Genetic Programming and Evolvable Machines
Accepted DOI 10.1007/s10710-016-9273-9

Genetic Improvement of GPU Software

William B. Langdon · Brian Yee Hong Lam · Marc Modat · Justyna Petke · Mark Harman

2 August 2016

Abstract We survey Genetic Improvement (GI) of general purpose computing on graphics cards. We summarise several experiments which demonstrate four themes. Experiments with the gzip program show that genetic programming (GP) can automatically port sequential C code to parallel code. Experiments with the StereoCamera program show that GI can upgrade legacy parallel code for new hardware and software. Experiments with NiftyReg and BarraCUDA show that GI can make substantial improvements to current parallel CUDA applications. Finally, experiments with the pknotsRG program show that with semi-automated approaches, enormous speed ups can sometimes be had by growing and grafting new code with genetic programming in combination with human input.

Keywords Genetic Programming · SBSE · GI-GPGPU · metaprogramming · Grammar Based Genetic Programming · nVidia CUDA · parallel computing · Dynamic Programming · GPGPU · GGGP

PACS Computer science and technology, 89.20.Ff · Computer vision, 42.30.Tz · computed tomography, 87.57.Q- · magnetic resonance imaging, 87.61.-c · nuclear medicine imaging, 87.57.U- · biomolecules, 87.15.A-

W. B. Langdon, J. Petke, H. Harman
Department of Computer Science, University College London
E-mail: W.Langdon@cs.ucl.ac.uk

B. Y. H. Lam
University of Cambridge Metabolic Research Laboratories, Addenbrooke's Hospital

M. Modat
Leonard Wolfson Experimental Neurology Centre, University College London

1 Introduction

Since the effective end of the doubling of CPU processor clock speeds in 2004 with the launch of Intel's 3 GHz Pentium, the use of parallel graphics hardware (GPUs) for main stream computing has become increasingly common. Although general purpose computing on GPUs [Owens *et al.*, 2008] was originally introduced to take advantage of low cost hardware developed for consumer games market, today GPGPU software may be found on GPUs costing less than $50 to super computers with budgets of $325 million. Indeed (as of Dec 2015) two of the top ten fastest computers on the planet are substantially composed of nVidia GPUs (Tesla K20, see Table 2 page 22).

Despite the continued exponential rise in parallel processing power (see Figure 1), sequential programs continue to dominate. This is because parallel programming is hard for people. Indeed the non-standard Single Instruction Multiple Data (SIMD) heavily multiple threaded programming required by GPUs (see Figure 2) is especially difficult for people to program efficiently. To compound the double difficulty of "parallel programming" and the SIMD (or SIMT[1]) programming model, until recently GPGPU programming support tools were poor. Recent versions of nVidia's CUDA include a usable performance profiler and C/C++ debugger and there are a few other commercial tools to support GPGPU software development. Probably the best approach, where possible, e.g. for matrix intensive code, is to avoid writing programs for the GPU but instead rely on pre-existing library routines and call them from sequential code running on the host (i.e. the PC or server to which the GPU is attached, see top of Figure 3). Some languages (such as Matlab and R) can automatically exploit GPU hardware for matrix operations.

nVidia provides help for GPGPU programming on its hardware, e.g. via extensive documentation, online forums, a large number of C/C++ examples and domain specific packages, such as nvBio for Bioinformatics applications. A notable open source GPGPU package is Thrust. Both nvBio and Thrust make heavy use of C++ templates, which tends to make them hard to understand. Also it appears that Thrust is sometimes difficult to integrate with other code which accesses the same GPU, and nvBio has some bugs. GPGPU programming has been and still remains hard. This is probably why there are still relatively few GPU Bioinformatics applications [Langdon and Harrison, 2008]. Unfortunately GPGPU applications will not become wide spread whilst GPGPU programming remains an uber-geek activity.

Our approach to deskilling GPGPU software development is different. Largely we take the approach of hoping the existing commercial and open source high level libraries and packages will be successful at easing the path to the development of correct code but rely on the machine to transform correct code into efficient parallel code. Indeed we show (Sections 5–7) that Genetic Improvement can automatically tailor C source code to get high performance specific to each different GPU.

[1] Single Instruction Multiple Threads, SIMT

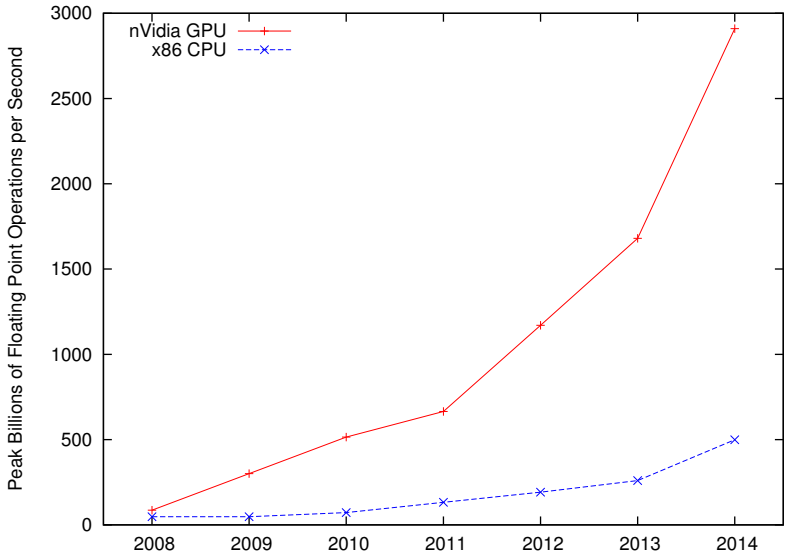

Fig. 1 Exponential growth in *peak* processing power. Data from nVidia

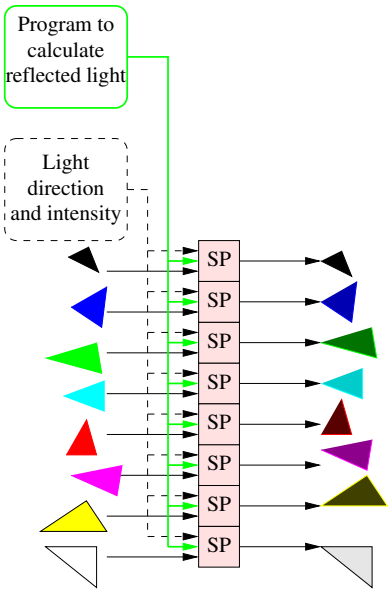

Fig. 2 An example of SIMD parallel processing. The stream processors (SP) simultaneously run the same program on different data and produce different answers. This program has two inputs. One describes a triangle (orientation, position, colour, how shiny or matt its surface is). The second input refers to a common light source and so all stream processors use the same value. Each stream processor calculates the apparent colour of its individual triangle. Notice, here, each output is independent of all the others and so they can all be calculated in parallel.

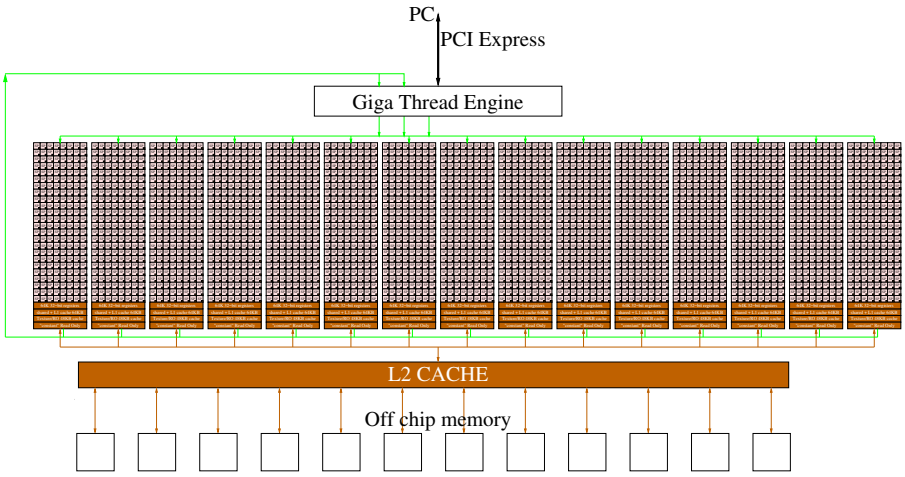

Fig. 3 A Tesla K40 contains a PCI Express interface to the host personal computer (PC), thread handling logic, 15 SMX multiprocessors, and 12 Gigabytes of on board memory. Each SMX contains 192 stream processors (total 2880). See also Table 2 page 22.

2 Genetic Improvement for GPGPU

Genetic Improvement is an up and coming topic with a growing range of applications [Langdon, 2015b; Langdon, 2015a]. Whilst [Brady *et al.*, 2014] used genetic programming to automatically create better than the state-of-the art reflectance functions these are for use with GPUs in traditional computer graphics, rather than general purpose computation on GPUs (GPGPU). Similarly [Sitthi-amorn *et al.*, 2011] used GP to improve existing graphics shaders, but again this is using graphics hardware for graphics applications rather than GPGPU. (Although Section 5 deals with images, its slightly different in that it extracts implicit rather than explicit information from multiple images.) Also in [Langdon, 2010] we used GPUs but this was purely to speed genetic programming itself. So far there are five substantial genetic improvement experiments which deal with general computing on GPUs:

1. Section 4 describes evolving a replacement to the most computationally demanding part of the popular Unix compression utility gzip [Langdon and Harman, 2010].
2. Section 5 discusses evolving changes to code written by nVidia's vision processing expert which several years later enabled it to be used on modern hardware and software. In addition to the speed ups given by the more powerful GPUs now available, the GI software gave up to a seven fold speed up on real stereo image pairs [Langdon and Harman, 2014b]. See Figures 19 (page 21) and 20 (page 23).
3. In Section 6 genetic improvement is applied to the most computationally intensive code used when performing registration of images of the human brain. The NiftyReg package [Modat *et al.*, 2010] contains many such GPU

kernels. So far it has proved impractical to update and tune them all by hand for every software update and for new graphics hardware. The GI approach [Langdon *et al.*, 2014] shows one way in which it may be plausible to attempt to do this automatically.

4. Section 7 discusses the first application of Genetic Improvement in use. BarraCUDA [Klus *et al.*, 2012] is a state of the art C++ program which maps short DNA fragments against a reference genome. Aligning (i.e. mapping) short DNA sequences is the first step to assembling a complete DNA sequence for the organism. (Typically a human patient but the process and indeed the software, applies to any living organism. E.g. bacteria, fly or cabbage). Once aligned interesting variations (mutations) in the individual can be found.

   BarraCUDA was initially created by a team of nVidia, GPGPU and Bioinformatics experts who ported the popular BWA [Li and Durbin, 2010] tool to run on GPU hardware. Nevertheless, Section 7 will show that genetic programming in combination with manual effort was able to give more than a 100 fold improvement in a critical component. Naturally the overall improvement is more pedestrian but still more than sufficient to convince the owners of BarraCUDA to accept the GI version of the code and make it available[2]. (The GI version has been downloaded more than a 1000 times.) The GI version has been fully integrated, including applying fixes (e.g. to bugs inherited from BWA).

   The old and new releases of BarraCUDA have been run against BWA and Bowtie2 [Langdon and Lam, 2015]. Even a £50 GPU running BarraCUDA can be faster than BWA on a twelve core CPU. With a top end nVidia Tesla GPU, BarraCUDA can be more than ten times faster than BWA on a 12 core CPU.

5. Section 8 summarises the first use of the grow and graft GP approach to evolving better GPU code [Langdon and Harman, 2015a]. RNA is an important biological polymer related to DNA. As with many other molecules, RNA's chemical activity is largely dictated by its shape. Unlike proteins, the shape, indeed the range of shapes, into which RNA folds can be predicted by computer models using Dynamic Programming.

   Dynamic Programming is essentially a matrix method and so should prove easily parallelisable. However pknotsRG [Steffen and Giegerich, 2006; Reeder and Giegerich, 2004] gave very poor performance. This is because typically RNA molecules used in computational experiments are quite short and this limits the degree of parallelism available in the Dynamic Programming matrix, however a modern GPU can readily process many thousands of such matrices. By telling evolution where to evolve new code and using the existing code as a harness, grow and graft GP was able to evolve correct code which ran at up to ten thousand times faster for the shortest RNA molecules. See Figure 28 (page 30).

[2] BarraCUDA is on SourceForge `http://sourceforge.net/projects/seqbarracuda/`

The next section recaps the Genetic Improvement process, including why we use grammar based Genetic Programming, the genetic operations and the use of the existing software and test suite as a test oracle to specify the required functionality. After Sections 4 to 8 (see above), Section 9 puts these results into the wider context and Section 10 concludes. (Appendix A gives the internet locations of various Genetic Improvement tools.)

3 Genetic Improvement of Software

By starting from an existing program GI has several advantages. The first is the existing code becomes a *de facto* specification. (In the case of automatic bug repair it is assumed that the buggy program is almost correct and only a small part of the existing functionality needs changing [Weimer *et al.*, 2010].) Every new version of the program produced by GI can be compared against the original. Typically test cases are used both 1) to see if the GI mutant still retains the functionality of the original and 2) to see if it is in some measurable way better than the original.

3.1 Combining Man and Evolution

In non-GPU work we have investigated the active combination of human coding and evolution. For example in [Harman *et al.*, 2014] we showed that given strong manual hints (e.g., adding the Google translate API to the function set) GP could evolve an international bi-translation feature. Again giving hints (such as a target routine) GP could graft the newly evolved code into a social media package of more than 200 000 lines of C code. More recently this "grow and graft" approach has been used to add a Python citation service based on Google Scholar to a sizeable existing web server [Jia *et al.*, 2015].

As mentioned in Section 1, GPU programming remains difficult. Various approaches have been advanced to move programmers to a higher level. For example, [Merrill *et al.*, 2012] advocates the use of C++ high level templates. However, from personal experience, such indirection seems to place additional load on the human programmer. Unfortunately the current GI approach is still very much research in progress, but it too has the goal of allowing the machine to do more of the heavy lifting work of getting the code to work efficiently, whilst leaving the role of specifying what needs to be done to the programmer.

The first part of the GI approach to CUDA programming is to use simple optimisation on key CUDA parameters, such as block size. Where there are one or even two such parameters, it may be feasible to try all reasonable values (exhaustive search) before evolving the code. Typically these key parameters are also re-optimised after evolution. As the number of parameters increases, the number of options increases, typically exponentially. For example in Section 7, the number of configuration options is $2^{13} \times 7 \times 37 = 2\,121\,728$.

Therefore the chromosome was expanded to explicitly include these parameters (see Section 3.4). Effectively the GI becomes a co-evolution of a grammar based GP evolving code changes and a GA evolving the parameter changes.

In the case of StereoCamera Section 5, NiftyReg Section 6 and BarraCUDA Section 7, heavy use was made of conditionally compiled man-made code (see `#ifdef` etc. in grammar fragment in Figure 7 page 12). The conditional compilation switches become additional fixed parameters, whilst the new code, like the original code, is subject to change by the grammar based GP. The idea being, there are multiple ways of coding CUDA functionality. It is impossible for the novice (and very hard even for an expert CUDA GPU programmer) to know in advance which will be the most efficient. What is worse, code tuned for a particular user load and/or particular GPU may be less efficient with either different load or on a different GPU. For example, in the case of NiftyReg (Section 6), there are many hand written kernels and it has proved impossible in practice to re-work them by hand for more modern GPUs.

A common choice is where and in what format to store data. The GPU offers choices such as: on the host (via "zero copy"), on board global memory, on chip shared memory, texture memory, the texture's read-only cache, constant memory and registers. Each of these have different GPU dependent performance and limitations. The choice of array-of-structures or structure-of-arrays is well known but there may be other choices, such as should the programmer use bytes or words, or in the case of DNA strings, is it worth compressing the data into two bits for each base pair. The implications of any of these choices (which are forced onto the human programmer) are far from obvious. By the use of conventional coding practises, e.g. (inlined) functions and macros, and conditional compilation these choices can be deferred to the machine (in the form of evolution). Also when circumstances change (e.g. the introduction of global memory caches) we can automatically undo customisation for one GPU and re-optimise for another. As well as code mutations, the conditional compilation switches become parameters to be chosen by GI.

For example, the original version of BarraCUDA included a software level cache for the last 8 DNA bases. As with all caches, the goal was to produce better code with the same functionality. By wrapping the code to implement the cache inside conditional compilation under GI control, it was discovered that on modern Tesla-class GPUs, the hardware caches were sufficient and actually the software cache was now slowing down the kernel and could be removed.

3.2 The GI Fitness Function

In evolutionary computation the role of the fitness function is to guide the search. In these experiments, to reduce run-time only a small subset of the available tests need be run to decide which members of the current population will have descendants. To avoid the GI population becoming over specialised, which test cases are used is frequently changed (see also [Langdon, 1998;

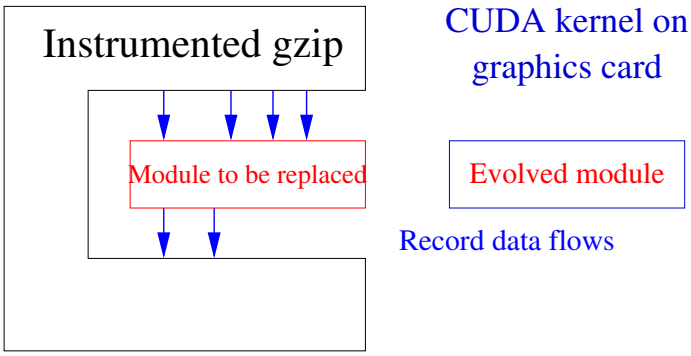

Fig. 4 The original code (left) is instrumented to record the input and output data (arrows) of the target function (red) every time it is called. These become the fitness function and test suite for the automatically evolved replacement module running on the GPU. In the context of gzip, the CUDA code (Figure 6) generated by GP is functionally identical to the C code inside gzip [Langdon and Harman, 2010].

Teller and Andre, 1997; Gathercole and Ross, 1994; Foster, 2001]). Only after evolution does the goal change to validating the evolved code. In validation it is common to use a large number of tests including, to check for over-fitting, tests that were not used during evolution. Indeed it may be feasible (as with gzip, Section 4) to operate the new GI system back-to-back with the original and verify they produce the same, or at least compatible, answers. (For example, the GI CUDA gzip code was run and compared with the original more than a million times and no difference was ever seen.)

Figure 4 shows GI being applied to part of an existing system. The original system is run to gather information about the data used by the module of interest. (In this case the function which uses most of the CPU time.) The function's inputs and corresponding outputs are recorded. These become the definition of the new CUDA kernel. I.e. given these inputs, the old code produced these answers, so the new code should produce the same answers. In floating point code we would allow certain tolerance between the old and the new. Indeed with noisy data or imprecise algorithms, we can imagine the fitness function allowing a bigger discrepancy between the two if there is some possible advantage, such as more quickly calculating a less accurate answer.

Here typically *better* is defined by the fitness function to mean faster but it could, for example, mean uses less memory [Risco-Martin *et al.*, 2010; Wu *et al.*, 2015], less energy [Schulte *et al.*, 2014a; Bruce, 2015; Burles *et al.*, 2015a] or gives a better tradeoff between speed and quality of the answers [Langdon and Harman, 2015b]. Indeed presenting the user with a range of options which tradeoff multiple axes of improvement may be an avenue to early adoption of GI [Harman *et al.*, 2012a].

In Figure 5 fitness testing is followed by selection. Since the goal is to generate programs which are better than the original, we typically run the original code on the current test cases to establish its performance as a baseline. To be eligible to have children each mutant has to be better than the baseline.

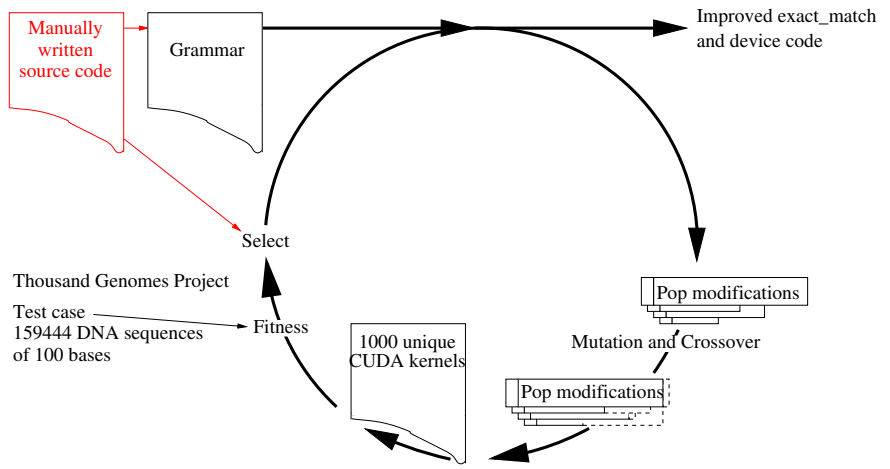

Fig. 5 Major components of Genetic Improvement (GI). Left: system to be improved and its test suite. Right: genetic programming optimises patches which originate from a grammar that describes the original system line by line. (Shown for BarraCUDA Section 7.) Each generation mutation and crossover create new patches. Each patch's fitness is evaluated by applying it to the grammar and then reversing the grammar to get a patched version of the system. Each patched system is tested on a small randomised subset of the test suite and its answers and resource consumption compared to that of the original system. Patches responsible for better systems procreate into the next generation.

In multi-objective problems, we typically take a relaxed definition of better and only demand the mutant exceed the original's performance in one objective. At present, with a noisy fitness, we use a tolerance to ensure the mutant is truly better. However, one could imagine other systems successfully using selection schemes with more relaxed definitions of what it means to be better than the original code. We use weak selection in the form of binary rank selection, which means each member of the top half of the population produces two children. Again it is possible that schemes with a higher selection pressure, e.g. tournament selection, could be used [Langdon and Poli, 2002].

3.2.1 Ensuring for Loop Termination

Although in recent years GPUs and their software have become more sophisticated, running a kernel that loops indefinitely may require manual intervention to restore control. Hence our evolutionary system always ensures loops terminate. Typically loops are aborted after a predetermined number of iterations and the misbehaving kernel given a poor fitness value. In the CUDA code shown in Figure 6, this is implemented by the macro `ok`. Each time `ok` is invoked, it increments a hidden per-thread counter; after the counter exceeds a generous limit, `ok` returns false and the thread terminates. The grammar which constrains GP (see Section 3.3) is constructed to ensure evolution cannot avoid calling `ok` and cannot affect the counter.

```
__device__ int kernel978(const uch *g_idata, const int strstart1, const int strstart2)
{
int thid = 0;
int pout = 0;
int pin = 0 ;
int offset = 0;
int num_elements = 258;
 for (offset = 1 ; G_idata( strstart1+ pin ) == G_idata( strstart2+ pin ) ;offset ++ )

{

if(!ok()) break;
thid = G_idata( strstart2+ thid ) ;
  pin = offset ;

}

return pin ;
}
```

Fig. 6 C++ code of the gzip CUDA kernel automatically generated by GI.

3.2.2 Protecting Array Indexes

As with all C code, there is no defined behaviour for when an index goes out of the bounds of the corresponding array. Modern GPU and more recent software tend to be well behaved. However, since not being defined includes the possibility that the GPU locks up until manually rebooted, in practise GP systems usually ensure array indexes are well-behaved:

- Figure 6 shows the use of the wrapper function `G_idata`, which allows evolved code to access array elements safely by forcing its input to lie in the valid index range of the array. The array bounds are known. Any index outside the valid range is masked into the valid range, without fitness penalty. The grammar is written to ensure `G_idata` is the only way to access the array.
- GPUs also provide specialist arrays called textures. Although primarily intended to allow arrays storing images to be read via specialised GPU hardware, they can be configured to give defined (and safe) behaviour on index out of bounds errors. However, details of texture caches are not documented and may vary between GPUs. Also caches are always limited. Therefore, replacing an array with a texture may affect performance in unexpected ways.
- In the case of recent GPUs and modern versions of CUDA, it is possible to run the kernels under the protection of the CUDA memcheck tool. memcheck can be used to detect and report addressing errors. However, it imposes a considerable and unpredictable overhead, necessitating well behaved kernels be run again without memcheck to get accurate performance data. The advantage of this approach is that, in the second run, the kernel does not incur the overhead of using `G_idata` or require the use of textures.

Often the fitness function will penalise code whose indexes go out of range and so prevent the evolution of extreme cases, which might exceed the GPU's hardware protection or memcheck's abilities.

3.3 Evolving via a Grammar

Figure 5 shows how GI adapts the traditional evolutionary algorithms search technique. The first (off-line) step is to automatically generate a grammar which describes the target program and legal modifications of it. (See Figure 7. Although in BNF format, the grammar is entirely dedicated to this one program and is not general like, for example, the grammar describing the C programming language.) By operating at the CUDA source code level, the grammar approach gives the great advantage of rendering everything, including the GP itself, visible. However, this comes with the disadvantage that (in many computer languages) the GI modified code must be compiled. Therefore, the compiler must be run many times and becomes a significant overhead. (There are ways of reducing this overhead, see Section 3.3.1.)

In the main evolutionary loop, GI evolves not complete programs but changes to the original program. In the grammar based approach, these are represented in plain text as variable length lists of mutations. These are applied to the grammar, which is then effectively reversed to generate new source code. (The grammar is written in plain text, e.g. Figure 7. It describes the source code and source code can be generated from it. Hence modified source code can be generated from the modified grammar.) The modified source code is compiled and the resulting program subject to fitness testing (see Section 3.2).

The grammar ensures the mutated code is syntactically correct (e.g. all the brackets match, there are semi-colons where there should be). However, there may be other, semantic errors, which cause the compilation to fail. Almost all compilation errors are caused by moving a variable out of scope. In some cases, see Section 3.7, the genetic operations can be limited to ensure all mutants compile. GP individuals (patch lists) which cause the code to fail to compile are not permitted to enter the breeding pool of parents of the next generation.

3.3.1 Reducing or avoiding the Cost of Compilation

The major overhead in GI (as with all non-trivial evolutionary programming applications) is the fitness function. In GI the fitness overhead is usually dominated by 1) the compiler overhead and 2) running the program to be improved. There are several ways which might reduce the compiler overhead, some of which have already been demonstrated:

- Use a faster compiler. E.g. replace gcc with tcc.
- Remove (or reduce) compiler optimisation switches, e.g. `-O`.
- Compile only code which GI has changed (c.f. the Unix `make` tool).
- Pre-compile code which cannot be changed, e.g. C `.h` include files.
- Compile more than one mutation together in the same file, see Figure 8.

```
<KStereo.cuh_53>       ::= "OUTYPE *" <optrestrict_KStereo.cuh_52> "disparityPixel,\n"
<KStereo.cuh_78>       ::= "#ifdef LOCAL_disparityPixel\n"
<KStereo.cuh_79>       ::= "float disparityPixel_L[ROWSperTHREAD];\n"
<KStereo.cuh_80>       ::= "#endif /*LOCAL_disparityPixel*/\n"
<KStereo.cuh_85>       ::= "#ifdef SHARED_disparityPixel\n"
<KStereo.cuh_86>       ::= <optvolatile_KStereo.cuh_86> "extern __shared__ OUTYPE disparityPixel_S[];\n"
<optvolatile_KStereo.cuh_86> ::= " volatile "
<KStereo.cuh_88>       ::= <optvolatile_KStereo.cuh_86>
                      "int* const disparityMinSSD = (int*)&disparityPixel_S[ROWSperTHREAD*BLOCK_W];\n"
<KStereo.cuh_89>       ::= <optvolatile_KStereo.cuh_86>
                      "int* const col_ssd = &disparityMinSSD[ROWSperTHREAD*BLOCK_W];\n"
<KStereo.cuh_93>       ::= "#else /*SHARED_disparityPixel */\n"
<KStereo.cuh_95>       ::= <optvolatile_KStereo.cuh_86>
                      "extern __shared__ int disparityMinSSD[];\n"
<KStereo.cuh_96>       ::= <optvolatile_KStereo.cuh_86>
                      "int* const col_ssd = &disparityMinSSD[ROWSperTHREAD*BLOCK_W];\n"
<KStereo.cuh_100>      ::= "#endif /*SHARED_disparityPixel */\n"
<KStereo.cuh_153>      ::= "#ifdef DPER\n"
<KStereo.cuh_154>      ::= " if" <IF_KStereo.cuh_154> " \n"
<IF_KStereo.cuh_154>   ::= "(dblockIdx==0)"
<KStereo.cuh_155>      ::= "#endif /* DPER */\n"
<pragma_K3>            ::= "#pragma unroll 3\n"
<pragma_K11>           ::= "#pragma unroll 11\n"
<_KStereo.cuh_160>     ::= "init_disparityPixel(X,Y,i);"
<KStereo.cuh_161>      ::= "" <_KStereo.cuh_161> "\n"
<_KStereo.cuh_161>     ::= "init_disparityMinSSD(X,Y,i);"
<KStereo.cuh_224>      ::= "" <_KStereo.cuh_224> "\n"
<_KStereo.cuh_224>     ::= "ssd += col_ssd[i+threadIdx.x];"
<KStereo.cuh_262>      ::= <pragma_KStereo.cuh_262>
                      "for(" <for1_KStereo.cuh_262> ";"
                      "OK()&&" <for2_KStereo.cuh_262> ";"
                      <for3_KStereo.cuh_262> ") \n"
<pragma_KStereo.cuh_262> ::= ""
<for1_KStereo.cuh_262> ::= "row = 1"
<for2_KStereo.cuh_262> ::= "row < ROWSperTHREAD && (row+Y < (height+RADIUS_V))"
<for3_KStereo.cuh_262> ::= "row++"
<KStereo.cuh_326>      ::= " if" <IF_KStereo.cuh_326> " \n"
<IF_KStereo.cuh_326>   ::= "(X < width && Y < height)"
<KStereo.cuh_348>      ::= "" <_KStereo.cuh_348> "\n"
<_KStereo.cuh_348>     ::= "__syncthreads();"
```

Fig. 7 Fragments of grammar created from stereoKernel (Section 5). Total 423 rules. The type of a rule is given by the part of its name between the `<` and `_KStereo.cuh`. Untyped rules (e.g. `<KStereo.cuh_53>`) cannot be directly changed by GI. (However `OUTYPE`, like `LOCAL_disparityPixel`, `SHARED_disparityPixel`, `ROWSperTHREAD`, `BLOCK_W`, `DPER`, etc., is a macro which is controlled by a GI parameter, see Section 3.4.) Mutation respects these types. Thus mutation `<IF_KStereo.cuh_326><IF_KStereo.cuh_154>` replaces the contents of the `if` on line 326 with the contents of the `if` on line 154. (The evolved solution is given in Section 5, page 23 onwards and Figure 22 page 24.)

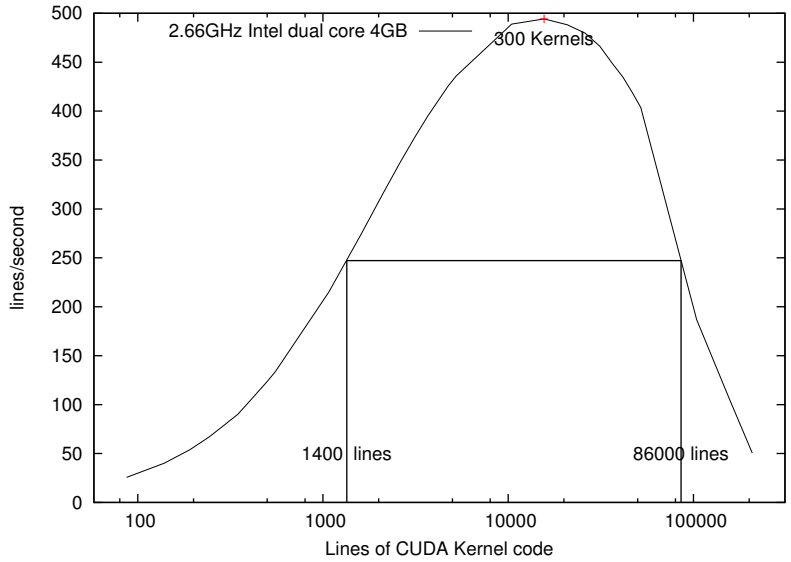

Fig. 8 As expected [Harris, 1997], compiling multiple kernels together is faster than compiling them separately. For the nvcc CUDA compiler (version 5.0 V0.2.1221) and NiftyReg the peak is when compiling 300 kernels together, which is 19.3 times faster than running the compiler once for each. Note log horizontal scale.

- Distribute the compilation across multiple host computers [Harding and Banzhaf, 2009].
- Mutate at lower levels than the source code. E.g. Java byte code [Schuler and Zeller, 2009] or machine code [Schulte *et al.*, 2014b; Schulte *et al.*, 2015].
- Pre-embed every single mutation to the source code, compile once to create one binary executable which has run-time switches to selectively enable/disable each mutation. In mutation analysis (mutation testing) this super mutant [Langdon *et al.*, 2010] is known as a "mutation schemata" [Untch *et al.*, 1993].

Bug repair schemes tend to operate not on the source code but on the program's abstract syntax tree (AST) [Weimer *et al.*, 2010]. By performing only legal changes to the AST the program can be modified quickly in a way that is guaranteed to be syntactically correct. However, like our grammar based approach, most errors are caused by moving variables out of scope (but see Section 3.7).

3.4 Genetic Operations

The genetic operations (i.e. mutation and crossover) act via each program's grammar on its source code (see Section 3.3 and Figure 7). While AST based

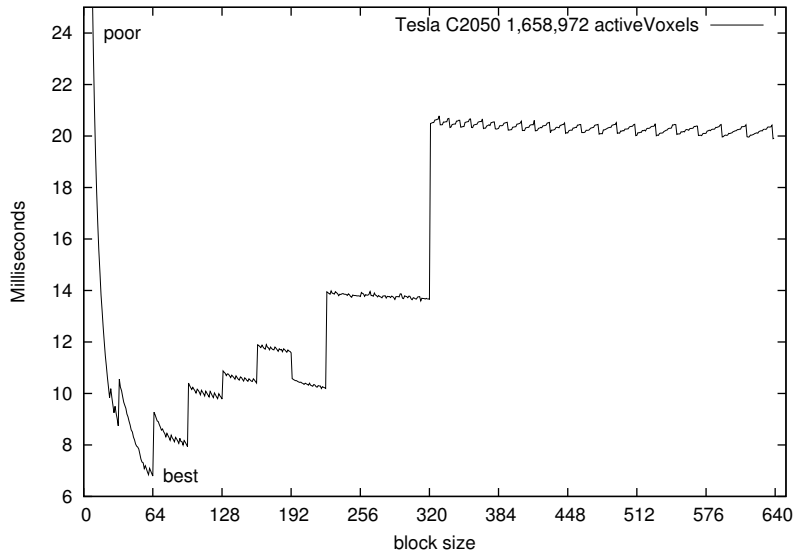

Fig. 9 Ruggedness of performance landscape. Time taken by NiftyReg kernel for different settings of one parameter. Mean of five runs at each parameter setting. GPU timings are highly reproducible. On average (median) the observed standard deviation is 0.01% of the mean, and in more than 90% of cases it is less than 0.1%.

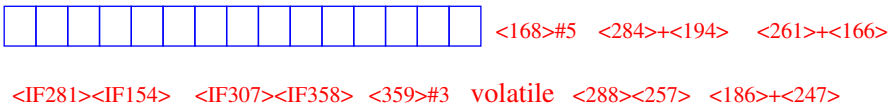

Fig. 10 Extended chromosome. (Used, for example with BarraCUDA, Section 7.) A fixed number of parameters in blue (left). Followed by a variable length list of grammar patches separated by spaces. There is no limit to the number of patches, so the search space is infinite. (To save space in the figure shown on two lines but actually all on a single line.)

approaches [Weimer *et al.*, 2010] typically work at the statement level, grammar based GI modifies lines of code. (Depending on the human programmer, lines of code and C statements are typically similar.) GI can only manipulate the existing code. It does not invent new code. The basic operations are: delete a line of code, replace a line of code with another from elsewhere in the program and insert a copy of another line. Each code change is represented in plain text using source file names and line numbers. A GI individual is then a list of changes, again represented as a simple text string.

In the case of CUDA some control parameters make a huge difference to performance (e.g. see Figure 9, but see also Section 3.5). Therefore, as mentioned on page 7, in some cases, these and various conditional compilation flags are made explicit targets for optimisation. This is done by including in the chromosome a fixed part (like a genetic algorithm) with its own point mutation and uniform crossover operations (see Figures 10, 11, 12 and 13).

<284>+<194> volitile <247><186><180><231><358><154><174>+<176>

<284>+<194> volitile <247><186><180><231><358><154><174>+<176><288>+<161>

Fig. 11 Example of mutation to the variable length part of a GI individual. Patch <288>+<161> is appended to parent (top) extending the child (bottom). This causes a copy of source line 161 to be inserted before line 288 in the kernel source code. (For clarity the fixed part omitted and full grammar rule names simplified to just their line numbers.)

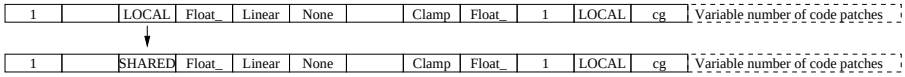

Fig. 12 Example of mutation to the configuration part at the start of a GI individual. In this example and in Figure 13 (all from StereoCamera Section 5) the usual GI variable length patch list is augmented by a binary GA like fixed list of parameters represented in plain text. Top: parent Bottom: offspring.

3.4.1 Mutation

Where configuration parameters are used, half of mutations are made to them. In which case one is chosen uniformly at random and its current value is replaced by another of its possible values again chosen uniformly at random, see Figure 12. The other half of the mutations are made to the code. In which case the mutation operator appends an additional code patch to the parent (see Figure 11).

3.4.2 Crossover

Crossover creates a new GP individual from two different members of the better half (Section 3.2) of the current population. Where fixed parameters are used, the child inherits each of them at random from either parent (uniform crossover [Syswerda, 1989], see Figure 13). Whereas in [Langdon and Harman, 2015b] we used append crossover which deliberately increases the size of the offspring, here, on the variable length part of the genome, we use an analogue of Koza's tree GP crossover [Koza, 1992]. Two crossover points are chosen uniformly at random. The part between the two crossover points of the first parent is replaced by the patches between the two crossover points of the second parent to give a single child. (Cf. two point or double crossover [Cavicchio, Jr, 1970].) On average, this gives no net change in length.

3.5 Software is Not Fragile

Whilst in Section 3.4 we showed the GI (particularly the GI-GPU) search space is in some parts rugged, here we present more evidence why modern search based optimisation can still make progress.

It is often assumed that computer programs are fragile and that any single change will destroy them totally. Figure 14, and also [Schulte *et al.*, 2014b], show this is not true. Even though our mutation operators (Section 3.4.1)

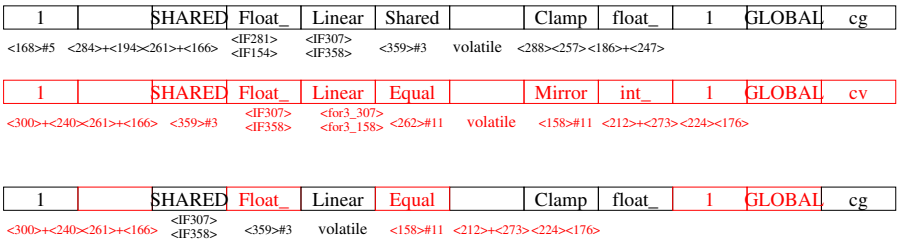

Fig. 13 Example of crossover between GI parents. Parts of two above median parents (top and middle) recombined to yield a child (bottom). (For clarity variable code mutation part shown underneath fixed mutation part and some of the code patches shown stacked to save horizontal space.)

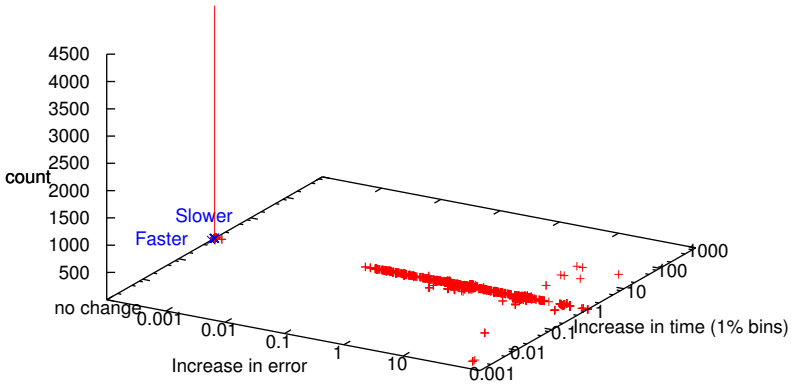

Fig. 14 Impact of all (7079) one change mutations on StereoCamera kernel [Langdon and Harman, 2014b] running on an nVidia GeForce GT 730 graphics card. 2336 (33%) ran but made errors (+). Nine mutants failed to compile due to a bug in the nVidia CUDA 6.0.1 compiler (fixed in CUDA 7.0). 16 caused infinite 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 (×). Note log scales.

are like changes a human programmer might make, Figure 14 shows that about 5 in 8 random changes do not change StereoCamera's (Section 5) CUDA kernel's output on a randomly chosen test image pair. This test means that all 76 800 pixel values are identical. (Cf. equivalent mutants [Yao *et al.*, 2014]).

Figure 15 shows the evolution of the effect of our mutation and crossover operations. Notice we see similar effects: whilst some programs which were working are broken by source code changes, a substantial proportion continue to work.

3.6 Continuing Evolution Despite Compilation Errors

Although apparently tedious, in the case of compiling populations of CUDA kernels it turns out to be relatively easy to ensure the failure of one member of the population to compile does not influence the compilation of others. (This

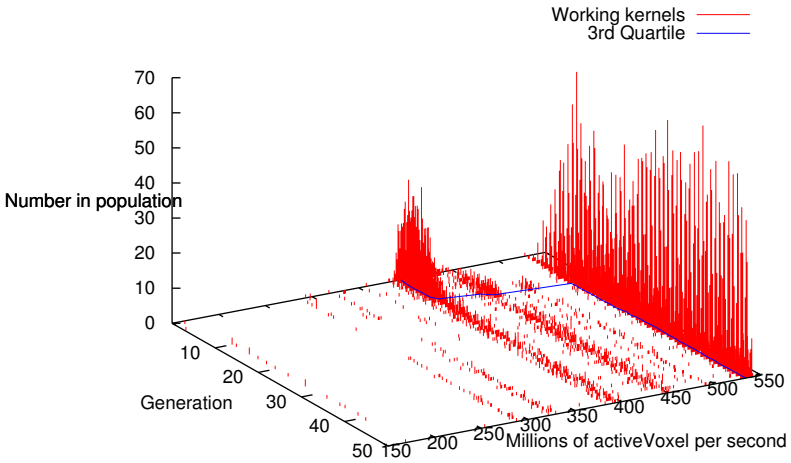

Fig. 15 Evolution of NiftyReg (Section 6) population (300) on GTX 295. (Histogram per generation, million voxel/sec bins). After generation 10, on average 58% of the population process more than half a billion voxels per second. The fraction of the population that produce wrong answers is 48% in the random initial generation, but only 30% on average afterwards (not plotted).

could be done by compiling each member of the population independently but this incurs the overheads mentioned in Section 3.3.1.) A simple script can process the compilation error messages and from the line number where the error is reported infer which GI individual(s) failed to compile. These are excluded from the population and the compilation re-tried.

Typically the compiler will report many errors before stopping and so several failing GI individuals can be excluded at once. However, the GCC compiler has a limit of 100 reported errors, after which it gives up. Therefore, it is necessary each time the population is compiled to check in case there are compilation errors which were present before but were not previously reported. Although repeatedly running the compiler suggests a large overhead, this is not so in practise, as typical errors are reported by the compiler early in its operation. This causes the compiler to stop before it gets to its more expensive operations, like machine code generation. That is, for example, the source code will have to be parsed more than once, but parsing and error checking, are relatively cheap compiler operations.

3.7 Avoiding out of Scope Compilation Errors

Again although apparently tedious, it turns out to be relatively easy when converting CUDA code to a grammar to 1) keep a track of the scope of each variable and 2) use this to provide a list of line numbers where lines of code containing variables may be inserted without those variables violating their scope limitations. This analysis is done once before evolution starts.

In Section 7 the scope analysis is made slightly more complex by the use of conditional compilation `#if` symbols, which cut across C scope. Nevertheless in practice it is entirely feasible to restrict mutations by line number ranges and thus ensure most mutated code compiles.

3.8 Post Evolution Clean up

Evolution's tendency to create overly large solutions has been remarked many times [Tackett, 1994; Angeline, 1994]. Whilst tree based GP has specific geometric reasons for it [Poli *et al.*, 2007] it appears to be inherent in evolution of variable length structures [Langdon and Poli, 1997b; Banzhaf and Langdon, 2002; Soule and Heckendorn, 2002] and as expected bloat also arises in GI patch lists. As with Weimer's bug repair, we include an explicit post evolution phase to remove unnecessary changes.

Typically the best GI individual in the last generation is minimised by starting at its beginning and progressively temporarily removing each individual mutation and comparing the performance of the new kernel with the evolved one. Unless the new kernel is worse the mutation is excluded permanently. Often to encourage removal of mutations with little impact, those that make only a small difference to the kernel timing are also removed.

4 CUDA gzip

The purpose of the gzip experiment is to show evolution can evolve desired functionality using test cases gleaned from existing (human written) code. We chose the `longest_match` C function from the well known gzip Unix compression tool. gzip was written when Unix was young. It works by scanning the file to be compressed for sequences of bytes which occur multiple times and replacing them in the compressed file by a shorter code. The longer the replaced repeated sequences are the better the compression. `longest_match` is given a string of bytes and scans forward up to 64 Kbytes in the file to be compressed looking for the longest sequence of bytes which match its input. `longest_match` is the most computationally demanding part of gzip and in current releases of Unix it is replaced with assembler code. In this first example, a BNF grammar based upon nVidia's own CUDA examples was created by hand and test cases where taken from the Software-artefact Infrastructure Repository (SIR) [Hutchins *et al.*, 1994]. The grammar ensures code that looks like an appropriate CUDA kernel with the right number of inputs is created, whilst the fitness function ensured a replacement for `longest_match` was evolved.

The crosses (+) in Figure 16 show that the distribution of inputs to `longest_match` is highly non-uniform. If our goal was just to optimise the code, it might make sense to concentrate upon the few cases which occur frequently. (Possibly leaving the infrequent cases to be processed by unmodified

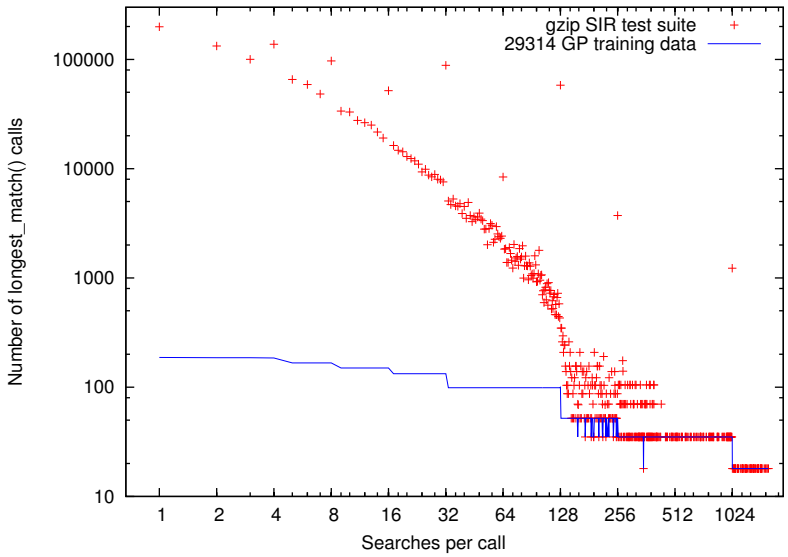

Fig. 16 Distribution of number of strings that gzip searches for each byte compressed. 211 tests from SIR. Total 1 599 028 calls of `longest_match`. Note log scales.

code.) However, the new code must deal correctly with all inputs. (Here we evolve correct but naive kernels that do not provide any speed up.) To avoid evolution excessively concentrating upon the short popular test cases, these occur less frequently in the test cases used by the fitness function. By avoiding excessive duplication 1 599 028 tests were reduced to 29 315 more uniformly spread examples (line in Figure 16). In each generation 100 of these were randomly chosen and used to assess the fitness of every evolved CUDA kernel in the population.

In gzip a complete grammar was created by hand from example code supplied by nVidia. The BNF grammar is coded so that each rule either has no alternatives or there are exactly two alternative productions. Which alternative is to be used is given by the genotype (see Figures 17 and 18).

The right hand side of Figure 18 uses [Daida *et al.*, 2005]'s circular lattice tree format on the final population to illustrate the gzip grammar based GP population converging from the root like tree based GP [Langdon and Poli, 1997a; McPhee and Hopper, 1999; Soule and Heckendorn, 2002; Burke *et al.*, 2004]. (See also `http://www.cs.ucl.ac.uk/staff/W.Langdon/gypse/` and supplementary data.)

In the later experiments, Sections 5 to 8, evolution works by mutating the grammar, rather than deciding which option to take. Hence those grammars are simply a list of rules and productions which, after applying the list of patches, Section 3.3, are expanded in order without the possibility of alternatives.

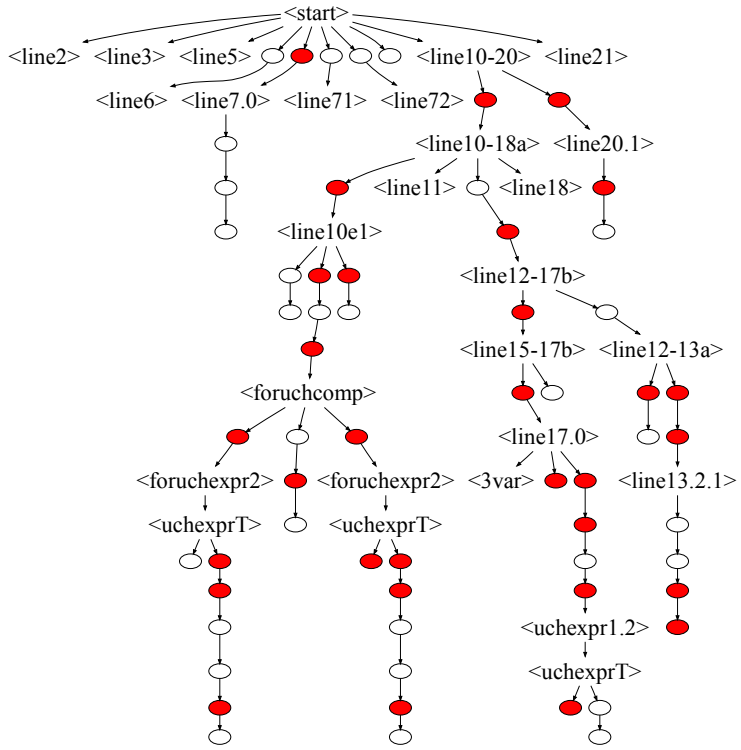

Fig. 17 Path through the grammar taken by GI to create the gzip CUDA kernel evolved in generation 55. Figure 6 (page 10) gave the resulting C++ program. Ovals indicate binary decision rules. With shaded ovals the second option was used.

5 StereoCamera

StereoCamera was written by nVidia's image processing expert to demonstrate the first version of CUDA and nVidia hardware could process stereo image pairs in real time [Stam, 2008]. Nonetheless, as we shall see, it is possible to evolve substantial improvements starting from Stam's code. (In the animation `http://www.cs.ucl.ac.uk/staff/W.Langdon/egp2014/AC_k20c_video.gif` the inferred distance, z, is shown using false colours for the stereo video recording used to train the GI. `AC_k20c_video.gif` is also in the supplementary data.) Figure 19 explains StereoCamera's algorithm using as an example a holdout stereo image pair. With correct positioning of the page before your eyes, it is just about possible to get a three-dimensional effect.

Table 1 gives the speed up for six types of GPUs. In the case of StereoCamera, two key algorithm specific parameters, `ROWSperTHREAD` and `BLOCK_W`,

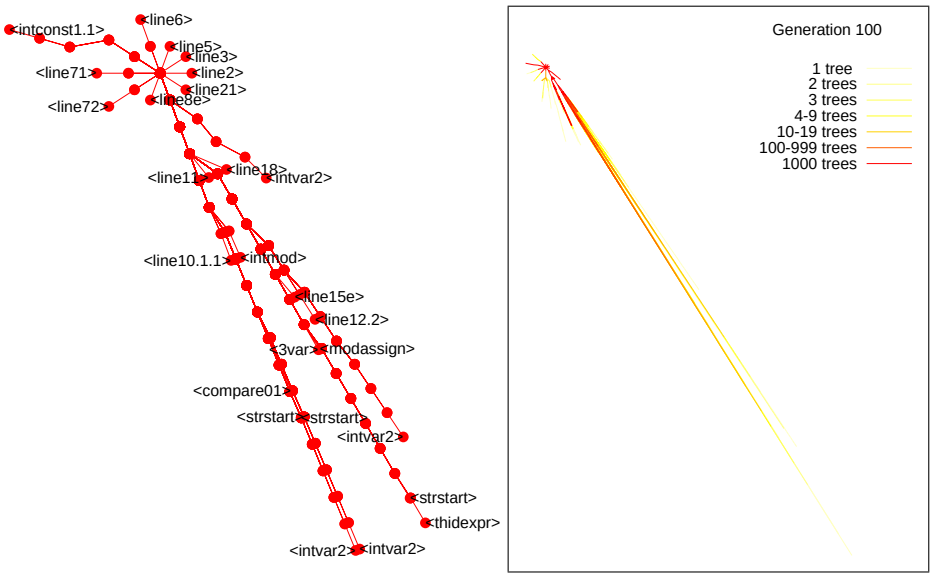

Fig. 18 Left: Grammar expansion for an evolved gzip CUDA C++ kernel. Identical to Figure 17 but displayed as a circular lattice [Poli *et al.*, 2008, p 136] [Daida *et al.*, 2005]. The last rule in each branch of the BNF grammar tree is labelled. Right: Aligning all 1000 trees in the GP population at their roots (red) stresses much of trees near their roots are identical. Unique parts (light) are only far from root.

Fig. 19 Schematic of stereo disparity calculation. Top: left and right stereo images. Bottom: output. For each pixel stereoKernel calculates the sum of squared differences (SSD) between 11×11 regions centred on the pixel in the left image and the same pixel in the right hand image (blue, not to scale). This is the SSD for zero disparity. The right hand 11×11 region is moved one place to the left and new SSD is calculated (SSD for 1 pixel of disparity). This is repeated 50 times. Each time a smaller SSD is found, it is saved. Thus each output pixel's final value is the distance between left and right images which gives the maximum similarity between them (across an 11×11 region). Real time performance is obtained by parallel processing and reducing repeated calculations.

Table 1 Mean speed across all 2516 I2I 320×240 stereo image pairs. ± is standard deviation. Times in microseconds. Tuning NVS 290 *increases* `ROWSperTHREAD` from 40 to 120, otherwise pretuning reduces it to 5. Post evolution tuning leaves `ROWSperTHREAD` as 5, except C2050 (14) and GTX 580 (15).

GPU name	Original	Pretuned	Ratio	GI	Speedup
Quadro NVS 290	27402±116	26019±152			1.053±0.01
GeForce GTX 295	5448± 14	1518± 4			3.589±0.01
Tesla T10	5256± 12	1436± 3	3.661±0.01	1359±38	3.861±0.11
Tesla C2050	4632± 25	3017± 15	1.535±0.01	1130± 5	4.099±0.02
GeForce GTX 580	3077± 21	1650± 6	1.865±0.01	722±29	4.248±0.17
Tesla K20c	4362± 21	1839± 18	2.373±0.03	638± 1	6.837±0.04

Table 2 GPU Hardware. Year each was announced by nVidia in column 2. Third column is CUDA compute capability level. Each GPU chip contains a number of identical and more or less independent multiprocessors (column 4). Each MP contains a number of stream processors (cores, column 5) whose speed is given in column 7. Size of on board memory (column 8) is followed by the measured data rate (ECC on) between the GPU and its on board memory in last column. GTX 295 and Tesla K80 are a dual GPUs, performance figures given for one half.

Name	Announced		MP ×	cores	Clock	Caches		On board memory	
					GHz	L1 KB	L2 MB	GB	GB/s
Quadro NVS 290	2007	1.1	2 ×	8 = 16	0.92	none		0.25	4
GeForce GTX 295	2009	1.3	30 ×	8 = 240	1.24	none		0.87	92
Tesla T10	2009	1.3	30 ×	8 = 240	1.30	none		4.00	72
Tesla C2050	2010	2.0	14 ×	32 = 448	1.15	16/48	0.75	2.62	101
GeForce GTX 580	2010	2.0	16 ×	32 = 512	1.54	16/48	0.75	1.50	161
Tesla K20	2012	3.5	13 ×	192 = 2496	0.71	16/32/48	1.25	5.00	140
Tesla K40	2013	3.5	15 ×	192 = 2880	0.88	16/32/48	1.50	11.00	180
Tesla K80	2014	3.7	13 ×	192 = 2496	0.82	16/32/48	1.50	11.00	138
GT 730	2014	2.1	2 ×	48 = 96	1.40	16/32/48	0.12	4.00	23

were pre-tuned. By reducing `ROWSperTHREAD` from the original 40 to 5, pretuning itself gave considerable speed ups (columns 4-5 in Table 1). Whereas for the NVS 290, tuning `ROWSperTHREAD` increased it from 40 to 120, which gave a small improvement (last columns in Table 1). nVidia's value for `BLOCK_W` (64) proved to be optimal for all six GPUs.

Even using memcheck, but without explicit array index protection, such as using `G_idata` (Section 3.2.2), the older NVS 290 and GTX 295 hardware (Table 2 column 2) would always lock-up before the end a GP run. In Table 1 the "GI" columns for the NVS 290 and GTX 295 rows are blank and the last column refers to the speed up achieved by tuning `ROWSperTHREAD` and `BLOCK_W`.

With the four more modern GPUs, the best individual from the last generation (50) was minimised to remove unneeded mutations, which contributed little to its overall performance, and retuned. Typically this reduced the number of changes needed by about half (T10 31→14, C2050 17→10, GTX580 26→13 and K20 29→10). The speeds of the re-tuned kernels are given in Table 1 under heading "GI". In each case this gave a significant speed up (last

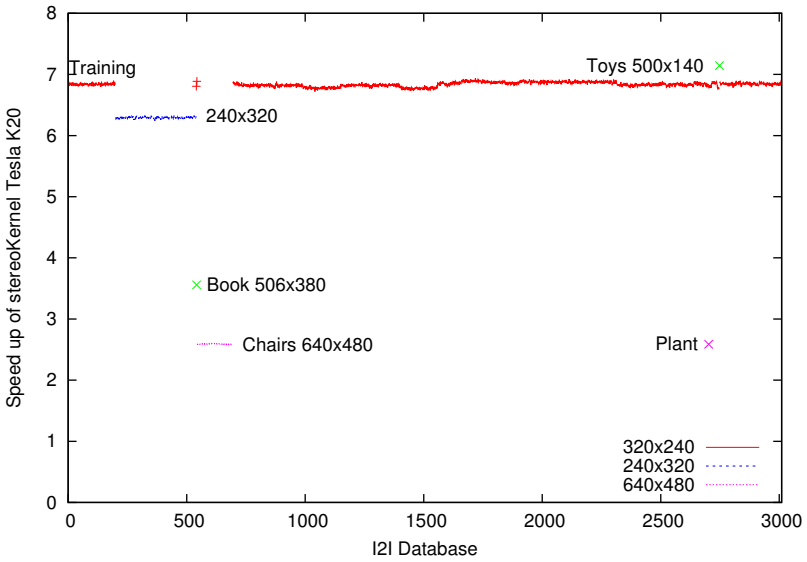

Fig. 20 Performance of GI improved K20 Tesla kernel on all 3010 stereo pairs in Microsoft's I2I database relative to original kernel on the same image pair on the same GPU. Fifty of first 200 pairs used in training. The evolved kernel always gives the same answer and is always much better, especially on images of the same size and shape as it was trained on.

```
DPER=1 STORE_disparityMinSSD=SHARED XHALO=1 STORE_disparityPixel=SHARED
<pragma_KStereo.cuh_359><pragma_K3> <_KStereo.cuh_161>+<_KStereo.cuh_224>
<_KStereo.cuh_348> <optvolatile_KStereo.cuh_86> <pragma_KStereo.cuh_262><pragma_K11>
<IF_KStereo.cuh_326><IF_KStereo.cuh_154>
```

Fig. 21 Best stereoKernel individual in generation 50 of K20 Tesla run after minimising, Section 3.8, removed less useful components. (Auto-tuning made no further improvements.) Some of the grammar rules are given in Figure 7 (page 12).

column of Table 1) compared to both the original kernel and the original kernel with the best `ROWSperTHREAD` setting. The speedup of the improved K20 kernel on all of the I2I stereo images is given in Figure 20. The speed up for the other five GPUs varies in a similar way to the K20. Across images of the same size and shape as the training data performance is very consistent.

The rest of this section describes in detail the evolved code changes. (The busy reader may skip to Section 6.) For brevity we describe in detail only one of the evolved CUDA K20 stereo kernels. The best of generation 50 individual changes 6 of the 12 fixed configuration parameters [Langdon and Harman, 2014b, Tab. 2] and includes 23 grammar rule changes. (Parts of the grammar were given in Figure 7 page 12.) After removing less useful components (Section 3.8) four configuration parameters were changed and there were six code changes. See Figures 21 and 22.

The four configuration changes are: 1) DPER is enabled. (Thus the new kernel calculates two disparity values in parallel.) 2) disparityPixel and 3) disparityMinSSD are now stored in fast on chip shared memory. 4) XHALO is

Original code	New code
`int * __restrict__ disparityMinSSD,`	*kernel argument* `disparityMinSSD` *deleted*
`volatile extern __attribute__((shared))`	`extern __attribute__((shared))`
`int col_ssd[];`	`int col_ssd[];`
`volatile int* const reduce_ssd =`	`int* const reduce_ssd =`
`&col_ssd[(64 )*2 -64];`	`&col_ssd[(64 )*2 -64];`
line inserted	`#pragma unroll 11`
`if(X < width &&' Y < height)`	`if(dblockIdx==0)`
__syncthreads();	*line deleted*
line inserted	`#pragma unroll 3`

Fig. 22 Evolved changes to K20 Tesla StereoKernel. (Produced by GI grammar changes in Figure 21). For brevity, except for the kernel's arguments, disparityPixel and disparityMinSSD changes from global to shared memory are omitted.

enabled, meaning the code spreads the work more uniformly over more parallel threads.

The final code changes, Figure 22, are:

- disable volatile, potentially allowing more compilation optimisations.
- insert `#pragma unroll 11` before `for` loop on line 262.
- insert `#pragma unroll 3` before another `for` loop on line 359.
- Mutation `<_KStereo.cuh_161>+<_KStereo.cuh_224>` causes line 224 to be inserted before line 161.
- Mutation `<IF_KStereo.cuh_326><IF_KStereo.cuh_154>` replaces `X < width && Y < height` by `dblockIdx==0`. This replaces a complicated expression by a simpler (and so presumably faster) expression, which itself has no effect on the logic since both are always true. In fact, given the way `if(dblockIdx==0)` is nested inside another `if`, the compiler may optimise it away entirely. I.e. evolution has found a way of improving the GPU kernel by removing a redundant expression.
  The original purpose of `if(X < width && Y < height)` was to guard against reading outside array bounds when calculating SSD. However, the array index is also guarded by `i < blockDim.x`
- delete `__syncthreads()` on line 348. `__syncthreads()` forces all threads to stop and wait until all reach it. Line 348 is at the end of code which may update (with the smaller of two disparities values) shared variables disparityPixel and disparityMinSSD. In effect evolution has discovered it is safe to let other threads proceed since they will not use the same shared variables before meeting other `__syncthreads()` elsewhere in the code. Removing synchronisation calls potentially allows greater overlapping of computation and I/O leading to an overall saving.

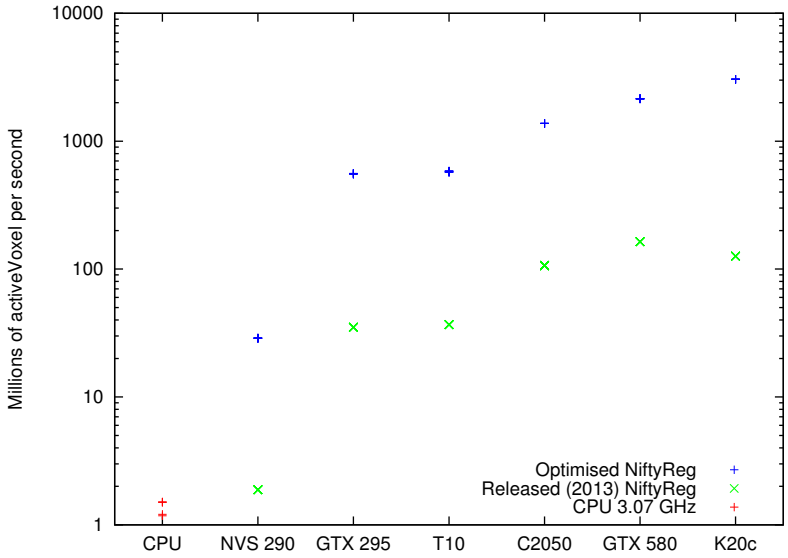

Fig. 23 Performance of modified NiftyReg 3D registration CUDA kernel `reg_spline_getDeformationField3D` after optimisation by evolution, bloat removal and with optimal block size and `-arch`. (GPU speeds for 10 holdout images lie almost exactly on top of each other.) Note log vertical scale.

6 Three Dimensional Medical Imaging

NiftyReg [Modat *et al.*, 2010] is a comprehensive 2D and 3D medical imaging package. A common requirement is to match patient images either initially against a reference image or during or following treatment against earlier images for the same patient. This matching process is known as image registration and NiftyReg provides sophisticated algorithms for doing this. In order to demonstrate GI we chose a computationally demanding task of registering 3D NMR brain scans. At typical millimetre resolution, each scan contains about 10 million voxels arranged in a cube spanning the patient's head, although only about 10–15% cover the brain. For offline analysis parallel cloud or cluster computing can be considered, however, they are unsuitable for clinical use, whereas GPUs offer the possibility of in-theatre use since they can be installed close to the brain surgeon and have excellent real-time stable response characteristics. With this in mind NiftyReg had been ported to CUDA, the GPU giving up to a 15 fold speed up (depending on GPU, see data plotted with $\times$ in Figure 23).

Since the CUDA kernel is used as part of an iterative gradient method each active voxel must be calculated exactly. That is, there is no scope for trading precision for speed. In the GI approach, the values calculated for each voxel by the mutated CUDA code is compared with the answer produced by the existing serial code. Despite the computational power of the GPU, the hand optimised CUDA kernel could still take up to 70% of the elapsed time.

The expertly optimised 3D registration CUDA kernel was used as the feedstock by GI, which was run on six different GPUs. Their performance on holdout images (i.e. not used during training) is shown in Figure 23 (note log y-axis). In each case GI, using evolution and manually coded options, was able to improve the expert hand written code by more than an order of magnitude [Langdon *et al.*, 2014].

7 Improving BarraCUDA DNA Alignment

BarraCUDA [Klus *et al.*, 2012] is a state of the art C++ program which maps short DNA fragments against a reference genome. Aligning (i.e. mapping) short DNA sequences is the first step to assembling a complete DNA sequence for the organism. (Typically a human patient but the process and indeed the software, applies to any living organism. E.g. bacteria, fly or cabbage). Once aligned interesting variations (mutations) in the individual can be found.

High end next generation sequence (NextGen, NGS) DNA scanners can produce in the region of billions of short DNA sequences per day. This is almost as fast as the alignment software can process them and typically parallel processing is essential to keep up.

We use DNA strings generated by The 1000 Genomes Project [Durbin and others, 2010], which has mapped all common human genetic variation, and alignment against the human reference genome [International Human Genome Sequencing Consortium, 2001] to train an improved version of BarraCUDA [Langdon *et al.*, 2015].

We concentrate upon the "aln" operation in BarraCUDA, which is used to align short DNA sequences (see Figure 24). BarraCUDA's speed comes from the GPU's ability to process hundreds of thousand of DNA queries in parallel. Typically a buffer full (16 MBytes) of query strings are processed in parallel and written to the default output stream `stdout` before the next group are read from disk.

Off-line BarraCUDA is used to convert the reference genome stored as plain text strings into a prefix table. A prefix table gives the location of every string in the reference which starts with a particular series of DNA characters. If the query sequence is short it may occur many times. Longer strings tend to occur less frequently. Some (even modest length) queries do not occur anywhere in the reference. This happens frequently with real DNA queries due to either 1) mutations which are not in the reference genome and 2) noise. Although long (64 bit) addresses are used to record match locations the prefix table still gives some compression as many strings are repeated many times in natural genomes.

The previous version of BarraCUDA divided the DNA query strings into substrings of at least 32 bytes. So a typical query of 100 base pairs would be split into three roughly equal substrings. These (depending upon the results of the last query) are processed sequentially by the `split_inexact_match_caller` CUDA kernel (see Figure 24).

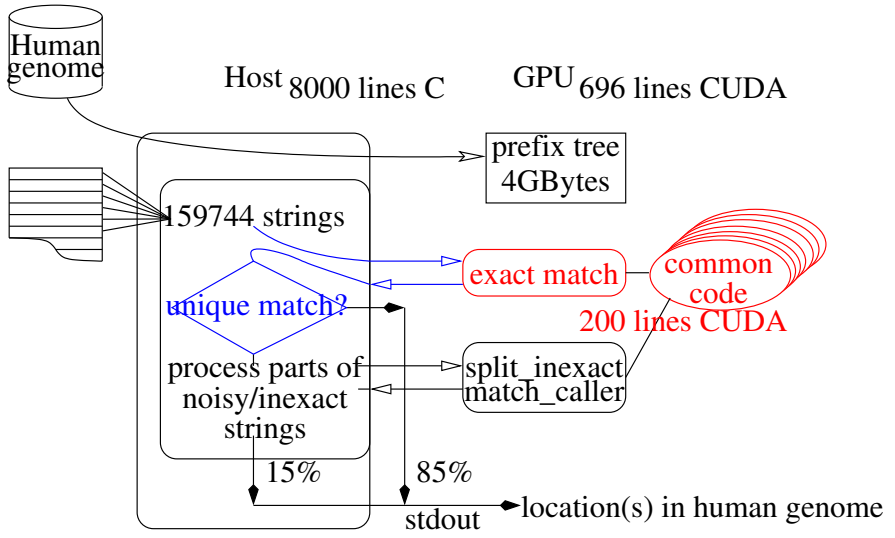

Fig. 24 BarraCUDA reads the compressed reference genome from disk and stores it in the GPU's on board memory. Up to 159 744 DNA query strings are processed in parallel by the `exact_match` kernel on the GPU. The $\approx 15\%$ that do not match uniquely are passed to the original `split_inexact_match_caller` kernel. The principal manual change is to call `exact_match` kernel. `exact_match` and common routines are optimised by evolution.

`split_inexact_match_caller` is complicated both internally and externally. The external complication is dealt with by sequential BarraCUDA code on the host, which must keep track of all the multiple potential matches found for the earlier sections of each DNA query sequence (or indeed when no potential matches were found). Where multiple potential matches are reported by `split_inexact_match_caller` the host code must decide which are worth exploring further. As in `split_inexact_match_caller` itself, there are complicated heuristics to trade quality of partial matches against both run-time and the size of data structures used to store them.

Internally every parallel thread in `split_inexact_match_caller` has to keep track of multiple potential matches. `split_inexact_match_caller` uses a depth first strategy so that the thread keeps exploring the current matches until either it reaches the end of the query sequence or it can find no exact matches in the prefix table. If a thread is told its query string does not exist in the reference genome, it backtracks (again using heuristics) to a suitable branch point and where it tries the next of the three other possible DNA bases and scans forwards again. Note the heavy use of the forward scan operation (part of the "common code" in Figure 24) and that although initially in step, each thread acts independently and may quickly diverge from the operation of its neighbours. (SIMT, Section 1, does not permit threads to do different things simultaneously. Thus divergence forces some threads to be inactive until all those in the current "warp" resynchronise.)

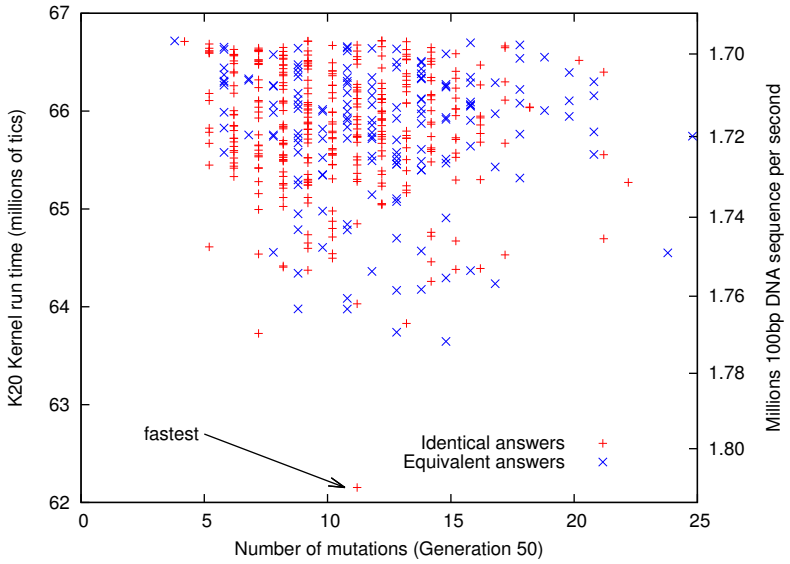

Fig. 25 Distribution of speed and number of changes in top 500 correct Tesla K20 `cuda_find_exact_matches` kernels in final BarraCUDA GP population.

Although the quality of the data is variable, typically about 85% of DNA queries match exactly once in the reference genome. In the GI version of BarraCUDA the `exact_match` kernel is used. It is much simpler than `split_inexact_match_caller` in that it does not attempt to back track (and so cause threads to diverge) and it is used to process the whole of each query string (rather than chunks of about 32). The remaining 15% of strings are dealt with by the previous code. Note `split_inexact_match_caller` also benefits from the GI optimisations to the common code.

After optimisation a single Tesla K20 can process almost two million DNA queries per second, Figure 25. This is more than 100 times faster than the BarraCUDA's original speed but this rate excludes slow host and `split_inexact_match_caller` operations. Since each base in each DNA string must be looked up in the prefix table and this will require reading at least 64 bytes of global memory (half a K20 cache line) the fastest kernel (Figure 25) is reading at least $1.8\ 10^6 \times 100 \times 64 = 1.152\ 10^{10}$ bytes per second (11GB/sec). This compares poorly with the K20's maximum bandwidth, 140GB/sec, see Table 2, suggesting there is still scope for improvement.

8 RNA Binding Energy, CUDA Dynamic Programming

RNA folding is one of several Bioinformatics problems that can be solved using Dynamic Programming [Liu *et al.*, 2006; Manavski and Valle, 2008; Luo *et al.*, 2013]. RNA is a linear biological polymer whose monomer compo-

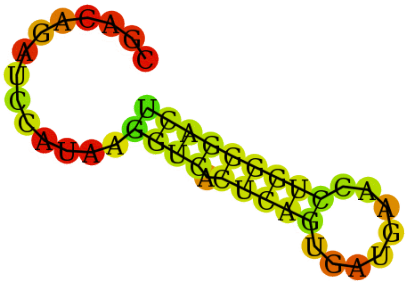

Fig. 26 A fold of RNA sequence `CGACAGAUCCAUAAGGUCACUCAGUGAUGAACCUGGGGACU`

nents bind tightly along the polymer's chain but also bind weakly with each other sideways, see Figure 26. How a RNA molecule folds up can be predicted by calculating the maximum self interaction energy. In the Dynamic Programming approach the total energy is the sum of local interactions, i.e. the interaction between chain link i and another link j. The folding configuration which gives the strongest self-binding is the physical shape of the RNA molecule. A symmetric $n+1 \times n+1$ square matrix is used to keep track of energy calculations, see Figure 27. n is the length of the RNA molecule, typically between 20 and 1000. (In the CompaRNA version of RNAstrand[3] the most popular length is 76 and the median 280 [Langdon and Harman, 2015a, Fig. 2]). Dynamic Programming starts in a corner of the matrix and then expands in a diagonal fashion. Thus in the second step, two elements of the matrix are calculated. These are independent and so can be done in parallel. At the third step, three elements of the matrix can be processed in parallel. And so on. Thus potentially Dynamic Programming is well suited to parallel computation such as provided by GPUs. However, the performance of the CUDA version of pknotsRG [Reeder *et al.*, 2007] was disappointing, simply because the Dynamic Programming matrices (and hence the degree of parallelism) is far too small to keep a GPU busy.

The original version of pknotsRG calculates the shape of one RNA molecule at a time. As a proof of concept a Grow and Graft Genetic Programming (GGGP) [Harman *et al.*, 2014; Jia *et al.*, 2015] approach was taken whereby the host code was manually changed to request the GPU process all the RNA molecules in parallel. See right hand side of Figure 27. Evolution automatically grew a small bit of CUDA code which was inserted into the existing kernel. This graft transformed the kernel so that instead of dealing with one Dynamic Programming matrix at a time it could process (depending upon n) hundreds of thousands of Dynamic Programming matrices in parallel. In the best case this gave a speed up on 10 000 fold, Figure 28.

[3] `http://iimcb.genesilico.pl/comparna/site_media/entire_datasets/rnastrand.zip` 163.3MB down loaded 3 Apr 2015

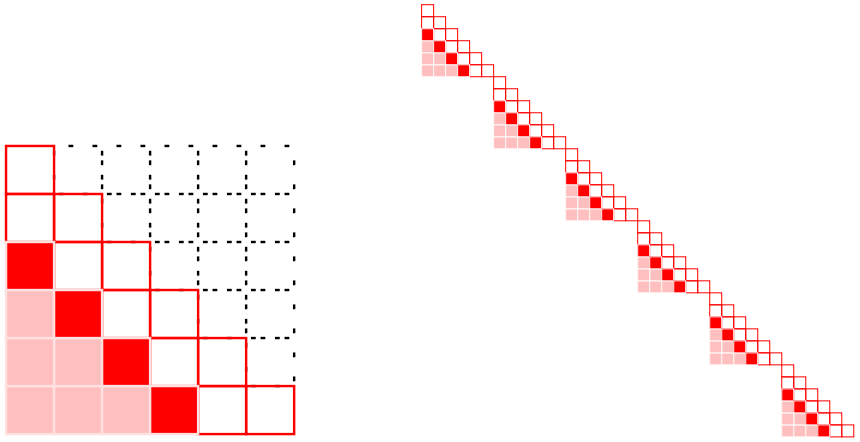

Fig. 27 Left: $(n+1) \times (n+1)$ Dynamic Programming matrix. Only lower half is used. Active front can be calculated in parallel using from 1 to $n+1$ threads. (n is length of RNA molecule.) Right: GPU allows many Dynamic Programming matrices to be calculated in parallel.

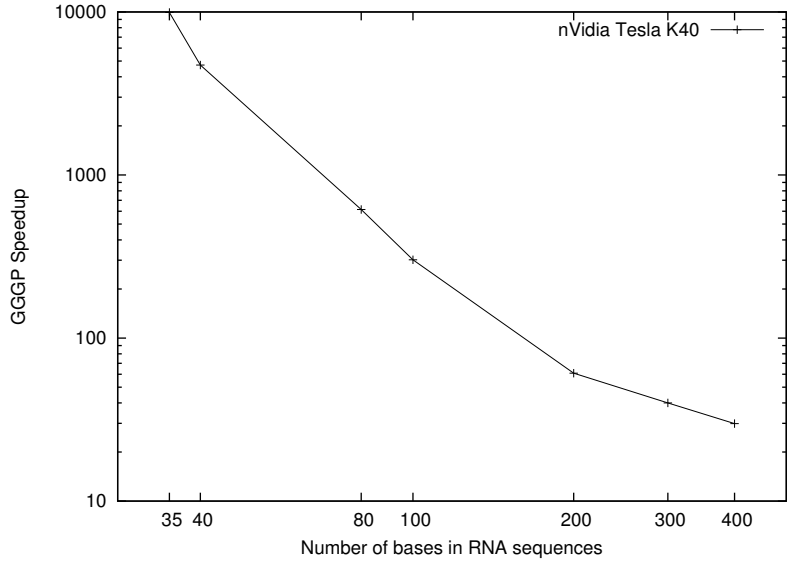

Fig. 28 Ratio between original speed of CUDA version of pknotsRG and CUDA version after grow and graft change to allow processing multiple sequences in parallel for different RNA lengths. Note log scales.

Potentially this transforms computational modelling of RNA. Instead of dealing with RNA molecules one at a time, complete populations of millions of RNA molecules might be simulated.

9 Discussion

9.1 Controlling Mutations via Grammar v. AST and other Representations

In Sections 4 to 8 a BNF grammar was used to describe the program code being modified and the possible mutations to it. The grammar approach has been widely adopted both for GPU and non-GPU work (e.g. [Langdon and Harman, 2015b; Petke *et al.*, 2014b; Bruce *et al.*, 2015]).

In some earlier work which evolved small programs from scratch Koza GP Lisp like S-expression trees were used, e.g. [White *et al.*, 2008]. In the case of Java [Orlov and Sipper, 2011, Section II.B] argue strongly against a grammar approach and instead suggest that since the Java compiler has already taken care of many syntactic and semantic constraints by the time it generates the Java byte code, it makes sense to evolve the byte code directly (see also [Lukschandl *et al.*, 1998; Klahold *et al.*, 1998; Harvey *et al.*, 1998]).

Genetic Improvement work at the machine code level has considered Intel x86, ARM [Schulte *et al.*, 2013] and MIPS RISC [Schulte *et al.*, 2015] hardware. Typically genetic operations (delete, insert, swap and crossover) only make use of knowledge about the lengths of machine code instructions. For example, where mutated machine op-codes have different lengths single byte nops are used to minimise disruption. E.g. to avoiding changing the addresses of the targets of jumps. Where this is unavoidable, [Schulte *et al.*, 2013, page 320] says the mutants typically have reduced fitness. This problem is avoided with MIPS, since the reduced instruction set computing op-codes are all exactly one word long [Schulte *et al.*, 2015].

Much of the GI work on automatic bug repair, like our grammar approach, works at the source-level [Le Goues *et al.*, 2012b]. Rather than representing the program via a grammar, typically CIL is used to analyse the source code and represent it as an Abstract Syntax Tree (AST). GP then operates on the AST at the statement level to evolve a list of changes to the AST. Fault localisation heuristics are used to concentrate changes where they are likely to be effective. To evaluate a GP individual's fitness the patch list is applied to the original program's AST and the modified AST is then converted to source code, compiled and the resulting program run on a few (typically less than five) test cases. Alternatively the LLVM compiler may be used to convert source code into its intermediate representation (IR) which is then evolved like the AST produced by CIL [Schulte *et al.*, 2015].

Since the AST approach inspired our grammar approach, it is natural they should be similar. Notice both manipulate existing human code. Although possible in both, in neither case has it proved necessary to add the ability to generate entirely new code. In both cases almost all failures to compile are caused by variables being out of scope. [Le Goues *et al.*, 2012b, page 961] suggests using *semantic* checks to avoid such compilation errors. This appears to be similar to the scoping checks described in Section 3.7. The bug fixing work typically operates at the statement level, whereas the grammar approach modifies lines of code. Due to the way C programmers code, these are often the

same. Since human programmers care a great deal about the layout of their source code, it is conceivable that working with their layout might possibly sometimes give an advantage. In Sections 4 to 8 additional heuristics were not used to direct genetic operations [Langdon, 1995], however performance profiling has been used elsewhere, e.g. [Langdon and Harman, 2015b]. Although the CIL approach is apparently more elegant, the grammar approach has the pragmatic advantage of working with plain text files, rather than binary data structures, thus all operations, including genetic operations, are readily visible.

9.2 Application of Genetic Improvement outside GPGPU

We have concentrated upon using evolution to improve manually written general purpose GPU software. It is clear that evolutionary computing can be very widely applied to software engineering [Harman *et al.*, 2012b]. In particular genetic improvement [Langdon and Petke, 2016] has been applied to sequential C/C++ code and in some cases substantial speed ups have been found [Langdon and Harman, 2015b] and in other cases better programs [Petke *et al.*, 2014b] [4] or programs that take less energy [Bruce *et al.*, 2015] or less memory [Wu *et al.*, 2015] have been evolved. In addition to the foundational work on bug fixing (e.g. [Le Goues *et al.*, 2012a]), genetic improvement has been used whilst automatically transplanting code from one application to another [Barr *et al.*, 2015], it can reduce the amount of code [Landsborough *et al.*, 2015] and has been demonstrated in embedded systems [Schulte *et al.*, 2015; Yeboah-Antwi and Baudry, 2015; Burles *et al.*, 2015b]. Future areas might include improving software product lines [Lopez-Herrejon and Linsbauer, 2015] or approximate computing [Mrazek *et al.*, 2015]. These different authors have made different design choices. It is a strength of evolutionary computing that such diverse approaches are successful, however it cannot claim to be optimal. Indeed it seems that the GI representation, operators and selection parameters made in the previous sections are unlikely to be optimal.

9.3 Manual Effort

Our research has shown the automatic evolution of improved code. It is difficult to quantify precisely how much manual effort was needed, particularly to give the exact balance between coding and research. However, in the first three experiments, gzip, StereoCamera and NiftyReg (Sections 4, 5 and 6), genetic programming was used with little manual intervention. For example, only in the first, the evolution of the CUDA version of gzip, was the grammar hand made. In the later sections on BarraCUDA (Section 7) and particularly on the 10 000 fold speed up found with grow and graft GP (Section 8), evolution was used to both tune manually inserted hooks as well the original code, or to evolve code at manually designated places. It is difficult to predict how large

[4] [Petke *et al.*, 2014b] was subsequently judged human competitive [Petke *et al.*, 2014a].

an improvement might be evolved. Obviously it depends upon the existing software but also on the existing or future hardware. As well as fully automatic evolution, we also have great hopes for the man with machine synergy [Harman *et al.*, 2012a], as Section 8 starts to demonstrate.

9.4 Using Genetic Improvement with other Optimization Techniques

Although genetic improvement has been demonstrated at lower levels (i.e. machine code, assembly code and Java byte code) a potential advantage of evolving the source code is access to high level tools, particularly the compiler but also other automated optimising tools. For example, the GI version of Bowtie2 [Langdon and Harman, 2015b] was compiled with -O2. In all of the GPU programs we have used nVidia's optimising compiler, nvcc. In principle, other techniques, including commercial tools, could be used inconjunction with genetic improvement. Whilst such tools could be used as the population evolves, they may be computationally expensive. Therefore, for practical reasons it may be that their use is deferred until after the evolution has finished, when they might be just applied to the best of run individual, perhaps after it has been cleaned up and unessential mutations removed (Section 3.8).

9.5 Applying GI to a new GPU Application

If we are starting from a CPU program, the first issue is does a GPU version make sense. Is it possible that any GPU version would be practical? Is there enough parallelism in the task to make it possible to split it into hundreds of thousands of threads? Will those threads be more or less independent? How much data will need to be placed on the GPU? Does your GPU have space for it? How long will it take to transfer it between the host computer and the GPU? At what speed will that data be needed by the GPU processing cores? [Langdon, 2012]. Arithmetic intensity is the ratio of instructions performed per data item moved. Typically it is measured in floating point operations, (FLOPs) per byte, F/B. If F/B is much less than one then using GPUs can still make sense but the GPU code may be bandwidth limited and so you will not get close to the GPUs' peak performance.

If starting with GPU code, are there key parameters which can be optimised without coding changes? Typically the first one to consider is the number of threads per block. Figure 9 gives an example where all possible values were tried. However typically it is sufficient to try just multiples and sub-multiples of 32. The best value can depend on the GPU used. With more than one parameter the number of sensible options to try rapidly explodes and so you should start to consider using a heuristic search, i.e. to use GI on the parameters. (For example Table 1 shows the effect of tuning both CUDA block size and the algorithm specific `ROWSperTHREAD` parameter.) Figure 9 shows the parameter space may be rugged and we would expect interactions

between key parameters, suggesting simple hill climbing may not be an effective search strategy. This lead us to try a genetic algorithm approach with GI (see Figure 10). Although potentially not giving the full advantages of GI, GPU parameter changes may be easier to integrate into existing systems and may need less testing and be more readily accepted.

A halfway house is to hand code alternatives and let evolution choose between them. For example, in Section 7, a cache of recently used bytes had been implemented. You might allow different sizes for this cache and then request evolution to find the best one. (In this case evolution discovered it could dispense with the cache all together, Section 3.1.) On a GPU data and threads interact in surprising ways. Optimising one part of the code can interact with other parts, leading its performance to degrade. Even with just a few optimisation targets, these interactions can mean optimisation by hand can rapidly lead to confusion and disappointment.

9.6 General Lessons

The previous sections have shown that the general GI finding that existing code can be used as its own *de-facto* specification can also be applied to GPU code. Section 4 showed, at least for modest, but non-trivial, amounts of code it may be possible to use tradition serial (i.e. non-GPU) code as the specification for GPU code and to use it to evolve suitable new GPU code. Sections 5 to 8 used running existing GPU code to define via test cases both what the evolved GPU code should do and set a minimum performance level it should exceed. In principle, an unlimited number of test cases can be generated and the results of running the new GPU code on them can be automatically compared with the results of running the original code (either GPU or serial) on them.

The message on testing is mixed. When inducing functionality from test cases, it seems a small number of frequently replaced tests which cover the desired functionality of the new code may be sufficient. In general using fewer tests will reduce fitness testing effort and so speed evolution. In the absence of regularisation, rapid changing of test cases may avoid overfitting. Rapid random turn over appears to be effective and more sophisticated techniques, such as [Gathercole and Ross, 1994; Foster, 2001], may not be needed. Whereas to optimise code it may be better to have many test cases which emulate the expected operational load. The natural distribution of test cases may be highly non-uniform and so may not be suitable for learning all the functionality. That is, there is a tension between learning and doing well on common cases. One can even imagine a GI system, a bit like Section 7, which was optimised for the common cases but left the remaining low frequency exceptions to the original (i.e. unoptimised) code.

Although the NiftyReg package contains many kernels, Section 6 concentrates upon one, albeit a computationally demanding one. In future it would be good to show GI being applied in bulk. That is, mass production still needs to be demonstrated.

In BarraCUDA, Section 7, we have the first use of GI optimised code. It could also serve as a platform for further experiments. BarraCUDA and BWA's code contains many hand-coded heuristics which curtail the search for matching sections of the human genome or direct it in specific directions. These are complex. They were originally developed for BWA running on serial computers. It may be they need to be retuned for traditional CPUs, CPU vector instructions or for modern GPUs. Also, with the advent of higher speed computers and higher speed DNA scanners, the original compromises between speed and quality of solution may need adjusting [Harman *et al.*, 2012a]. There might be scope for using GI on them. Potentially GI could be used as a hyperheuristics to re-train these hand-coded embedded heuristics.

The 10000 fold speed up of pknotsRG found in Section 8, makes it possible to consider the conformations of millions of short RNA molecules. This should have enabled the GPU version to become the dominant implementation of RNA folding tools. However the base code is no longer supported and is not compatible with more recent implementations.

10 Conclusions

It has been known for sometime that Genetic Programming can automatically create small but non-trivial programs and functions. E.g. good hashing algorithms [Hussain and Malliaris, 2000], cache management [Paterson and Livesey, 1997], garbage collection [Risco-Martin *et al.*, 2010] and pseudo random numbers [White *et al.*, 2008]. Indeed large systems can be evolved from components, for example, via web service composition [Rodriguez-Mier *et al.*, 2010] or selectively linking object files [Foster and Somayaji, 2010]. With the advent of Genetic Improvement [White *et al.*, 2011; Langdon, 2015b] we have seen that it can automatically repair bugs [Arcuri and Yao, 2008; Weimer *et al.*, 2009] and find considerable performance advantages [Langdon and Harman, 2015b] in substantial programs. GI is now being demonstrated to improve functional [Petke *et al.*, 2014b] and non-functional properties of existing applications, including reduced energy consumption [Bruce, 2015] and reduced memory use [Wu *et al.*, 2015].

As we showed in Section 3.5, software is not fragile in the sense that any change will break it entirely. Rather although there may be some such fatal changes but if we are prepared to take a population approach, some of the mutated population will continue to work and indeed substantial improvements may evolve.

Effective programming of parallel computers has long been recognised as being hard for people. The economic approach has traditionally been to balance scare expert programmers' time against the cost of the hardware. GPGPU programming suffers twice over under this rule. Firstly GPGPU is rightly regarded as hard and there are few expert CUDA programmers. And secondly, the rational for GPGPU was the falling cost of GPU hardware. Consequently the economic thing is to knock up a quick CUDA kernel which calculates the

right answer but not spend programmer time optimising it. It is now becoming more common to leave discovery of the best setting for a key CUDA parameter to automated optimisation [Reguly and Giles, 2012]. Using modern SBSE [Harman and Jones, 2001] techniques, such as Genetic Programming, this can be readily expanded to optimise multiple parameters. However key choices such as data location and layout are usually burnt into the source code and thus not subject to optimisation. Although [Sitthi-amorn *et al.*, 2011] have shown regular elements in the source code, such as for loops, may be optimised. Here, as well as undirected automatic code modification, we have used manual coding to expose many more aspects of the source code (e.g. data type, data width, data location, texture and cache configuration) to automated optimisation. Notice the goal of the manual code is only that the kernel calculates correct answers, we leave the problem of efficient coding to Genetic Improvement. Indeed we have shown GI can efficiently tune CUDA source code dedicated to a wide range of diverse GPUs.

In Sections 4 to 8 we concentrated upon evolving improvements to existing GPGPU applications. Firstly demonstrating porting sequential legacy code to GPGPU, and secondly obtaining speedups in some cases from 7 times to 10^4 fold. Whilst initially we showed GP working in a traditional high A-to-I ratio mode with no human help, in the later sections, GI is used with manual coding. This culminated in Section 8, where we demonstrate the GGGP approach, in which evolution was directed to where changes might be needed and achieved a speedup of up to ten thousand fold. Whilst BarraCUDA, described in Section 7, has been downloaded more than a thousand times, has been incorporated into BioBuilds by Lab7 and has been ported by IBM to their power 8 super computer range.

Acknowledgements I would like to thank Yue Jia and txbob.
Teslas donated by nVidia.

A FTP kits

A.1 StereoCamera

The grammar-based genetic programming system is available via `ftp.cs.ucl.ac.uk` file `genetic/gp-code/StereoCamera_1_1.tar.gz` and training images are in `StereoImages.tar.gz` The GI version of StereoCamera is in `StereoCamera_v1_1c.zip` [Langdon and Harman, 2014a].

A.2 NiftyReg

Code etc. in `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/niftycuda.tar.gz`

The improved NiftyReg code is in `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/nifty_reg-1.3.9_patch.tar.gz` [Langdon *et al.*, 2014].

A.3 BarraCUDA

GI tools for BarraCUDA are available via FTP and `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/barracuda_gp.tar.gz`

The latest release of BarraCUDA can be downloaded from `http://sourceforge.net/projects/seqbarracuda/?source=typ_redirect` [Langdon *et al.*, 2015].

A.4 pknotsRG

The Genetic Improvement system which evolved the better version of pknotsRG is available via `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/pknotsGI.tar.gz` [Langdon and Harman, 2015a].

A.5 Drawing Trees in a Circular Lattice (like Figure 18)

`http://www.cs.ucl.ac.uk/staff/W.Langdon/lisp2dot.html` gives some code to display trees as circular lattices [Daida *et al.*, 2005].

A.6 GenProg

Le Goues' bug fixing system (Section 3) is available at `http://genprog.cs.virginia.edu/` [Weimer *et al.*, 2010].

References

[Angeline, 1994] Peter John Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994.

[Arcuri and Yao, 2008] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 162–168, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[Banzhaf and Langdon, 2002] W. Banzhaf and W. B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3(1):81–91, March 2002.

[Barr *et al.*, 2015] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In Tao Xie and Michal Young, editors, *International Symposium on Software Testing and Analysis, ISSTA 2015*, pages 257–269, Baltimore, Maryland, USA, 14-17 July 2015. ACM. ACM SIGSOFT Distinguished Paper Award.

[Brady *et al.*, 2014] Adam Brady, Jason Lawrence, Pieter Peers, and Westley Weimer. genBRDF: discovering new analytic BRDFs with genetic programming. *ACM Transactions on Graphics*, 33(4):114:1–114:11, July 2014.

[Bruce *et al.*, 2015] Bobby R. Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In Sara Silva et al., editors, *GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pages 1327–1334, Madrid, Spain, 11-15 July 2015. ACM, ACM.

[Bruce, 2015] Bobby R. Bruce. Energy optimisation via genetic improvement A SBSE technique for a new era in software development. In William B. Langdon et al., editors, *Genetic Improvement 2015 Workshop*, pages 819–820, Madrid, 11-15 July 2015. ACM.

[Burke *et al.*, 2004] Edmund K. Burke, Steven Gustafson, and Graham Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, February 2004.

[Burles *et al.*, 2015a] Nathan Burles, Edward Bowles, Bobby R. Bruce, and Komsan Srivisut. Specialising Guava's cache to reduce energy consumption. In Yvan Labiche and Marcio Barros, editors, *SSBSE*, volume 9275 of *LNCS*, pages 276–281, Bergamo, Italy, September 5-7 2015. Springer.

[Burles *et al.*, 2015b] Nathan Burles, Jerry Swan, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, and Nadarajen Veerapen. Embedded dynamic improvement. In William B. Langdon et al., editors, *Genetic Improvement 2015 Workshop*, pages 831–832, Madrid, 11-15 July 2015. ACM.

[Cavicchio, Jr, 1970] Daniel Joseph Cavicchio, Jr. *Adaptive search using simulated evolution*. PhD thesis, Dept. of Computer and Communication Sciences, University of Michigan, Ann Arbor, August 1970.

[Daida *et al.*, 2005] Jason M. Daida, Adam M. Hilss, David J. Ward, and Stephen L. Long. Visualizing tree structures in genetic programming. *Genetic Programming and Evolvable Machines*, 6(1):79–110, March 2005.

[Durbin and others, 2010] Richard M. Durbin et al. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 28 Oct 2010.

[Foster and Somayaji, 2010] Blair Foster and Anil Somayaji. Object-level recombination of commodity applications. In Juergen Branke et al., editors, *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 957–964, Portland, Oregon, USA, 7-11 July 2010. ACM.

[Foster, 2001] James A. Foster. Review: Discipulus: A commercial genetic programming system. *Genetic Programming and Evolvable Machines*, 2(2):201–203, June 2001.

[Gathercole and Ross, 1994] Chris Gathercole and Peter Ross. Dynamic training subset selection for supervised learning in genetic programming. In Yuval Davidor et al., editors, *Parallel Problem Solving from Nature III*, volume 866 of *LNCS*, pages 312–321, Jerusalem, 9-14 October 1994. Springer-Verlag.

[Harding and Banzhaf, 2009] Simon L. Harding and Wolfgang Banzhaf. Distributed genetic programming on GPUs using CUDA. In Ignacio Hidalgo et al., editors, *Workshop on Parallel Architectures and Bioinspired Algorithms*, pages 1–10, Raleigh, NC, USA, 13 September 2009. Universidad Complutense de Madrid.

[Harman and Jones, 2001] Mark Harman and Bryan F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, December 2001.

[Harman *et al.*, 2012a] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark. The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs. In *The 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 12)*, pages 1–14, Essen, Germany, September 3-7 2012. ACM.

[Harman *et al.*, 2012b] Mark Harman, Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11:1–11:61, November 2012.

[Harman *et al.*, 2014] Mark Harman, Yue Jia, and William B. Langdon. Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In Claire Le Goues and Shin Yoo, editors, *Proceedings of the 6th International Symposium, on Search-Based Software Engineering, SSBSE 2014*, volume 8636 of *LNCS*, pages 247–252, Fortaleza, Brazil, 26-29 August 2014. Springer. Winner SSBSE 2014 Challange Track.

[Harris, 1997] Christopher Harris. *An investigation into the Application of Genetic Programming techniques to Signal Analysis and Feature Detection*. PhD thesis, University College, London, UK, 26 September 1997.

[Harvey *et al.*, 1998] Brad Harvey, James A. Foster, and Deborah Frincke. Byte code genetic programming. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, pages 59–63, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Stanford University Bookstore.

[Hussain and Malliaris, 2000] Daniar Hussain and Steven Malliaris. Evolutionary techniques applied to hashing: An efficient data retrieval method. In Darrell Whitley et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, page 760, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[Hutchins *et al.*, 1994] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering, ICSE-16*, pages 191–200, May 1994.

[International Human Genome Sequencing Consortium, 2001] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 15 Feb 2001.

[Jia *et al.*, 2015] Yue Jia, Mark Harman, William B. Langdon, and Alexandru Marginean. Grow and serve: Growing Django citation services using SBSE. In Shin Yoo and Leandro Minku, editors, *SSBSE 2015 Challenge Track*, volume 9275 of *LNCS*, pages 269–275, Bergamo, Italy, 5-7 September 2015.

[Klahold *et al.*, 1998] Stefan Klahold, Steffen Frank, Robert E. Keller, and Wolfgang Banzhaf. Exploring the possibilites and restrictions of genetic programming in Java bytecode. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, pages 120–124, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Stanford University Bookstore.

[Klus *et al.*, 2012] Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Cheung, Graham Pullan, Ian McFarlane, Giles S. H. Yeo, and Brian Y. H. Lam. BarraCUDA - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5(27), 2012.

[Koza, 1992] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[Landsborough *et al.*, 2015] Jason Landsborough, Stephen Harding, and Sunny Fugate. Removing the kitchen sink from software. In William B. Langdon et al., editors, *Genetic Improvement 2015 Workshop*, pages 833–838, Madrid, 11-15 July 2015. ACM.

[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*, pages 2376–2383, Barcelona, 18-23 July 2010. IEEE.

[Langdon and Harman, 2014a] W. B. Langdon and M. Harman. Genetically improved CUDA kernels for stereocamera. Research Note RN/14/02, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK, 20 February 2014.

[Langdon and Harman, 2014b] William B. Langdon and Mark Harman. Genetically improved CUDA C++ software. In Miguel Nicolau et al., editors, *17th European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 87–99, Granada, Spain, 23-25 April 2014. Springer.

[Langdon and Harman, 2015a] William B. Langdon and Mark Harman. Grow and graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In William B. Langdon et al., editors, *Genetic Improvement 2015 Workshop*, pages 805–810, Madrid, 11-15 July 2015. ACM.

[Langdon and Harman, 2015b] William B. Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, February 2015.

[Langdon and Harrison, 2008] W. B. Langdon and A. P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*, 12(12):1169–1183, October 2008. Special Issue on Distributed Bioinspired Algorithms.

[Langdon and Lam, 2015] W. B. Langdon and Brian Yee Hong Lam. Genetically improved barraCUDA. Research Note RN/15/03, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK, 28 May 2015.

[Langdon and Petke, 2016] William B. Langdon and Justyna Petke. Genetic improvement. IEEE Software Blog, February 3 2016.

[Langdon and Poli, 1997a] W. B. Langdon and R. Poli. An analysis of the MAX problem in genetic programming. In John R. Koza et al., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 222–230, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Langdon and Poli, 1997b] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry et al., editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 June 1997.

[Langdon and Poli, 2002] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[Langdon *et al.*, 2010] William B. Langdon, Mark Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416–2430, December 2010.

[Langdon *et al.*, 2014] William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3D medical image registration CUDA software with genetic programming. In Christian Igel et al., editors, *GECCO '14: Proceeding of the sixteenth annual conference on genetic and evolutionary computation conference*, pages 951–958, Vancouver, BC, Canada, 12-15 July 2014. ACM.

[Langdon *et al.*, 2015] William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. Improving CUDA DNA analysis software with genetic programming. In Sara Silva et al., editors, *GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pages 1063–1070, Madrid, 11-15 July 2015. ACM.

[Langdon, 1995] W. B. Langdon. Directed crossover within genetic programming. Research Note RN/95/71, University College London, Gower Street, London WC1E 6BT, UK, September 1995.

[Langdon, 1998] William B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 1998.

[Langdon, 2010] W. B. Langdon. A many threaded CUDA interpreter for genetic programming. In Anna Isabel Esparcia-Alcazar et al., editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, pages 146–158, Istanbul, 7-9 April 2010. Springer.

[Langdon, 2012] W.B. Langdon. Creating and debugging performance CUDA C. In Francisco Fernandez de Vega et al., editors, *Parallel Architectures and Bioinspired Algorithms*, volume 415 of *Studies in Computational Intelligence*, chapter 1, pages 7–50. Springer, 2012.

[Langdon, 2015a] W. B. Langdon. Genetic improvement of software for multiple objectives. In Yvan Labiche and Marcio Barros, editors, *SSBSE*, volume 9275 of *LNCS*, pages 12–28, Bergamo, Italy, September 5-7 2015. Springer. Invited keynote.

[Langdon, 2015b] William B. Langdon. Genetically improved software. In Amir H. Gandomi et al., editors, *Handbook of Genetic Programming Applications*, chapter 8, pages 181–220. Springer, 2015.

[Le Goues *et al.*, 2012a] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Martin Glinz, editor, *34th International Conference on Software Engineering (ICSE 2012)*, pages 3–13, Zurich, June 2-9 2012.

[Le Goues *et al.*, 2012b] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and operators for improving evolutionary software repair. In Terry Soule et al., editors, *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 959–966, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[Li and Durbin, 2010] Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[Liu *et al.*, 2006] Weiguo Liu, Bertil Schmidt, Geritt Voss, Andre Schroder, and Wolfgang Muller-Wittig. Bio-sequence database scanning on a GPU. In *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, Rhodes, Greece, 25-29 April 2006. IEEE Press.

[Lopez-Herrejon and Linsbauer, 2015] Roberto E. Lopez-Herrejon and Lukas Linsbauer. Genetic improvement for software product lines: An overview and a roadmap. In William B. Langdon et al., editors, *Genetic Improvement 2015 Workshop*, pages 823–830, Madrid, 11-15 July 2015. ACM.

[Lukschandl *et al.*, 1998] Eduard Lukschandl, Magus Holmlund, and Eirk Moden. Automatic evolution of Java bytecode: First experience with the Java virtual machine. In Riccardo Poli et al., editors, *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, pages 14–16, Paris, France, 14-15 April 1998. CSRP-98-10, The University of Birmingham, UK.

[Luo *et al.*, 2013] Ruibang Luo, Thomas Wong, Jianqiao Zhu, Chi-Man Liu, Xiaoqian Zhu, Edward Wu, Lap-Kei Lee, Haoxiang Lin, Wenjuan Zhu, David W. Cheung, Hing-Fung

Ting, Siu-Ming Yiu, Shaoliang Peng, Chang Yu, Yingrui Li, Ruiqiang Li, and Tak-Wah Lam. SOAP3-dp: Fast, accurate and sensitive GPU-based short read aligner. *PLoS ONE*, 8(5):e65632, 2013.

[Manavski and Valle, 2008] Svetlin Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[McPhee and Hopper, 1999] Nicholas Freitag McPhee and Nicholas J. Hopper. Analysis of genetic diversity through population history. In Wolfgang Banzhaf et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1112–1120, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[Merrill *et al.*, 2012] Duane Merrill, Michael Garland, and Andrew Grimshaw. Policy-based tuning for performance portability and library co-optimization. In *Innovative Parallel Computing (InPar), 2012*. IEEE, May 2012.

[Modat *et al.*, 2010] Marc Modat, Gerard R. Ridgway, Zeike A. Taylor, Manja Lehmann, Josephine Barnes, David J. Hawkes, Nick C. Fox, and Seybastien Ourselin. Fast free-form deformation using graphics processing units. *Computer Methods and Programs in Biomedicine*, 98(3):278–284, 2010.

[Mrazek *et al.*, 2015] Vojtech Mrazek, Zdenek Vasicek, and Lukas Sekanina. Evolutionary approximation of software for embedded systems: Median function. In William B. Langdon et al., editors, *Genetic Improvement 2015 Workshop*, pages 795–801, Madrid, 11-15 July 2015. ACM.

[Orlov and Sipper, 2011] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, April 2011.

[Owens *et al.*, 2008] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. Invited paper.

[Paterson and Livesey, 1997] Norman Paterson and Mike Livesey. Evolving caching algorithms in C by genetic programming. In John R. Koza et al., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 262–267, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Petke *et al.*, 2014a] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Using genetic improvement & code transplants to specialise a C++ program to a problem class. 11th Annual Humies Awards 2014, 14 July 2014. Winner Silver.

[Petke *et al.*, 2014b] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In Miguel Nicolau et al., editors, *17th European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 137–149, Granada, Spain, 23-25 April 2014. Springer.

[Poli *et al.*, 2007] Riccardo Poli, William B. Langdon, and Stephen Dignum. On the limiting distribution of program sizes in tree-based genetic programming. In Marc Ebner et al., editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 193–204, Valencia, Spain, 11-13 April 2007. Springer.

[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).

[Reeder and Giegerich, 2004] Jens Reeder and Robert Giegerich. Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics. *BMC Bioinformatics*, 5(1):104, 2004.

[Reeder *et al.*, 2007] Jens Reeder, Peter Steffen, and Robert Giegerich. pknotsRG: RNA pseudoknot folding including near-optimal structures and sliding windows. *Nucleic Acids Research*, 35(suppl 2):W320–W324, 2007.

[Reguly and Giles, 2012] Istvan Reguly and Mike Giles. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *Innovative Parallel Computing (InPar), 2012*, pages 230–241, San Jose, USA, 13-14 May 2012. IEEE.

[Risco-Martin *et al.*, 2010] Jose L. Risco-Martin, David Atienza, J. Manuel Colmenar, and Oscar Garnica. A parallel evolutionary algorithm to optimize dynamic memory managers

in embedded systems. *Parallel Computing*, 36(10-11):572–590, 2010. Parallel Architectures and Bioinspired Algorithms.

[Rodriguez-Mier *et al.*, 2010] Pablo Rodriguez-Mier, Manuel Mucientes, Manuel Lama, and Miguel I. Couto. Composition of web services through genetic programming. *Evolutionary Intelligence*, 3(3-4):171–186, 2010.

[Schuler and Zeller, 2009] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for java. In Hans van Vliet and Valérie Issarny, editors, *ESEC/SIGSOFT FSE*, pages 297–298, Amsterdam, Netherlands, 24-28 August 2009. ACM.

[Schulte *et al.*, 2013] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS 2013, pages 317–328, Houston, Texas, USA, March 16-20 2013. ACM.

[Schulte *et al.*, 2014a] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'14*, pages 639–652, Salt Lake City, Utah, USA, 1-5 March 2014. ACM.

[Schulte *et al.*, 2014b] Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, 15(3):281–312, September 2014.

[Schulte *et al.*, 2015] Eric Schulte, Westley Weimer, and Stephanie Forrest. Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair. In William B. Langdon et al., editors, *Genetic Improvement 2015 Workshop*, pages 847–854, Madrid, 11-15 July 2015. ACM. Best Paper.

[Sitthi-amorn *et al.*, 2011] Pitchaya Sitthi-amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(6):article:152, December 2011. Proceedings of ACM SIGGRAPH Asia 2011.

[Soule and Heckendorn, 2002] Terence Soule and Robert B. Heckendorn. An analysis of the causes of code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 3(3):283–309, September 2002.

[Stam, 2008] Joe Stam. Stereo imaging with CUDA. Technical report, nVidia, V 0.2 3 Jan 2008. StereoImaging.pdf distributed with `StereoCamera_v1_1c.zip`.

[Steffen and Giegerich, 2006] Peter Steffen and Robert Giegerich. Table design in dynamic programming. *Information and Computation*, 204(9):1325–1345, 2006.

[Syswerda, 1989] Gilbert Syswerda. Uniform crossover in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the third international conference on Genetic Algorithms*, pages 2–9, George Mason University, 4-7 June 1989. Morgan Kaufmann.

[Tackett, 1994] Walter Alden Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, USA, 1994.

[Teller and Andre, 1997] Astro Teller and David Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In John R. Koza et al., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Untch *et al.*, 1993] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. In *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, pages 139–148, Cambridge, Massachusetts, 1993.

[Weimer *et al.*, 2009] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In Stephen Fickas, editor, *International Conference on Software Engineering (ICSE) 2009*, pages 364–374, Vancouver, May 16-24 2009.

[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.

[White *et al.*, 2008] David R. White, John Clark, Jeremy Jacob, and Simon M. Poulding. Searching for resource-efficient programs: low-power pseudorandom number generators.

In Maarten Keijzer et al., editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1775–1782, Atlanta, GA, USA, 12-16 July 2008. ACM.

[White *et al.*, 2011] David R. White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, August 2011.

[Wu *et al.*, 2015] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In Sara Silva et al., editors, *GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pages 1375–1382, Madrid, 11-15 July 2015. ACM.

[Yao *et al.*, 2014] Xiangjuan Yao, Mark Harman, and Yue Jia. 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, pages 919–930, Hyderabad, India, 2014. ACM.

[Yeboah-Antwi and Baudry, 2015] Kwaku Yeboah-Antwi and Benoit Baudry. Embedding adaptivity in software systems using the ECSELR framework. In William B. Langdon et al., editors, *Genetic Improvement 2015 Workshop*, pages 839–844, Madrid, 11-15 July 2015. ACM.