Software is Not Fragile

William B. Langdon, Justyna Petke

Software is Not Fragile

William B. Langdon and Justyna Petke,

CREST, Department of Computer Science,
University College London Gower Street, London WC1E 6BT, UK

Abstract. Trying all simple changes (first order mutations) to executed C, C++ and CUDA source code shows software engineering artefacts are more robust than is often assumed. Of those that compile, up to 89% run without error. Indeed a few mutants are improvements. Program fitness landscapes are smoother. Analysis of these programs, a parallel nVidia GPGPU kernel, all CUDA samples and the GNU C library shows many lines of code and integer values are repeated and may follow Zipf's law.

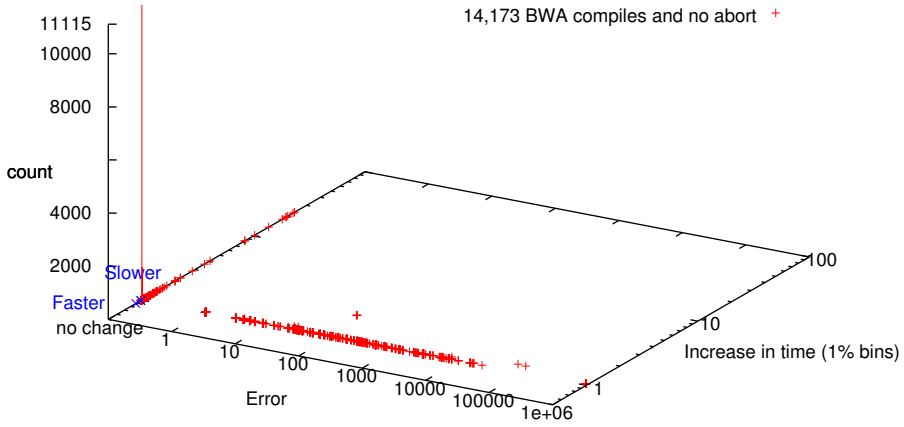

Fig. 1. Impact of all possible executable changes to example program (Section 2.1). Of the 23% which compile and run normally 89% produce the same answer as the original code ("no change"). Indeed 3 of them are faster (×).

1 Introduction

It is often assumed that computer programs are fragile and any single change will destroy them totally. Figures 1, 2, and 3 show this is not true. We automatically make changes like those a human programmer might make. I.e. delete a line of a program, replace a line with another and insert a copy of a line. Figure 1 shows the impact of all possible changes to the part of the program which is used. 63% do not compile and 14% abort when run. However, most of the modified programs which compile and run normally produce exactly the same result as the original program source code. Indeed a few produce identical answers but are slightly faster. We suggest often wholesome changes can be quickly found.

Software engineering continues to produce some of the most complex human artefacts on the planet. Mostly it has succeeded in its goal of describing in complete detail the computer systems people create, maintain and use. Nonetheless, despite extensive tools, large software systems are beyond comprehension. They cannot be fully understood by anyone no matter how clever, nor can they be understood by groups of people, not even the team of experts who may have been working on them for years. Nonetheless they continue to yield enormous economic advantage which has lead to the world economy being addicted to software.

Genetic Improvement (GI) [1,2,3] applies search-based optimisation techniques (SBSE) [4], such as genetic algorithms (GA) [5], genetic programming (GP) [6] and hill climbing to make measurable improvement in existing human written code. Although GAs [7] and GP [8] have been analysed for sometime, very little theory is available which can predict how the performance of real software systems will change in response to changes to their source code. In SBSE this is known as the fitness landscape, which depends both on the program and the mechanisms available to change it.

Section 3 shows software, like other human endeavours, can, both in terms of source code [9] and numeric values, follow Zipf's law [10]. But we start in Section 2 with two leading open source Bioinformatics tools (one in C, the other in C++) and an open source parallel CUDA GPGPU kernel and show, for these examples, software is not as fragile as is often assumed. For these diverse source code examples, we start to map their software fitness landscapes. This initial mapping suggests that software is much more resilient than is commonly assumed. This in turn supports the idea of artificial evolution of programs via mutating existing code (GI).

2 Mutating Useful Software

The following three subsections describe the impact of making simple changes (of the 3 types described at the start of Section 1) to three programs written in C, C++ and CUDA. We call these changes mutations. We only make one change at a time and so these are known as first order mutations. Multiple changes are known as higher order mutants [11]. In Figures 1, 2, and 3 we report the change in output (excluding logging information) and the change in run-time. In each case the mutations target code which is used by the test case and yet many mutations do not change the output at all (cf. equivalent mutants [12]).

2.1 BWA

BWA [13] is a leading open source Bioinformatics tool written in the C programming language for matching short next generation DNA sequences to the human genome. It comprises 33 .c and 20 .h files containing 13 000 lines of code. We down-loaded the complete installation kit (version 0.7.12) from GitHub and automatically converted all the .c C source files into a specialised BNF grammar

of 18 621 rules. As with our earlier work [14], we use the grammar to control the mutations [11]. It ensures all mutations are syntactically correct (in that brackets match, there are semi-colons where required, etc.) however, due to variable out-of-scope and other errors, it need not ensure a mutant compiles.

We also obtained a test sequence from http://fg.cns.utexas.edu/fg/course_notebook_chapter_seventeen.html. We used the GNU gcov test analysis tool to discover all the lines of C source code executed. (gcov shows 532 lines of the BWA program are used by this test case.)

We use the three types of mutation: delete a line of source code, replace it with a copy of another line of code and insert a copy of another line of code. As with our previous work [14], we restrict replacements and inserts to be of the same type and from the same .c source file. Notice mutations are limited to re-using existing source code. They do not create new code from scratch. To ensure all mutations are executed, all mutations either: delete or replace a line which is executed or insert a source line immediately before it. There are 61 775 possible mutations of BWA. We generate and test them all (see Figure 1).

To catch indefinite loops, we impose a CPU limit. The limit is about 40 times longer than normal operation. We also automatically remove computer jobs (known as processes to the Unix operating system) which failed to terminate normally. Our zombie.awk killer removed nine such zombies.

2.2 Stereo Pair CUDA Image Processing

StereoCamera is open source CUDA code written by nVidia's vision processing expert [15]. It takes two stereo images and from the parallax differences between them it infers distances between objects in the images and the camera. We had previously used our BNF grammar approach to automatically evolve considerable speedups [16]. Here we reuse our automatically generated grammar to start to map the fitness landscape for CUDA software. As a test case we use a stereo image pair taken inside a typical modern office (provided by Microsoft's I2I database). In order to get a somewhat different example, we concentrate on the parallel CUDA code written for the graphics card, (known as a GPU kernel) and ignore the program code which runs on a normal personal computer. The kernel contains 276 lines of CUDA code.

The StereoCamera GPGPU [17] kernel is more constrained than normal C program code. 1) all of it will be used, 2) we can readily force indefinite loops to stop running and 3) (excluding a compiler bug) the grammar will ensure mutants always compile.

We deliberately exclude changes to tuning parameters, CUDA specific pragma and kernel parameter changes (which were used in [16]) as by design they cannot break the existing CUDA kernel.

We tried all possible single change mutants (i.e. first order mutants). Figure 2 shows that about 5 in 8 source code mutations do not change the kernel's output on the randomly chosen test image pair.

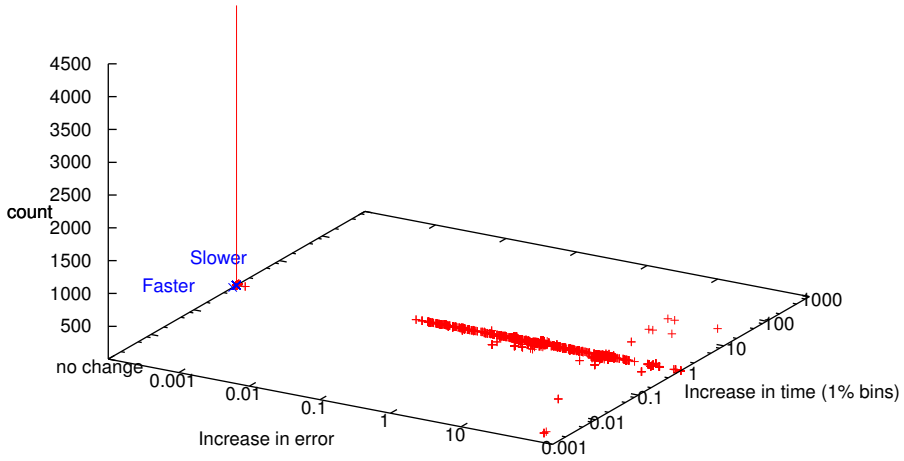

Fig. 2. Impact of all (7079) one change mutations on StereoCamera kernel [16] running on an nVidia GeForce GT 730 graphics card. Nine mutants failed to compile due to a bug in the nVidia CUDA 6.0.1 compiler (fixed in CUDA 7.0). 16 caused infinite loops, 318 others failed at run-time. 4400 (62%) mutants do not change the output at all ("no change"), indeed at least 41 of them are faster (×).

2.3 Bowtie2

Bowtie2 [18], like BWA, is a state-of-the-art next generation DNA analysis program. We made random source code mutations to Bowtie2 and measured their impact on random test cases, see Figure 3. Although many mutations cause Bowtie2 to fail (not plotted) and others cause it to produce very poor solutions (e.g. reducing quality by 36, left) others have less dramatic impact. Some slow down Bowtie2 and others make it faster. However, many changes have no impact on quality (although they may change Bowtie2's speed, plotted along x=0). Indeed a large number do not change its speed either (note spike at the origin). There are even a few mutations which give better quality solutions and even 139 which are both better on a random test and faster (plotted in Figure 3 with +). It is from these a seventy fold speed up can be evolved [14].

3 Zipfs Law

George Zipf (USA, 1902–1950) proposed a "universal" law of human behaviour [10] in which the frequency of repeated items is inversely proportional to their rank (or order). Thus when plotted on log-log scale we get an inverse power law with a slope of -1. This has been shown to hold for the frequency with which words in many languages are used, the population of cities and many other human endeavours, including software engineering [9].

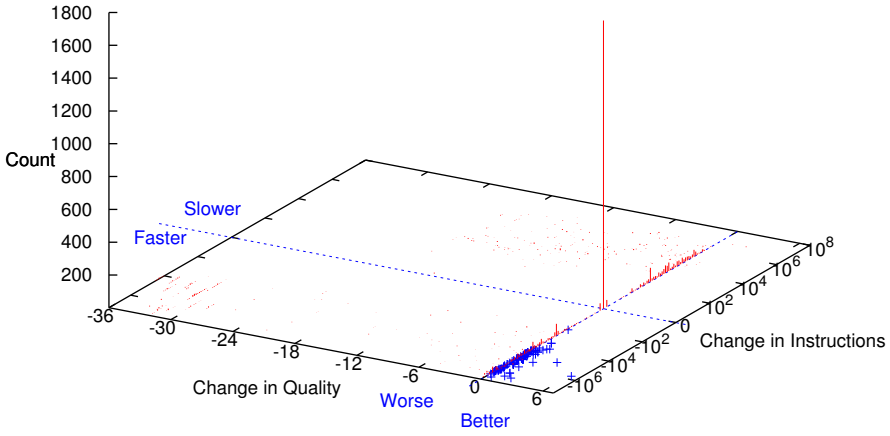

Fig. 3. Impact of single mutations to Bowtie2 [14]. Non-linear scales [19].

Using the grammars described in Section 2, Figure 4 shows the distributions of exactly repeated BWA C and Bowtie2 C++ lines of source code both approximately follow Zipf's law.

3.1 What's my favourite number?

The GNU C library, version 2.22 released 14 August 2015 (excluding its test suite), contains 845,360 lines of C code, which contain in total 1 203 104 integer constants (see Figure 5). Many glibc integers are associated with mapping non-ASCII character sets, e.g. Chinese.

nVidia's CUDA 7.0 comes with an extensive set of examples (including test code) totalling 85 711 lines of C++ or CUDA code. These contain 73 620 integer constants (see Figure 6). Many numbers in the CUDA 7.0 samples are taken from the OpenGL package. For example, the OpenGL source code includes integers for use as bit masks which are used to turn on image effects, such as stippling. Nevertheless we do not see pronounced clustering around the powers of 2, visible in our earlier work on CUDA 5.0 samples which included macro expansion.

In both the GNU C library and nVidia's samples, as with the whole lines of source code in Section 3, numeric values approximately follow Zipf's law.

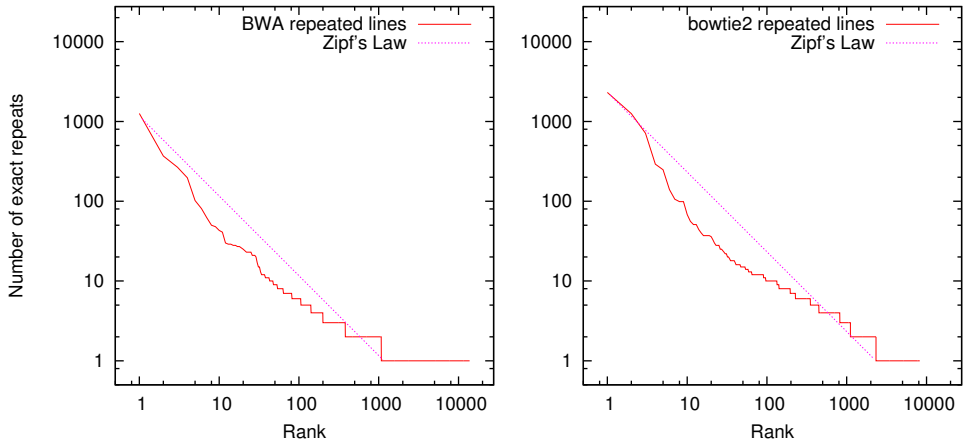

Fig. 4. Distribution of exactly repeated BWA C program source code (left) and Bowtie2 C++ source code (after macro expansion) [14, Fig. 5] (right). Zipf's law [10] predicts a straight line with slope of -1.

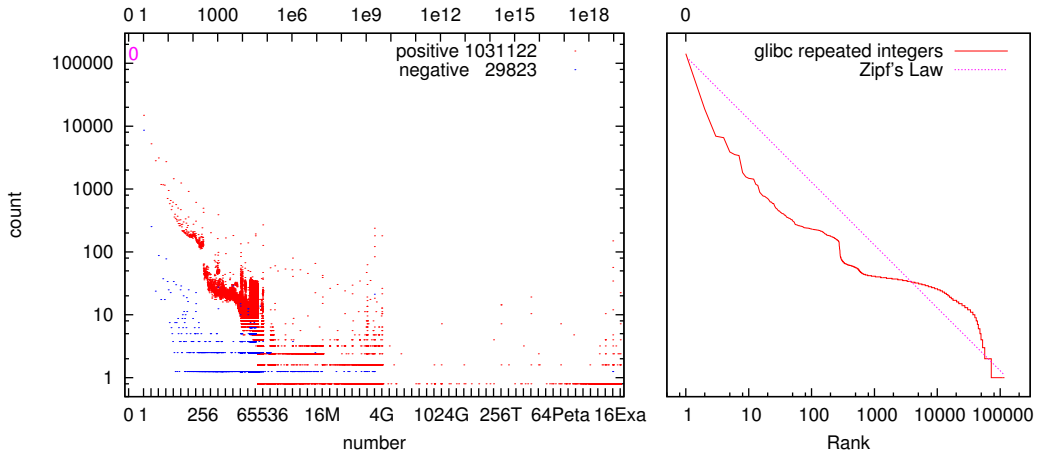

Fig. 5. Distribution of integer constants in GNU C library. Zero is the most common number, occurring in various formats a total of 142 159 times, followed by 1 (18 642) and -1 (6 907). Every integer between -28 and 40 957 occurs at least once. There are 116 685 distinct integer constants.

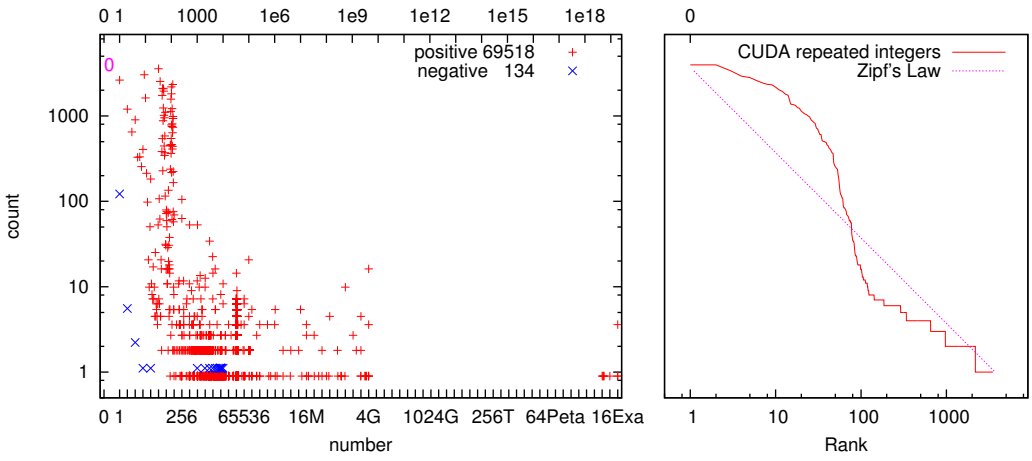

Fig. 6. Distribution of integer constants in CUDA 7.0 sample source code. Zero is again the most common number, occurring in various formats 3968 times. Surprisingly the second most popular number is 32 (3962), which probably reflects the nVidia GPU architecture. There are even fewer negative numbers than in the GNU C library (Figure 5). Every integer between -2 and 60 occurs at least once. There are 3490 distinct integer constants.

4 Conclusions

The existence of power laws in software has been previously reported. E.g. Louridas et al. [9] reported power laws in both the patterns with which functions or classes are used and in the frequency with which machine code instructions are used. Nevertheless the prevalence of Zipf's law in human-written program code does appear to be under-appreciated. Although Louridas et al. caution against getting carried away with the supposed universality of power-laws, it is tempting to suggest that the Zipf law we see with numbers might make it a good candidate for either numeric values to be used when creating test suites. Or indeed (although this is not yet done) numbers to be embedded into GI-created code might be drawn from a Zipfian distribution. Actual numerical values (see left hand sides of Figures 5 and 6) might be drawn with a preference for low entropy decimal, hexadecimal or octal strings. (In genetic programming [20,6] floating point numbers, some-times known as ephemeral random constants, ERCs, are typically drawn uniformly from a small range [21], although we have had good results from a tangent distribution [22].)

In Section 2 we saw in three different (albeit similar) industrial strength programming languages examples of high quality software (including serial and parallel programs) written by experts where it was comparatively easy to find simple source code changes which on a random test make no external difference. Indeed, if the new program code can avoid the trap of introducing variable

out-of-scope errors (i.e. avoid compiler errors) most mutants will not break the program immediately. (In a few cases [23,16,24,25,26] we have set up the GI system to carefully track the scope of variables and thereby avoided compilation errors.) Of course other programming languages are much more forgiving and will assume newly introduced variables should have been declared and will do this automatically for the lazy programmer.

It may be countered that we have not tested the changes sufficiently. This is deliberate. We know how much we have tested each change (exactly once). This gives us a solid benchmark from which to do comparisons. In real GI work, post evolution code, maybe subjected to much more rigorous validation. (In [23] in addition to manual validation, we tested the new code back-to-back with the old more than a million times. No difference was ever found.) Although great strides have been made with automatic testing [27], it remains true that devising tests to execute newly written program code is difficult. Indeed one often gets the impression that simply running the new code is considered sufficient, without caring if indeed it calculated the right answer. We have set up our experiment so that we know our mutated program code is executed. We can tell if it calculated the right answer, or at least we can tell if it calculated the same answer as the human-written program.

Our purpose is to put to bed the myth that any random change will destroy human-written programs. Instead we have given persuasive evidence that whilst a random changes might be bad, if you are prepared to try multiple times, perhaps adapting a population approach, you can quickly find equivalent mutants and you may, if guided by a suitable fitness function, find improvements to your software.

References

1. Langdon, W.B.: Genetically improved software. In Gandomi, A.H., Alavi, A.H., Ryan, C., eds.: Handbook of Genetic Programming Applications. (Springer) Forthcoming.
2. Langdon, W.B.: Genetic improvement of software for multiple objectives. In Labiche, Y., Barros, M., eds.: SSBSE. Volume 9275 of LNCS., Bergamo, Italy, Springer (2015) 12–28 Invited keynote.
3. Langdon, W.B., Petke, J., White, D.R.: Genetic improvement 2015 chairs' welcome. In Langdon, W.B., Petke, J., White, D.R., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 791–792
4. Harman, M., Jones, B.F.: Search based software engineering. Information and Software Technology **43**(14) (2001) 833–839
5. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. MIT Press (1992) First Published by University of Michigan Press 1975.
6. Poli, R., Langdon, W.B., McPhee, N.F.: 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).
7. Beyer, H.G., Langdon, W., eds.: Foundations of Genetic Algorithms, Schwarzenberg, Austria, ACM (2011)

8. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer-Verlag (2002)
9. Louridas, P., Spinellis, D., Vlachos, V.: Power laws in software. ACM Trans. Softw. Eng. Methodol. **18**(1) (2008) 2:1–2:26
10. Zipf, G.K.: Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology. Addison-Wesley Press Inc., Cambridge 42, MA, USA (1949)
11. Langdon, W.B., Harman, M., Jia, Y.: Efficient multi-objective higher order mutation testing with genetic programming. Journal of Systems and Software **83**(12) (2010) 2416–2430
12. Yao, X., Harman, M., Jia, Y.: A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Proceedings of the 36th International Conference on Software Engineering. ICSE 2014, Hyderabad, India, ACM (2014) 919–930
13. Li, H., Durbin, R.: Fast and accurate long-read alignment with Burrows-Wheeler transform. Bioinformatics **26**(5) (2010) 589–595
14. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation **19**(1) (2015) 118–135
15. Stam, J.: Stereo imaging with CUDA. Technical report, nVidia (2008)
16. Langdon, W.B., Harman, M.: Genetically improved CUDA C++ software. In Nicolau, M., Krawiec, K., Heywood, M.I., Castelli, M., Garcia-Sanchez, P., Merelo, J.J., Rivas Santos, V.M., Sim, K., eds.: 17th European Conference on Genetic Programming. Volume 8599 of LNCS., Granada, Spain, Springer (2014) 87–99
17. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings of the IEEE **96**(5) (2008) 879–899 Invited paper.
18. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nature Methods **9**(4) (2012) 357–359
19. Langdon, W.B.: Genetic improvement of programs. In Winkler, F., Negru, V., Ida, T., Jebelean, T., Petcu, D., Watt, S., Zaharie, D., eds.: 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2014), Timisoara, IEEE (2014) 14–19 Keynote.
20. Koza, J.R.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT press (1992)
21. Daida, J.M., Bertram, R.R., Stanhope, S.A., Khoo, J.C., Chaudhary, S.A., Chaudhri, O.A., Polito II, J.A.: What makes a problem GP-hard? analysis of a tunably difficult problem in genetic programming. Genetic Programming and Evolvable Machines **2**(2) (2001) 165–191
22. Langdon, W.B.: Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming! Volume 1 of Genetic Programming. Kluwer, Boston (1998)
23. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In Sobrevilla, P., ed.: 2010 IEEE World Congress on Computational Intelligence, Barcelona, IEEE (2010) 2376–2383
24. Langdon, W.B., Modat, M., Petke, J., Harman, M.: Improving 3D medical image registration CUDA software with genetic programming. In Igel, C., Arnold, D.V., Gagne, C., Popovici, E., Auger, A., Bacardit, J., Brockhoff, D., Cagnoni, S., Deb, K., Doerr, B., Foster, J., Glasmachers, T., Hart, E., Heywood, M.I., Iba, H., Jacob, C., Jansen, T., Jin, Y., Kessentini, M., Knowles, J.D., Langdon, W.B., Larranaga, P., Luke, S., Luque, G., McCall, J.A.W., Montes de Oca, M.A., Motsinger-Reif, A., Ong, Y.S., Palmer, M., Parsopoulos, K.E., Raidl, G., Risi, S., Ruhe, G., Schaul, T., Schmickl, T., Sendhoff, B., Stanley, K.O., Stuetzle, T., Thierens, D., Togelius,

J., Witt, C., Zarges, C., eds.: GECCO '14: Proceeding of the sixteenth annual conference on genetic and evolutionary computation conference, Vancouver, BC, Canada, ACM (2014) 951–958
25. Langdon, W.B., Lam, B.Y.H., Petke, J., Harman, M.: Improving CUDA DNA analysis software with genetic programming. In Silva, S., Esparcia-Alcazar, A.I., Lopez-Ibanez, M., Mostaghim, S., Timmis, J., Zarges, C., Correia, L., Soule, T., Giacobini, M., Urbanowicz, R., Akimoto, Y., Glasmachers, T., Fernandez de Vega, F., Hoover, A., Larranaga, P., Soto, M., Cotta, C., Pereira, F.B., Handl, J., Koutnik, J., Gaspar-Cunha, A., Trautmann, H., Mouret, J.B., Risi, S., Costa, E., Schuetze, O., Krawiec, K., Moraglio, A., Miller, J.F., Widera, P., Cagnoni, S., Merelo, J., Hart, E., Trujillo, L., Kessentini, M., Ochoa, G., Chicano, F., Doerr, C., eds.: GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, Madrid, ACM (2015) 1063–1070
26. Langdon, W.B., Harman, M.: Grow and graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In Langdon, W.B., Petke, J., White, D.R., eds.: Genetic Improvement 2015 Workshop, Madrid, ACM (2015) 805–810
27. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: 8^{th} European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11), ACM (2011) 416–419