP is used to automatically generate CUDA C code to run on a graphics card.

The idea is that for GP code generation to be competitive with conventional coding it needs to operate in a niche where either people find the problem hard or the right mix of skills is either hard to come by or difficult to retain. Although machines should be able to create vast quantities of artifacts cheaply, this has never applied to software, which remains labour intensive. We sidestep the scaling problem by looking at only generating small (but potentially valuable) parts of much bigger software systems. For example, glue logic to interface disparate systems, performance critical components or moving legacy code to new hardware or applications. Typically there is no formal specification and yet the requirements are well known but the solutions are not and there may be a novel and potentially

*Invited presentation to the Natural Computing Applications Forum, 12-13 July 2010, Surrey University

complex trade off between multiple requirements. For example, porting legacy code to a new hand-held device may have onerous restrictions on battery life, bandwidth and software memory foot print but need only real time response for a single user. Potentially a genetic search based technique can simultaneously try many wacky out-of-the-box approaches, whereas human designers would only be able to try one or two and these will be based upon their best up front guess, itself limited by previous experience. The automated approach can try many, perhaps leading to a multi-objective Pareto optimal trade off surface, from which the designer will make a choice. Indeed, the human designer, having chosen a novel GP generated approach, may decide to use conventional coding methods.

One approach is to use the existing legacy system as the environment in which the new source code will be created. The existing system is instrumented so that when it is exercised data flows through the to-be-evolved interface are recorded. This can quickly lead to vast volumes of data which can be used to train the GP system. The GP generated code is run inside a test harness and supplied with recorded data. How it transforms that data is compared with the transformation in the legacy system. The more faithfully the new code resembles it the better. However we can also use other, perhaps non-functional, properties to decide which of the many GP individuals to retain and breed from. For example, speed, size and memory usage. Other environmental aspects such as power consumption may need dedicated hardware or complex simulators.

In the proof of concept experiment [Langdon and Harman, 2010] the small CPU time performance critical function of GNU gzip was instrumented and gzip was run on a standard test suite. The function was then replaced by automatically generated parallel code which runs on a graphics card. This is a very early demonstration. The new code runs in a totally novel environment (GPUs do not run Unix) and provides a complete emulation of the original gzip but does not yet give a performance gain. This is the first time code has been ported to an nVidia CUDA kernel that has been automatically evolved. The new code has been tested literally millions of times without error.

References

[Langdon and Harman, 2010] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In Pilar Sobrevilla, editor, *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July 2010. IEEE. Forthcoming.

[Poli *et al.*, 2008] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[Weimer *et al.*, 2010] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, June 2010.