Chapter 8
Genetically Improved Software

William B. Langdon, Computer Science, University College, London

Abstract Genetic programming (GP) can dramatically increase computer programs' performance. It can automatically port or refactor legacy code written by domain experts and specialist software engineers. After reviewing SBSE research on evolving software we describe an open source parallel StereoCamera image processing application in which GI optimisation gave a seven fold speedup on nVidia Tesla GPU hardware not even imagined when the original state-of-the-art CUDA GPGPU C++ code was written.
Sources and data sets are available on line.

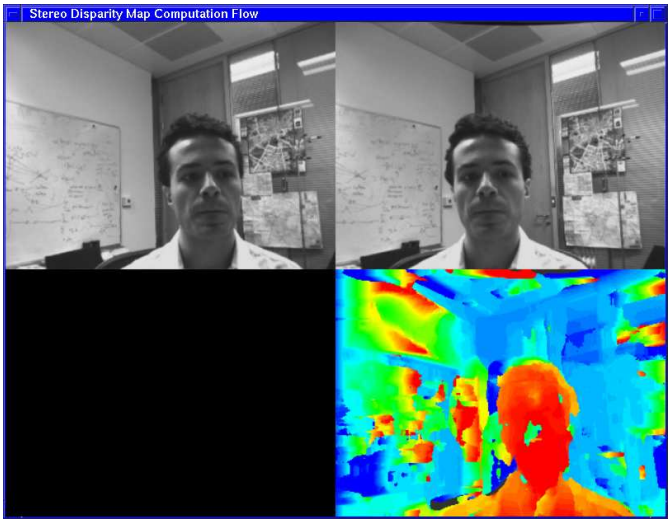

Fig. 8.1 Top: left and right stereo images. Bottom: Discrepancy between images, which can be used to infer distances.

Preprint Handbook of Genetic Programming Applications, Amir H. Gandomi, Amir H. Alavi and Conor Ryan *Eds.*, pp 181-220 Springer, 2015. DOI: 10.1007/978-3-319-20883-1_8

8.1 Introduction

As other chapters in the book show, genetic programing [Koza, 1992; Banzhaf *et al.*, 1998; Poli *et al.*, 2008] has been very widely applied[1]. For example in modelling [Kordon, 2010], prediction [Langdon and Barrett, 2004; Podgornik *et al.*, 2011; Kovacic and Sarler, 2014], classification [Freitas, 1997], design [Lohn and Hornby, 2006] (including algorithm design [Haraldsson and Woodward, 2014]), and creating art [Reynolds, 2011; Jacob, 2001; Langdon, 2004; Romero *et al.*, 2013]. Here we concentrate upon application of genetic programming to software itself [Arcuri and Yao, 2014]. We start by briefly summarising research which evolved complete software but mostly we will concentration on newer work which has very effectively side stepped, what John Koza referred to as the *S-word* in artificial intelligence, the *scaling* problem, by using genetic programming not to create complete software but rather to enhance existing (human written) software.

The next section describes early successes with using GP to evolve real, albeit small, code and for automatically fixing bugs and then Sections 8.3–8.6 describe recent success in which GP improved substantial (human written) C or C++ programs. The last part of the chapter (Section 8.7 onwards) describes in detail one of these. It shows how genetic programming was used to automatically evolve an almost seven fold speedup in parallel graphics code for extracting depth from stereoscopic image pairs. (See Figure 8.1.)

8.2 Background

8.2.1 Hashes, Caches and Garbage Collection

Three early examples of real software being evolved using genetic programming are: hashing, caching and garbage collection. Each has the advantages of being small, potentially of high value and difficult to do either by hand or by theoretically universal principles. In fact there is no universally correct optimal answer. Any implementation which is good in one circumstance may be bettered in another use case by software deliberately designed for that use case. Thus there are several examples where not only can GP generate code but for particular circumstances, it has exceeded the state-of-the art human written code. Whilst this is not to say a human could not do better. Indeed they may take inspiration, or even code, from the evolved solution. It is that to do so, requires a programmer skilled in the art, for each new circumstance. Whereas, at least in principle, the GP can be re-run for each new use case and so automatically generate an implementation specific to that user.

[1] Genetic programming bibliography http://www.cs.bham.ac.uk/~wbl/biblio/ gives details of more than nine thousand articles, papers, books, etc.

Starting with [Hussain and Malliaris, 2000] several teams have evolved good hashing algorithms ([Berarducci *et al.*, 2004], [Estebanez *et al.*,] and [Karasek *et al.*, 2011]).

Paterson showed GP can create problem specific caching code [Paterson and Livesey, 1997]. [O'Neill and Ryan, 1999] used their Grammatical Evolution [O'Neill and Ryan, 2001; O'Neill and Ryan, 2003] approach also to create code. Whilst [Branke *et al.*, 2006] looked at a slightly different problem: deciding which (variable length) documents to retain to avoid fetching them again across the Internet. (Following [Handley, 1994] several authors have sped up genetic programming itself by caching partial fitness evaluations, including me [Langdon, 1998]. However here we are interested in improving software in general rather than just improving genetic programming.)

Many languages allow the programmer to allocate and free chunks of memory as their program runs, e.g. C, C++ and Java. Typically the language provides a dynamic memory manager, which frees the programmer of the tedium of deciding exactly which memory is used and provides some form of garbage collection whereby memory that is no longer in use can be freed for re-use. Even with modern huge memories, memory management can impose a significant overhead. [Risco-Martin *et al.*, 2010] showed the GP can generate an optimised garbage collector for the C language.

8.2.2 Mashups, Hyper-heuristics and Multiplicity Computing

The idea behind web services is that useful services should be easily constructed from services across the Internet. Such hacked together systems are known as web mashups. A classic example is a travel service which invokes web servers from a number of airlines and hotel booking and car hire services, and is thus able to provide a composite package without enormous coding effort in itself. Since web services must operate within a defined framework ideally with rigid interfaces, they would seem to be ideal building blocks with which genetic programming might construct high level programs. Starting with Rodriguez-Mier, several authors have reported progress with genetic programming evolving composite web services [Rodriguez-Mier *et al.*, 2010; Fredericks and Cheng, 2013; Xiao *et al.*, 2012].

There are many difficult optimisation problems which in practise are efficiently solved using heuristic search techniques, such as genetic algorithms [Holland, 1992; Goldberg, 1989]. However typically the GA needs to be tweaked to get the best for each problem. This has lead to the generation of hyper-heuristics [Burke *et al.*, 2013], in which the GA or other basic solver is tweaked automatically. Typically genetic programming is used. Indeed some solvers have been evolved by GP combining a number of basic techniques as well as tuning parameters or even re-coding GA components, such as mutation operators [Pappa *et al.*, 2014].

A nice software engineering example of heuristics is compiler code generation. Typically compilers are expected not only to create correct machine code but also that it should be in some sense be "good". Typically this means the code should be fast or small. [Mahajan and Ali, 2008] used GP to give better code generation heuristics in Harvard's MachineSUIF compiler.

Multiplicity computing [Cadar *et al.*, 2010] seeks to over turn the current software mono-culture where one particular operating system, web browser, software company, etc., achieves total dominance of the software market. Not only are such monopolies dangerous from a commercial point of view but they have allowed widespread problems of malicious software (especially computer viruses) to prosper. Excluding specialist areas, such as mutation testing [DeMillo and Offutt, 1991; Langdon *et al.*, 2010], so far there has been only a little work in the evolution of massive numbers of software variants [Feldt, 1998]. Only software automation (perhaps by using genetic programming) appears a credible approach to N-version programming (with N much more than 3). N-version programming has also been proposed as a way of improving predictive performance by voting between three or more classifiers [Imamura and Foster, 2001; Imamura *et al.*, 2003] or using other non-linear combinations to yield a higher performing multi-classifier [Langdon and Buxton, 2001; Buxton *et al.*, 2001].

Other applications of GP include: creating optimisation benchmarks which demonstrate the relative strengths and weaknesses of optimisers [Langdon and Poli, 2005] and first steps towards the use of GP on mobile telephones [Cotillon *et al.*, 2012].

8.2.3 Genetic Programming and Non-Function Requirements

Andrea Arcuri was in at the start of inspirational work on GP showing it can create real code from scratch. Although the programs remain small, David White, he and John Clark [White *et al.*, 2011] also evolved programs to accomplish real tasks such as creating pseudo random numbers for ultra tiny computers where they showed a trade off between "randomness" and energy consumption.

The Virginia University group (see next section) also showed GP evolving Pareto optimal trade offs between speed and fidelity for a graphics hardware display program [Sitthi-amorn *et al.*, 2011]. Evolution seems to be particularly suitable for exploring such trade-offs [Feldt, 1999; Harman *et al.*, 2012] but (except for the work described later in this chapter) there has been little research in this area.

[Orlov and Sipper, 2011] describe a very nice system, Finch, for evolving Java byte code. The initial program to be improved is typically a Java program, which is compiled into byte code. Effectively the GP population instead of starting randomly [Lukschandl *et al.*, 1998] is seeded [Langdon and Nordin, 2000] with byte code from the initial program. The Finch crossover operator acts on Java byte code to ensure the offspring program area also valid java byte code. Large benefits arise because there is no need to compile the new programs. Instead the byte code can be run immediately. As Java is a main stream language, the byte code can be efficiently

executed using standard tools, such as Java virtual machines and just in time (JIT) compilers. Also after evolution, standard java tools can be used to attempt to reverse the evolved byte code into Java source code.

[Archanjo and Von Zuben, 2012] present a GP system for evolving small business systems. They present an example of a database system for supporting a library of books.

[Ryan, 1999] and [Katz and Peled, 2013] provide interesting alternative visions. In genetic improvement the performance, particularly the quality of the mutated program's output, is assessed by running the program. Instead they suggest each mutation be provably correct and thus the new program is functionally the same as the original but in some way it is improved, e.g. by running in parallel. [Katz and Peled, 2013] suggests combining GP with model checking to ensure correctness.

[Zhu and Kulkarni, 2013] suggest using GP to evolve fault tolerant programs. [Schulte *et al.*, 2014a] describes a nice system which can further optimise the low level Intel X86 code generated by optimising compilers. They show evolution can reduce energy consumption of non-trivial programs. (Their largest application contains 141 012 lines of code.)

8.2.4 Automatic Bug Fixing

As described in the previous two sections, recently genetic programming has been applied to the production of programs itself, however so far relatively small programs have been evolved. Nonetheless GP has had some great successes when applied to existing programs. Perhaps the best known work is that on automatic bug fixing [Arcuri and Yao, 2008]. Particularly the Humie award winning[2] work of Westley Weimer (Virginia University) and Stephanie Forrest (New Mexico) [Forrest *et al.*, 2009]. This has received multiple awards and best paper prizes [Weimer *et al.*, 2009; Weimer *et al.*, 2010]. GP has been used repeatedly to automatically fix most (but not all) real bugs in real programs [Le Goues *et al.*, 2012a]. Weimer and Le Goues have now shown GP bug fixing to be effective on several millions of lines of C++ programs. Once GP had been used to *do the impossible* others tried [Wilkerson and Tauritz, 2010; Bradbury and Jalbert, 2010; Ackling *et al.*, 2011] and it was improved [Kessentini *et al.*, 2011] and also people felt brave enough to try other techniques, e.g. [Nguyen *et al.*, 2013; Kim *et al.*, 2013]. Indeed their colleague, Eric Schulte, has shown GP can even work at abstraction levels other than source code. In [Schulte *et al.*, 2010] he showed bugs can be fixed at the level of the assembler code generated by the compiler or even machine code [Schulte *et al.*, 2013]. After Weimer and co-workers showed that automatic bugfixing was not impossible, people studied the problem more openly. It turns out, for certain real bugs, with modern software engineering support tools, such as bug localisation (e.g. [Yoo, 2012]), the problem may not even be hard [Weimer, 2013].

[2] Human-competitive results presented at the annual GECCO conference http://www.genetic-programming.org/combined.php

Formal theoretical analysis [Cody-Kenny and Barrett, 2013] of evolving sizable software is still thin on the ground. Much of the work presented here is based on GP re-arranging lines of human written code. In a very large study of open source software [Gabel and Su, 2010] showed that excluding white space, comments and details of variable names, any human written line of code has probably been written before. In other words, given a sufficiently large feedstock of human written code, current programs could have been written by re-using and re-ordering existing lines of code. In many cases in this and the following sections, this is exactly what GP is doing. [Schulte *et al.*, 2014b] provides a solid empirical study which refutes the common assumption that software is fragile. (See also Figure 8.2). While a single random change may totally break a program, mutation and crossover operations can be devised which yield populations of offspring programs in which some may be very bad but the population can also contains many reasonable programs and even a few slightly improved ones. Over time the Darwinian processes of fitness selection and inheritance [Darwin, 1859] can amplify the good parts of the population, yielding greatly improved programs.

8.3 Auto Porting Functionality

The Unix compression utility gzip was written in C in the days of Digital Equipment Corp.'s mini-computers. It is largely unchanged. However there is one procedure (of about two pages of code) in it, which is so computationally intensive that it has been re-written in assembler for the Intel 86X architecture (i.e. Linux). The original C version is retained and is distributed as part of Software-artifact Infrastructure Repository sir.unl.edu [Hutchins *et al.*, 1994]. SIR also contains a test suite for gzip. In *Genetic Improvement*, as with Le Goues' bug-fixing work, we start with an existing program and a small number of test cases. In the case of the gzip function, we showed genetic programming could evolve a parallel implementation for an architecture not even dreamt of when the original program was written [Langdon and Harman, 2010]. Whereas Le Goues uses the original program's AST (abstract syntax tree) to ensure that many of the mutated programs produced by GP compile, we have used a BNF grammar. In the case of [Langdon and Harman, 2010] the grammar was derived from generic code written by the manufacture of the parallel hardware. Note that it had nothing special to do with gzip. The original function in gzip was instrumented to record its inputs and its outputs each time it was called (see Figure 8.3). When gzip was run on the SIR test suite, this generated more than a million test cases, however only a few thousand were used by the GP[3]. Essentially GP was told to create parallel code from the BNF grammar which when given a small number of example inputs returned the same answers. The resulting parallel code is functionally the same as the old gzip code.

[3] Later work used even fewer tests.

8.4 Bowtie2^{GP} Improving 50 000 lines of C++

As Figure 8.4 shows, genetic programming produces populations of programs which may have different abilities on different scales. While Figure 8.4 shows speed versus quality, other tradeoffs have been investigated ([Harman *et al.*, 2012], see also [Schulte *et al.*, 2014a]). For example it may be impossible to simultaneously minimise execution time, memory foot print and energy consumption. Yet, conventionally human written programs choose one trade-off between multiple objectives and it becomes infeasible to operate the program with another trade-off. For example, consider approximate string matching.

Finding the best match between (noisy) strings is the life blood of Bioinformatics. Huge amounts of people's time and computing resources are devoted every day to matching protein amino acid sequences against databases of known proteins from

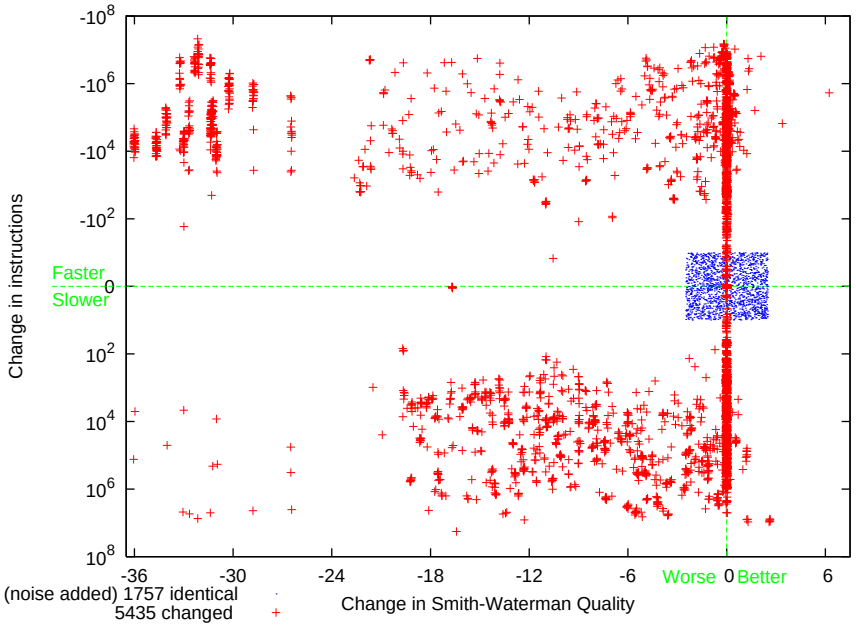

Fig. 8.2 C++ is not fragile. Performance versus speed for random mutations of Bowtie2. The horizontal axis shows the change in quality of Bowtie2 output, whilst the vertical axis (note non-linear scale) shows the change in the number of lines of code executed. As expected some mutations totally destroy the program, e.g. they fail to compile or abort (not plotted) or reduce the quality of the answer enormously (e.g. -36). Some are slower (lower half) and some are faster (top). However a large number have exactly the same quality as the original code (plotted above "0"). These may be either slower or faster. The rectangle of dots attempts to emphasise the 18% that are identical (in terms of quality of answer and run time) to the original code. To the right of the "0", there are even a few random programs which produce slightly better answers than the original code. It is these Darwinian evolution selects and breeds the next generation from. Total 10000 random program runs. Failed runs are not plotted.

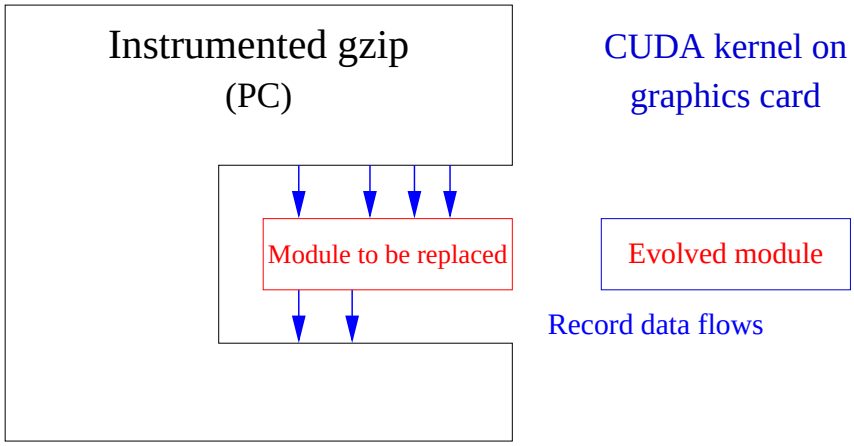

Fig. 8.3 Auto porting a program module to new hardware (a GPU). The original code is instrumented to record the inputs (upper blue arrows) to the target function (red) and the result (lower blue arrows) it calculates. Its inputs and outputs are logged every time every time it is called. These become the test suite and fitness function for the automatically evolved replacement module running on novel hardware. By inspecting the evolved CUDA code automatically generated by GP we can see that it is functionally identical to the C code inside gzip. Also it has been demonstrated by running back-to-back with the original code more than a million times [Langdon and Harman, 2010].

all forms of life. The acknowledge gold standard is the BLAST program [Altschul *et al.*, 1997] which incorporate heuristics of known evolutionary rates of change. It is available via the web and can lookup a protein in every species which has been sequences in a few minutes. Even before the sequencing of the human genome, the volume of DNA sequences was exploding exponentially at a rate like Moore's Law [Moore, 1965]. With modern NextGen sequencing machines throwing out 100s of millions (even billions) of (albeit very noisy) DNA base-pair sequences, there is no way that BLAST can be used to process this volume of data. This has lead to human written look up tools for matching NextGen sequences against the human genome. Wikipedia list more than 140 programs (written by some of the brightest people on the planet) which do some form of Bioinformatics string matching.

The authors of all this software are in a quandary. For their code to be useful the authors have to chose a point in the space of tradeoffs between speed, machine resources, quality of solution and functionality, which will: 1) be important to the Bioinformatics community and 2) not be immediately dominated by other programs. In practise they have to choose a target point when they start, as once basic design choices (e.g. target data sources and computer resources) have been made, few people or even research teams have the resources to discard what they have written and start totally from scratch. Potentially genetic programming offers them a way of exploring this space of tradeoffs [Feldt, 1999; Harman *et al.*, 2012]. GP can produce many programs across the trade-off space and so can potentially say "look here is a trade-off which you had not considered".

This could be very useful to the human, even if they refuse to accept machine generated code and insist on coding the solution themselves.

We have made a start by showing GP can transform human written DNA sequence matching code, moving it from one tradeoff point to another. In our example, the new program is specialised to a particular data source and sequence problem for which it is on average more than 70 times faster. Indeed on this particular problem, we were fortunate that not only is the variant faster but indeed it gives a slight quality improvement on average [Langdon and Harman, 2015].

8.5 Merging Boolean Satisfiability Code Written by Experts

The basic GI technique has also been used to create an improved version of C++ code from multiple versions of a program written by different authors. Boolean Satisfiability is a problem which appears often. MiniSAT is a popular SAT solver. The satisfiability community has advanced rapidly since the turn of the century. This has been due in part to a series of competitions. These include the "MiniSAT hack

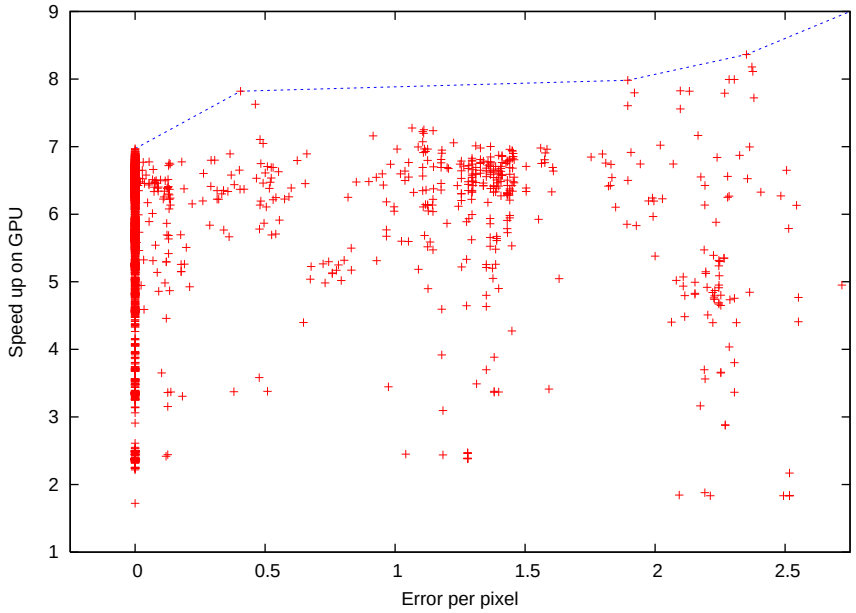

Fig. 8.4 Example of automatically generated Pareto tradeoff front [Harman *et al.*, 2012]. Genetic programming used to improve 2D Stereo Camera code [Stam, 2008] for modern nVidia GPU [Langdon and Harman, 2014b]. Left (above 0) many programs are faster than the original code written by nVidia's image processing expert (human) and give exactly the same answers. Many other automatically generated programs are also faster but give different answers. Some (cf. dotted blue line) are faster than the best zero error program.

track", which is specifically designed to encourage humans to make small changes to the MiniSAT code. The new code is available after each competition. MiniSAT and a number of human variants were given to GI and it was asked to evolve a new variant specifically designed to work better on a software engineering problem (interaction testing) [Petke *et al.*, 2014b]. At GECCO 2014 it received a Human Competitive award (HUMIE) [Petke *et al.*, 2014a].

8.6 Babel Pidgin: Creating and Incorporating New Functionality

Another prize winning genetic programing based technique has been shown to be able to extend the functionality of existing code [Harman *et al.*, 2014]. GP, including human hints, was able to evolved new functionality externally and then search based techniques [Harman, 2011] were used to graft the new code into an existing program (pidgin) of more than 200 000 lines of C++.

8.7 Improving Parallel Processing Code Written by Experts

There is increasing use of parallelism both in conventional computing but also in mobile applications. At present the epitome of parallelism are dedicated multi-core machines based on gaming graphics cards (GPUs). Although originally devised for the consumer market, they are increasingly being used for general purpose computing on GPUs (GPGPU) [Owens *et al.*, 2008] with several the world's fastest computers being based on GPUs. However, although support tools are improving, programming parallel computers continues to be a challenge [Langdon, 2012] and simply leaving code generation to parallel compilers is often insufficient. Instead experts, e.g. [Merrill *et al.*, 2012], have advocated writing highly parametrised parallel code which can then be automatically tuned. Unfortunately this throws the load back on to the coder [Langdon, 2011]. In the rest of the chapter we explain how genetic programming (see Figure 8.5) was able to automatically update for today's GPUs software written specifically by nVidia's image processing expert to show off the early generations of their graphics cards [Stam, 2008]. While originally [Langdon and Harman, 2014b] we considered six types of hardware, in the interests of brevity we shall concentrate on the most powerful (Tesla K20c). Performance of the other five GPUs and more details can be found in [Langdon and Harman, 2014b] and technical report [Langdon and Harman, 2014a]. GP gave more than a six fold performance increase relative to the original code on the same hardware. (Each Tesla K20c contains 2496 processing elements, arranged in 13 blocks of 192 and running at 0.71GHz. Bandwidth to its on board memory is 140Gbytes per second. See Figure 8.6.)

In another example a combination of manual and automated changes to production 3D medical image processing code lead to the creation of a version of a perfor-

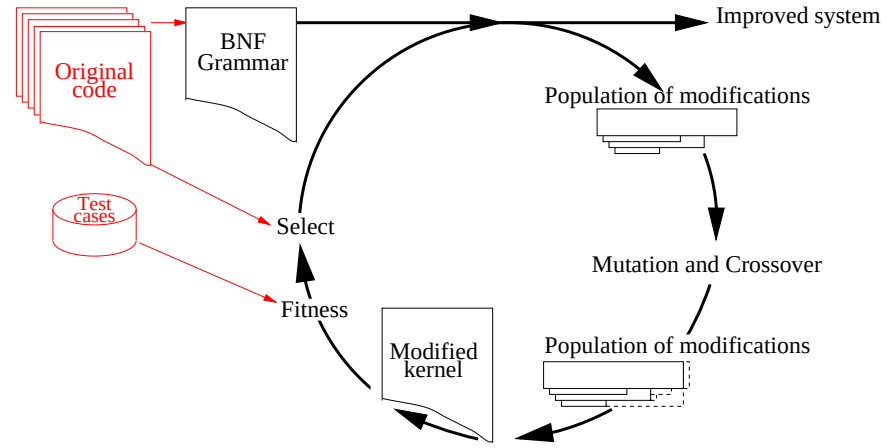

Fig. 8.5 Genetic Improvement of stereoKernel

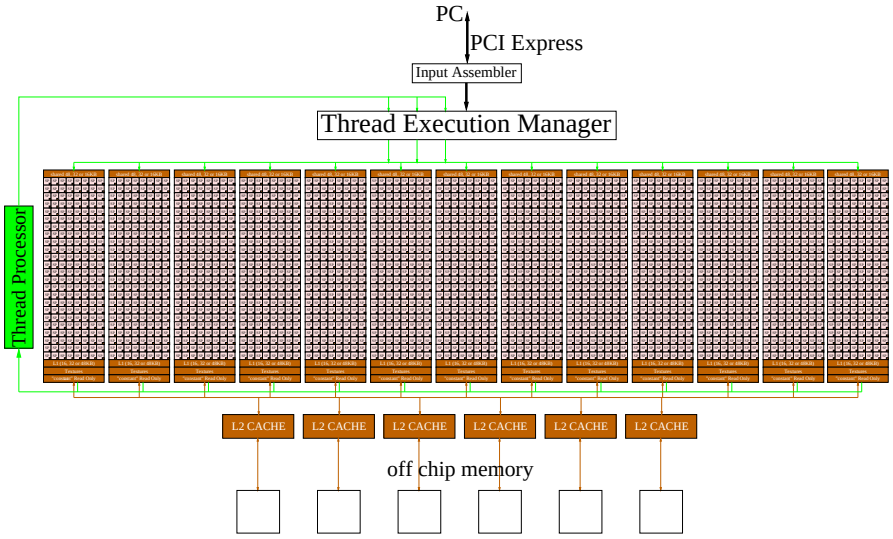

Fig. 8.6 Tesla K20c contains 13 SMX multiprocessors (each containing 192 stream processors), a PCI interface to the host PC, thread handling logic and 4800 MBytes of on board memory.

mance critical kernel which (on a Tesla K20c) is more than 2000 times faster than the production code running on an 2.67GHz CPU [Langdon *et al.*, 2014].

The next sections briefly gives the StereoCamera CUDA code. This is followed by descriptions of the stereo images (page 193), and the code tuning process (pages 195–204). The changes made specifically for the K20c Tesla are described in Section 8.16 (page 205) whilst the Appendix (pages 218–221) holds the com-

plete CUDA source code for the K20c Tesla. The code is also available in `Stereo Camera_v1_1c.zip`.

8.8 Source Code: StereoCamera

The StereoCamera system was written by nVidia's stereo image processing expert Joe Stam [Stam, 2008] to demonstrate their 2007 hardware and CUDA. StereoCamera was the first to show GPUs could give real time stereo image processing ($>$ 30 frames per second). StereoCamera V1.0b was downloaded from SourceForge but, despite the exponential increase in GPU performance, it had not been updated since 2008 (except for my bugfix). In the six years after it was written, nVidia GPUs went through three major hardware architectures whilst their CUDA software went through five major releases.

StereoCamera contains three GPU kernels plus associated host code. We shall concentrate upon one, stereoKernel which contains the main stereo image algorithm. For each pixel in the left image, GPU code stereoKernel reports the number of pixels the right image has to be shifted to get maximal local alignment (see Figure 8.7). [Stam, 2008] notes that the parallel processing power of the GPU allows the local discrepancy between the left and right images to be calculated using the sum of *squares* of the difference (SSD) between corresponding pixels and this sum is taken over the relatively large 11×11 area. It does this by minimising the sum of squares of the difference (SSD) between the left and right images in a 11×11 area around each pixel. Once SSD has been calculated, the grid in the right hand image is displaced one pixel to the left and the calculation is repeated. Although the code is written to allow arbitrary displacements, in practice the right hand grid is move a pixel at a time. SSD is calculated for 0 to 50 displacements and the one with the smallest SSD is reported for each pixel in the left hand image. In principle each pixel's value can be calculated independently but each is surrounded by a "halo" of five others in each direction.

Even on a parallel computer, considerable savings can be made by reducing the total number of calculations by sharing intermediate calculations [Stam, 2008, Fig. 3]. Each SSD calculation (for a given discrepancy between left and right images) involves summing 11 columns (each of 11 squared discrepancy values). By saving the column sums in shared memory adjacent computational threads can calculate just their own column and then read the remaining ten column values calculated by their neighbouring threads.

After one row of pixel SSDs have been calculated, when calculating the SSD of the pixels immediately above, ten of the eleven rows of SSD values are identical. Given sufficient storage, the row values could be saved and then 10 of them could be reused requiring only one row of new square differences to be calculated. However fast storage was scare on GPUs and instead Stam compromised by saving the total SSD (rather than the per row totals). The SSD for the pixel above is then the total SSD plus the contribution for the new row *minus* the contribution from the lowest

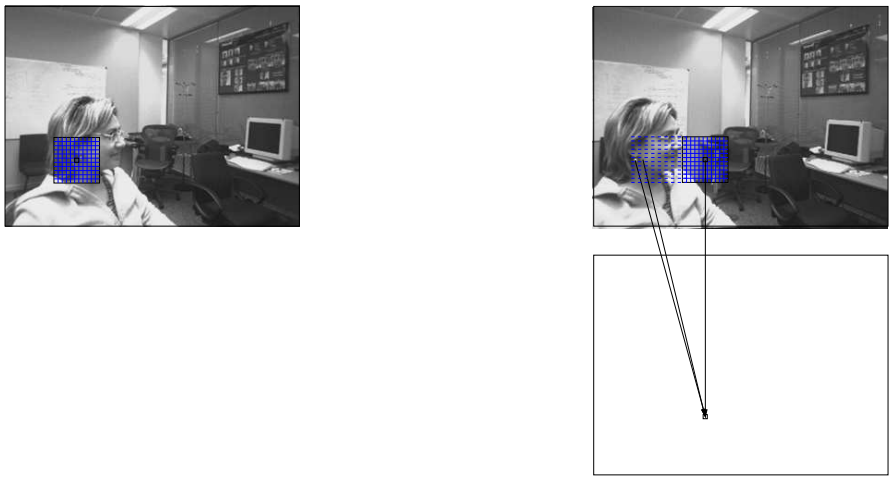

Fig. 8.7 Schematic of stereo disparity calculation. Top: left and right stereo images. Bottom: output. Not to scale. For each pixel stereoKernel calculates the sum of squared differences (SSD) between 11×11 regions centred on the pixel in the left image and the same pixel in the right hand image. This 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. Although the output pixel (bottom) may be updated many times, its final value is the distance moved by the 11×11 region which gives the smallest SSD. I.e. the distance between left and right images which gives the maximum similarity between them (across an 11×11 region). This all has to be done for every pixel. Real time performance is obtained by parallel processing and reducing repeated calculations.

row (which is no longer included in the 11×11 area). Stam took care that the code avoids rounding errors. The more rows which share their partial results, the more efficient is the calculation but then there is less scope for performing calculations in parallel. To avoid re-reading data it is desirable that all the image data for both left and right images (including halos and discrepancy offsets) should fit within the GPU's texture caches. The macro `ROWSperTHREAD` (40) determines how many rows are calculated together in series. The macro `BLOCK_W` (64) determines how the image is partitioned horizontally (see Figures 8.8 and 8.9). To fit the GPU architecture `BLOCK_W` will often be a multiple of 32. In practise all these factors interact in non-obvious (and sometimes undocumented) hardware dependent ways.

8.9 Example Stereo Pairs from Microsoft's I2I Database

Microsoft have made available for image processing research thousands of images. Microsoft's I2I database contains 3010 stereo images. Figure 8.7 (top) is a typical example. Many of these are in the form of movies taken in an office environment. Figure 8.1 shows the first pair from a typical example.

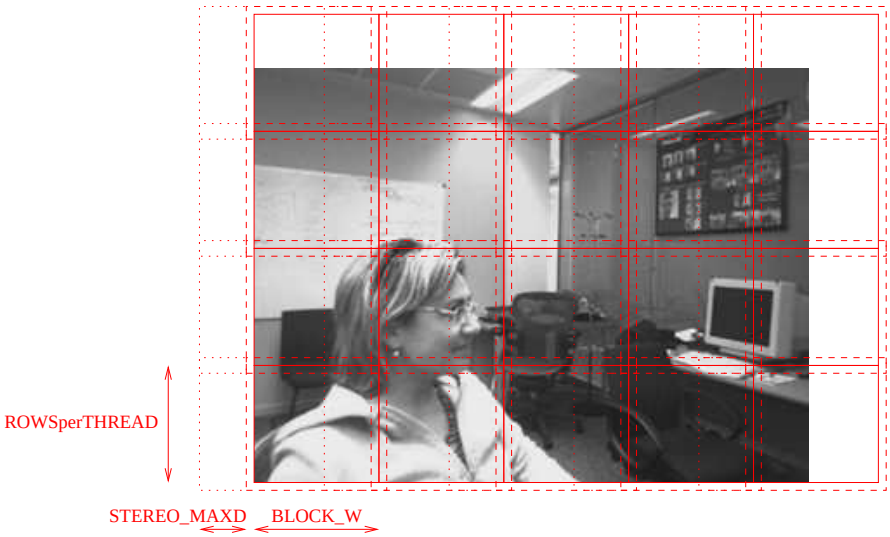

Fig. 8.8 The left and right images (solid rectangle) are split into `BLOCK_W`×`ROWSperTHREAD` tiles. The dashed lines indicate the extra pixels outside the tile which must be read to calculate values for pixels in the tile. The right hand image is progressively offset by between zero and `STEREO_MAXD` pixels (50, dotted lines).

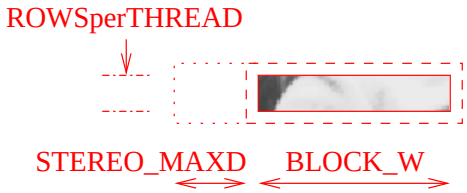

Fig. 8.9 An example of the part of the right hand of a stereo image pair which is processed by a block of CUDA threads. The area covered in the right image is eventually shifted `STEREO_MAXD` (50) pixels to the left. For most GPUs the original code did not use the optimal shape, see Figure 8.10. Although the width (`BLOCK_W`, 64) was correct, the height (`ROWSperTHREAD`) should be reduced from 40 to 5.

We downloaded i2idatabase.zip[4] (1.3GB) and extracted all the stereo image pairs and converted them to grey scale. Almost images all are 320×240 pixels. We took (up to) the first 200 pairs for training leaving 2810 for validation. Notice we are asking the GP to create a new version of the CUDA stereoKernel GPU code which is tuned to pairs of images of this type. As we shall see (in Section 8.15) the improved GPU code is indeed tuned to 320×240 images but still works well on the other I2I stereo pairs.

[4] http://research.microsoft.com/en-us/um/people/antcrim/data_i2i/i2idatabase.zip

8.10 Host Code and Baseline Kernel Code

The supplied C++ code is designed to read stereo images from either stereo webcams or pairs of files and using OpenGL, to display both the pair of input images and the calculated discrepancy between them on the user's monitor (see Figure 8.1). This was adapted to both compare answers generated by the original code with those given by the tuned GP modified code and to time execution of the modified GPU kernel code. These data are logged to a file and the image display is disabled.

The original kernel code is in a separately compiled file to ensure it is not affected by GP specified compiler options (particularly -Xptxas -dlcm, Table 8.1). For each pixel it generates a value in the range $0.0, 1.0, 2.0 \ldots 50.0$ being the minimum discrepancy between the left and right images. If a match between the left and right images cannot be found (i.e. SSD ≥ 500000) then it returns -1.0.

8.11 Pre- and Post- Evolution Tuning and Post Evolution Minimisation of Code Changes

In initial genetic programming runs, it became apparent that there are two parameters which have a large impact on run time but whose default settings are not suitable for the GPUs now available. Since there are few such parameters and they each have a small number of sensible values, it is feasible to run StereoCamera on all reasonable combinations and simply choose the best for each GPU. Hence the revised strategy is to tune `ROWSperTHREAD` and `BLOCK_W` before running the GP. (`DPER`, Section 8.12.2, is not initially enabled.) Figure 8.10 shows the effect of tuning `ROWSperTHREAD` and `BLOCK_W` for the GTX 295. As with [Le Goues *et al.*, 2012b] and our GISMOE approach [Langdon and Harman, 2015], after GP has run the best GP individual from the last generation is cleaned up by a simple one-at-a-time hill climbing algorithm. [Langdon and Harman, 2015] (Section 8.11) and finally `ROWSperTHREAD`, `BLOCK_W` and `DPER` are tuned again. (Often no further changes were needed.)

For each combination of parameters, the kernel is compiled and run. By recompiling rather than using run time argument passing, the nVidia nvcc C++ compiler is given the best chance of optimising the code (e.g. loop unrolling) for these parameters and the particular GPU.

`BLOCK_W` values were based on sizes of thread blocks used by nVidia in the examples supplied with CUDA 5.0. (They were 8, 32, 64, 128, 192, 256, 384 and 512.) All small `ROWSperTHREAD` values or values which divide into the image height (240) exactly were tested. (I.e., 1, … 18, 20, 21, 24, 26, 30, 34, 40, 48, 60, 80, 120 and 240.) Autotuning reduced `ROWSperTHREAD` (see Figure 8.9) from 40 to 5 before the GP was run. For the Tesla K20c, this gave a speed up of 2.373 ± 0.03 fold.

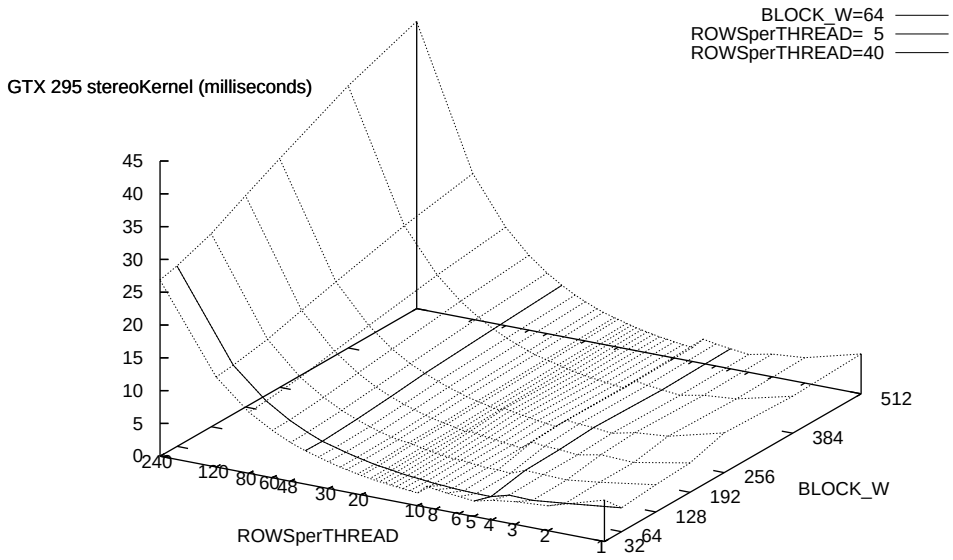

Fig. 8.10 The effect of changing the work done per thread (`ROWSperTHREAD`) and the block size (`BLOCK_W`) on CUDA kernel speed before it was optimised by GP. For all but one of the GPUs, stereoKernel is fastest at 5,64 (the default is 40,64).

The best GP individual in the last generation is minimised by starting at its beginning and progressively removing each individual mutation and comparing the performance of the new kernel with the evolved one. For simplicity this is done on the last training stereo image pair. Unless the new kernel is worse the mutation is excluded permanently. To encourage removal of mutations with little impact, those that make less than 1% difference to the kernel timing are also removed.

In the after evolution tuning, if GP had enabled `DPER` (Section 8.12.2) then as well as tuning `BLOCK_W` and `ROWSperTHREAD` the autotuner tried values 1, 2, 3 and 4 for `DPER`. (In the case of the Tesla K20c, GP enabled `DPER` but its default value, 2, turned out to be optimal.)

8.12 Alternative Implementations

8.12.1 Avoiding Reusing Threads: XHALO

As mentioned in Section 8.8 each row of pixels is extended by five pixels at both ends. The original code reused the first ten threads of each block to calculate these ten halo values. Much of the kernel code is duplicated to deal with the horizontal halo. GPUs have a special type of parallel architecture which means many identical operations can be run in parallel but if the code branches in different directions part

of the hardware becomes idle. (This is known as thread divergence.) Thus diverting ten threads to deal with the halo causes all the remaining threads in the warp to become idle. (Each warp contains 32 threads.) Option XHALO allows GP to use ten additional threads which are dedicated to the halo. Thus each thread only deals with one pixel. In practise the net effect of XHALO is to disable the duplicated code so that instead of each block processing vertical stripes of 64 pixels, each block only writes stripes 54 pixels wide.

8.12.2 Parallel of Discrepancy offsets: DPER

The original code (Section 8.8) steps through sequentially 51 displacements of the right image with respect to the left. Modern GPUs allow many more threads and often it is best to use more threads as it allows greater parallelism and may improve throughput by increasing the overlap between computation and I/O. Instead of stepping sequentially one at a time through the 51 displacements, the DPER option allows 2, 3 or 4 displacement SSD values to be calculated in parallel. As well as increasing the number of threads, the amount of shared memory needed is also increased by the same factor. Nevertheless only one (the smallest) SSD value per pixel need be compared with the current smallest, so potentially saving some I/O. Although the volume of calculations is little changed, there are also potential saving since each DPER block uses almost the same data.

8.13 Parameters Accessible to Evolution

The GISMOE GP system [Langdon and Harman, 2015] was extended to allow not only code changes but also changes to C macro `#defines`. The GP puts the evolved values in a C `#include .h` file, which is complied along with the GP modified kernel code and the associated (fixed) host source code.

Table 8.1 shows the twelve configuration parameters. Every GP individual chromosome starts with these 12, which are then followed by zero or more changes to the code. Figure 8.16 page 206 contains an example GP individual, whereas Figures 8.12 page 202, 8.13 and 8.14 contain simplified schematics of GP individuals.

8.13.1 Fixed Configuration Parameters

8.13.1.1 OUT_TYPE

The return value should be in the range -1 to 50 (Section 8.10). Originally this is coded as a `float`. OUT_TYPE gives GP the option of trying other types. Notice,

Table 8.1 Evolvable configuration macros and constants

Name	Default	Options	Purpose
Cache preference	None	None, Shared, L1, Equal	L1 v. shared memory
-Xptxas -dlcm		' ', ca, cg, cs, cv	nvcc cache options
OUT_TYPE	`float`	`float, int, short int, unsigned char`	C type of output
STORE_disparityPixel	GLOBAL	GLOBAL, SHARED, LOCAL	
STORE_disparityMinSSD	GLOBAL	GLOBAL, SHARED, LOCAL	
DPER	disabled		Section 8.12.2
XHALO	disabled		Section 8.12.1
__mul24(a,b)	__mul24	__mul24, $\star$	fast 24-bit multiply
GPtexturereadmode	`Normalized Float`	`NormalizedFloat`, `ElementType`, none	Section 8.13.1.4
texturefilterMode	`Linear`	`Linear`, `Point`	
textureaddressMode		`Clamp`, `Mirror`, `Wrap`	
texturenormalized		0, 1	

since the data will probably be used on the GPU, we do not use the fact that the smaller data types take less time to transfer between GPU and host. (I.e. all fitness times, Section 8.14.5.2, are on the GPU.)

8.13.1.2 STORE_disparityPixel and STORE_disparityMinSSD

disparityPixel and disparityMinSSD are major arrays in the kernel. Stam coded them to lie in the GPU's slow off chip global memory. These configuration options give evolution the possibility of trying to place them in either shared memory or in local memory. Where the compiler can resolve local array indexes, e.g. as a result of unrolling loops, it can use fast registers in place of local memory.

8.13.1.3 __mul24

For addressing purposes, older GPU's included a fast 24 bit multiply instruction, which is heavily used in the original code. It appears that in the newer GPUs __mul24 may actually be slower than ordinary (32 bit) integer multiply. Hence we give GP the option of replacing __mul24 with ordinary multiply.

8.13.1.4 Textures

CUDA textures are intimately linked with the GPU's hardware and provide a wide range of data manipulation facilities (normalisation, default values, control of boundary effects and interpolation) which the original code does not need but is obliged to use. The left and right image textures are principally used because they provide caching (which was not otherwise available on early generation GPUs.) We

allowed the GP to investigate all texture options, including not using textures. Some combinations are illegal but the host code gives sensible defaults in these cases.

Unfortunately it is tricky to ensure access directly to the data and via a texture produce identical answers. Once cause of differences is there can be a $\frac{1}{2}$ pixel discrepancy between direct access (which treats the images as 2D arrays) and textures where reference point is the centre of the pixel. This leads to small differences between direct access and the original code. Whilst such slight differences make little difference to the outputs' appearance, even so such GP individuals are penalised by the fitness function (Section 8.14.5). This may have inhibited GP exploring all the data access options.

8.14 Evolvable Code

Following the standard GISMOE approach [Langdon and Harman, 2015], the evolutionary cycle is amended so that we start by creating a BNF grammar from the supplied source code and the GP evolves linear patches to the code (applied via the grammar) rather than trees, cf. Figure 8.5 page 191. The source code, including XHALO and DPER (Sections 8.12.1 and 8.12.2), is automatically translated line by line into the grammar (see Figure 8.11). Notice the grammar is not generic, it represents only one program, stereoKernel, and variants of it. The grammar contains 424 rules, 277 represent fixed lines of C++ source code. There are 55 variable lines, 27 IF and 10 of each of the three parts of C `for` loops. There are also five CUDA specific types:

1. `#pragma unroll` allows GP to control the nvcc compiler's loop unrolling. pragma rules are automatically inserted before each `for` loop but rely on GP to enable and set their values. Using the type constraints GP can either: remove it, set it to `#pragma unroll`, or set it to `#pragma unroll` *n* (where *n* is 1 to 11).
2. optvolatile CUDA allows shared data types to be marked as `volatile` which influences the compiler's optimisation. As required by the CUDA compiler, the grammar automatically ensures all shared variables are either flagged as `volatile` or none are.
   The remaining three CUDA types apply to the kernel's header.
3. optconst Each of kernel's scalar inputs can be separately marked as `const`.
4. optrestrict All of the kernel's array arguments can be marked with `__restrict__` This potentially helps the compiler to optimise the code. On the newest GPUs (SM 3.5) optrestrict allows the compiler to access read only arrays via a read only cache. Since both only apply if all arrays are marked `__restrict__`, the grammar ensures they all are or none are.
5. launchbounds is again a CUDA specific aid to code optimisation. By default the compiler must generate code that can be run with any numbers of threads. Since GP knows how many threads will be used, specifying it via `__launch_bounds__` gives the compiler the potential of optimising the code. `__launch_`

`bounds__` takes an optional second argument which refers to the number of blocks that are active per streaming multiprocessor SMX. How it is used is again convoluted, but the grammar allows GP to omit it, or set it to 1, 2, 3, 4 or 5.

8.14.1 Initial Population

Each member of the initial population is unique. They are each created by selecting at random one of the 12 configuration constants (Table 8.1) and setting it at random to one of its non-default values. As the population is created it becomes harder to find unique mutations and so random code changes are included as well as the configuration change. Table 8.2 summarises the GP parameters.

Table 8.2 Genetic programming parameters for improving stereoKernel

Representation:	Fixed list of 12 parameter values (Table 8.1) followed by variable list of replacements, deletions and insertions into BNF grammar
Fitness:	Run on a randomly chosen 320×240 monochrome stereo image pair. Compare answer & run time with original code and time its execution. See Sections 8.14.5 and 8.14.6.
Population:	Panmictic, non-elitist, generational. 100 members. New randomly chosen training sample each generation.
Parameters:	Initial population of random single mutants heavily weighted towards the kernel header and shared variables. 50% truncation selection. 50% crossover (uniform for fixed part, 2pt for variable). 50% mutation 25% mutation random change to fixed part. 25% add code mutation (one of: delete, replace, insert, each equally likely). No size limit. Stop after 50 generations.

8.14.2 Weights

Normally each line of code is equally likely to be modified. However, only as part of creating a diverse initial population, the small number of rules in the kernel header (i.e. launchbounds, optrestrict, optconst and optvolatile) are 1000 times more likely to be changed than the other grammar rules. (Forcing each member of the GP population to be unique is also only done in the initial population.) In future, it might be worthwhile ensuring GP does not waste effort changing CUDA code which can have no effect by setting the weights of lines excluded by conditional compilation to zero.

```
<KStereo.cuh_52>   ::= "__attribute__((global)) " <launchbounds_KStereo.cuh_52>
                       " void KERNEL(\n"
#kernel
<launchbounds_KStereo.cuh_52> ::= ""
<launchbounds_K0>  ::= "\n" "#ifdef DPER\n" "__launch_bounds__(BLOCK_W*dperblock)\n"
                       "#else\n" "__launch_bounds__(BLOCK_W)\n"   "#endif /*DPER*/\n"
                    ...
<launchbounds_K5>  ::= "\n" "#ifdef DPER\n" "__launch_bounds__(BLOCK_W*dperblock,5)\n"
                       "#else\n" "__launch_bounds__(BLOCK_W,5)\n" "#endif /*DPER*/\n"
<optrestrict_KStereo.cuh_52> ::= " __restrict__ "
#kernelarg
<KStereo.cuh_53>   ::= "OUTYPE *" <optrestrict_KStereo.cuh_52> "disparityPixel,\n"
<KStereo.cuh_54>   ::= <optconst_KStereo.cuh_54> "size_t out_Pitch,\n"
<optconst_KStereo.cuh_54> ::= "const "
<KStereo.cuh_55>   ::= "#ifdef GLOBAL_disparityMinSSD\n"
<KStereo.cuh_56>   ::= "int *" <optrestrict_KStereo.cuh_52> "disparityMinSSD,\n"
<KStereo.cuh_57>   ::= "#if OUT_TYPE != float_ && OUT_TYPE != int_\n"
<KStereo.cuh_58>   ::= <optconst_KStereo.cuh_58> "size_t out_pitch,\n"
<optconst_KStereo.cuh_58> ::= "const "
<KStereo.cuh_59>   ::= "#endif\n"
<KStereo.cuh_60>   ::= "#endif /*GLOBAL_disparityMinSSD*/\n"
                   ...
<KStereo.cuh_72>   ::= ")\n"
                   ...
<KStereo.cuh_141>  ::= " if" <IF_KStereo.cuh_141>
                       "  extra_read_val = BLOCK_W+threadIdx.x;\n"
#"if
<IF_KStereo.cuh_141> ::= "(threadIdx.x < (2*RADIUS_H))"
                   ...
<KStereo.cuh_158>  ::= <pragma_KStereo.cuh_158> "for("
                       <for1_KStereo.cuh_158> ";" "OK()&&"
                       <for2_KStereo.cuh_158> ";"
                       <for3_KStereo.cuh_158> ") \n"
#for
<pragma_KStereo.cuh_158> ::= ""
#pragma
<pragma_K0>        ::= "#pragma unroll \n"
<pragma_K1>        ::= "#pragma unroll 1\n"
                   ...
<pragma_K11>       ::= "#pragma unroll 11\n"
<for1_KStereo.cuh_158> ::= "i = 0"
<for2_KStereo.cuh_158> ::= "i<ROWSperTHREAD && Y+i < height"
<for3_KStereo.cuh_158> ::= "i++"
<KStereo.cuh_159>  ::= "{\n"
<KStereo.cuh_160>  ::= "" <_KStereo.cuh_160> "\n"
#other
<_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_162>  ::= "}\n"
```

Fig. 8.11 Fragments of BNF grammar used by GP. Most rules are fixed but rules starting with <_, <IF_, <for1_, <pragma_, etc. can be manipulated using rules of the same type to produce variants of stereoKernel. Lines beginning with # are comments.

8.14.3 Mutation

Half of mutations are made to the configuration parameters (Table 8.1). In which case one of the 12 configuration parameters is chosen uniformly at random and its current value is replaced by another of its possible values again chosen uniformly at random, see Figure 8.12. 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 8.13). There are three possible code mutations: delete a line of code, replace a line and insert a line. The replacement and inserted lines of code are copied from stereoKernel itself (via the grammar). Notice GP does not create code. It merely rearranges human written code.

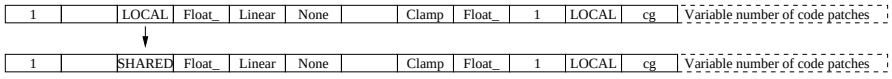

Fig. 8.12 Example of mutation to the configuration part at the start of a GP individual. Top: parent Bottom: offspring. The 12 configuration parameters are given in Table 8.1

<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. 8.13 Example of mutation to the variable length part of a GP individual. Patch <288>+<161> is appended to parent (top) causing in the child (bottom) a copy of source line 161 to be inserted before line 288 in the kernel source code. (For clarity the left hand part omitted and full grammar rule names simplified, e.g. to just the line numbers.)

8.14.4 Crossover

Crossover creates a new GP individual from two different members of the better half (Section 8.14.6) of the current population. The child inherits each of the 12 fixed parameters (Table 8.1) at random from either parent (uniform crossover [Syswerda, 1989], see Figure 8.14). Whereas in [Langdon and Harman, 2015] 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. On average, this gives no net change in length.

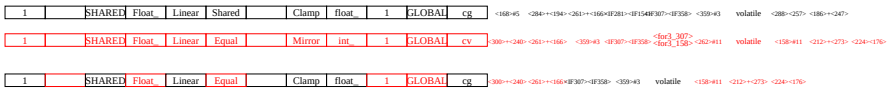

Fig. 8.14 Example of crossover. Parts of two above median parents (top and middle) recombined to yield a child (bottom).

8.14.5 Fitness

To avoid over fitting and to keep run times manageable, each generation one of the two hundred training images pairs is chosen [Langdon, 2010]. Each GP modified kernel in the population is tested on that image pair.

8.14.5.1 CUDA memcheck and Loop Overruns

Normally each GP modified kernel is run twice. The first time it is run with CUDA memcheck and with loop over run checks enabled. If no problems are reported by CUDA memcheck and the kernel terminates normally (i.e. without exceeding the limit on loop iterations) it is run a second time without these debug aids. Both memcheck and counting loop iterations impose high overheads which make timing information unusable. Only in the second run are the timing and error information used as part of fitness. If the GP kernel fails in either run, it is given such a large penalty, that it will not be a parent for the next generation.

When loop timeouts are enabled, the GP grammar ensures that each time a C++ `for` loop iterates a per thread global counter is incremented. If the counter exceeds the limit, the loop is aborted and the kernel quickly terminates. If any thread reaches its limit, the whole kernel is treated as if it had timed out. The limit is set to 100× the maximum reasonable value for a correctly operating good kernel.

8.14.5.2 Timing

Each of the streaming multiprocessor (SMXs) within the GPU chip has its own independent clock. On some GPUs `cudaDeviceReset()` resets all the clocks, this is not the case with the C2050. To get a robust timing scheme, which applies to all GPUs, each kernel block records both its own start and end times and the SMX unit it is running on. After the kernel has finished, for each SMX, the end time of the last block to use it and the start time of the first block to use it are subtracted to give the accurate duration of usage for each SMX. (Note to take care of overflow `unsigned int` arithmetic is used.) Whilst we do not compare values taken from clocks on different SMXs, it turns out to be safe to assume that the total duration of the kernel is the longest time taken by any of the SMXs used. (As a sanity check this GPU kernel time is compared to the, less accurate, duration measured on the

host CPU.) The total duration taken by the GP kernel (expressed as GPU clock tics divided by 1000) is the first component of its fitness.

8.14.5.3 Error

For each pixel in the left image the value returned by the GP modified kernel is compared with that given by the un-modified kernel. If they are different a per pixel penalty is added to the total error which becomes the second part of the GP individual's fitness.

If the unmodified kernel did not return a value (i.e. it was -1.0, cf. Section 8.10) the value returned by the GP kernel is also ignored. Otherwise, if the GP failed to set a value for a pixel, it gets a penalty of 200. If the GP value is infinite or otherwise outside the range of expected values (0..50) it attracts a penalty of 100. Otherwise the per pixel penalty is the absolute difference between the original value and the GP's value.

For efficiency, previously [Langdon and Harman, 2010] we batched up many GP generated kernels into one file to be compiled in one go. For simplicity, since we are using a more advanced version of nVidia's nvcc compiler, and GP individuals in the same population may need different compiler options, we did not attempt this. Typically it takes about 3.3 seconds to compile each GP generated kernel. Whereas to run the resulting StereoCamera program (twice see Section 8.14.5.1) takes about 2.0 seconds,

8.14.6 Selection

At the end of each generation we compare each mutant with the original kernel's performance on the same test case and only allow it to be a parent if it does well. In detail, it must be both faster and be, on average, not more than 6.0 per pixel different from the original code's answer. However mostly the evolved code passes both tests. At the end of each generation the population is sorted first by their error and then by their speed. The top 50% are selected to be parents of the next generation. Each selected parent creates one child by mutation (Section 8.14.3) and another by crossover with another selected parent (Section 8.14.4). The complete GP parameters are summarised in Table 8.2.

8.15 Results

The best individual from the last generation (50) was minimised to remove unneeded mutations which contributed little to its overall performance and retuned (Section 8.11). This reduced the length of the GP individual from 29 to 10. On

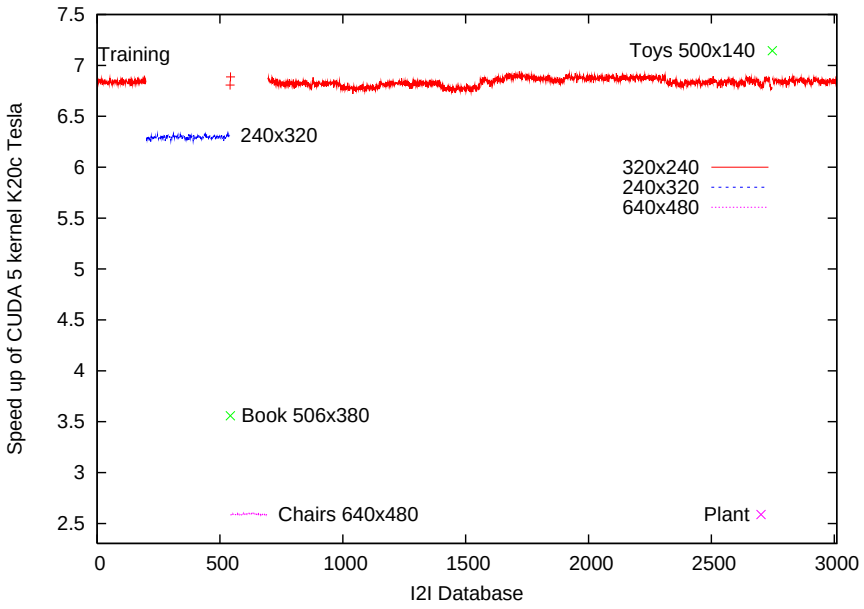

Fig. 8.15 Performance of GP improved K20c Tesla kernel on all 3010 stereo pairs in Microsoft's I2I database relative to original kernel on the same image pair on the same GPU. Fifty of first 200 pairs used in training. The evolved kernel is always much better, especially on images of the same size and shape as it was trained on.

the Tesla K20c, on average, across all 2516 I2I 320×240 stereo image pairs, GP sped up the original StereoCamera code almost seven fold. (The mean speed up is 6.837 ± 0.04.) By reducing `ROWSperTHREAD` from the original 40 to 5, pretuning (Section 8.11) itself gave a factor of 2.4 fold speed up. The original value of `BLOCK_W` (64) and the default value of DPER (2) were optimal for the Tesla K20c. I.e. the GP code changes gave another factor of almost three on top of the parameter tuning. The speedup of the improved K20c kernel on all of the I2I stereo images is given in Figure 8.15. The speed up for the other five GPUs varied in a similar way to the K20c. Finally, notice typically there is very little difference in performance across the images of the same size and shape as the training data

8.16 Evolved Tesla K20c CUDA Code

The best of generation 50 individual changes 6 of the 12 fixed configuration parameters (Table 8.1) and includes 23 grammar rule changes. After removing less useful components (Section 8.11) four configuration parameters were changed and there were six code changes. See Figures 8.16 and 8.17. The complete code is given in the appendix (pages 218–221).

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. 8.16 Best GP individual in generation 50 of K20c Tesla run after minimising, Section 8.11, removed less useful components. (Auto-tuning made no further improvements.) Top line (normal font) are four non-default values for the 12 fixed configuration parameters. Six code changes shown in `tt` font.

*int * __restrict__ disparityMinSSD,* //Global disparityMinSSD not kernel argument
volatile `extern __attribute__((shared)) int col_ssd[];`
volatile `int* const reduce_ssd = &col_ssd[(64 )*2 -64];`
`#pragma unroll 11`
if(X < width `&&` *Y < height)* replaced by `if(dblockIdx==0)`
__syncthreads();
`#pragma unroll 3`

Fig. 8.17 Evolved changes to K20c Tesla StereoKernel. (Produced by GP grammar changes in Figure 8.16). Highlighted `code` is inserted. Code in *italics* is removed. For brevity, except for the kernel's arguments, disparityPixel and disparityMinSSD changes from global to shared memory are omitted. The appendix, pages 218–221, gives the complete source code.

The evolved configuration parameters mean that DPER is enabled and the new kernel calculates two disparity values in parallel (Section 8.12.2), disparityPixel and disparityMinSSD are stored in shared memory (Section 8.13.1.2) and XHALO is enabled (Section 8.12.1).
The final code changes, Figure 8.17, are:

- disable volatile, Section 8.14.
- insert `#pragma unroll 11` before the `for` loop that steps through the `ROWSperTHREAD` - 1 other rows (Section 8.8).
- insert `#pragma unroll 3` before the `for` loop that writes each of the `ROWS perTHREAD` rows of disparityPixel from shared to global memory. Its not clear why evolution chose to ask the nvcc compiler to unroll this loop (which is always executed 5 times) only 3 times. But then when nvcc decides to do loop unrolling is obscure anyway.
- Mutation `<_KStereo.cuh_161>+<_KStereo.cuh_224>` causes line 224 to be inserted before line 161. Line 224 potentially updates local variable `ssd`, however `ssd` is not used before the code which initialises it. It is possible that the compiler spots that the mutated code cannot affect anything outside the kernel and simply optimises it away. During minimisation removing this mutation gave a kernel whose run time was exactly on the removal threshold.
- Mutation `<IF_KStereo.cuh_326><IF_KStereo.cuh_154>` replaces `X < width && Y < height` by `dblockIdx==0`. This replace 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. GP has found a way of improving the GPU kernel by removing a redundant expression.
The original purposed 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 GP 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. As well as reducing the number of instructions, removing synchronisation calls potentially allows greater overlapping of computation and I/O leading to an overall saving.

8.17 Discussion

Up to Intel's Pentium, Moore's Law [Moore, 1965] had applied not only to the doubling of the number of transistors on a silicon chip but also to exponential rises in clock speeds. Since 2005 mainstream processor clock speeds have remained fairly much unchanged. However Moore's Law continues to apply to the exponential rise in the number of available logic circuits. This has driven the continuing rise of parallel multi-core computing. In mainstream computing, GPU computing continues to lead in terms of price v. performance. However GPGPU computing [Owens *et al.*, 2008] (and parallel computing in general) is still held back by the difficulty of high-performance parallel programming [Langdon, 2011; Merrill *et al.*, 2012].

When programming the GPU, in addition to the usual programming tasks, there are other hardware specific choices, e.g. where to store data. Even for the expert it is difficult to find optimal choices for these while simultaneously programming. [Merrill *et al.*, 2012] propose heavy use of templates in kernel code in an effort to separate algorithm coding for data storage etc. However templates are in practise even harder to code and current versions of the compiler cannot optimally make choices for the programmer. As Section 8.11 shows, it can be feasible to remove the choice of key parameters (typically block size) from the programmer. Instead their code is run with all feasible values of the parameter and the best chosen. There are already tools to support this. Such enumerative approaches are only feasible with a small number of parameters. The GP approach is more scalable and allows mixing both parameter tuning and code changes. To be fair, we should say at present all the approaches are still at the research stage rather than being able to assist the average graphics card programmer.

Future new requirements of StereoCamera might be dealing with: colour, moving images (perhaps with time skew), larger images, greater frame rates and running on mobile robots, 3D telephones, virtual reality gamesets or other low energy portable

devices. We can hope our GP system could be used to automatically create new versions tailored to new demands and new hardware.

In some cases modern hardware readily gives on line access to other important non-functional properties (such as power or current consumption, temperature and actual clock speeds). Potentially these might also be optimised by GP. [White *et al.*, 2008] showed it can be possible to use GP with a cycle-level power level simulator to optimise small programs for embedded systems. ([Schulte *et al.*, 2014a] recently extended this to large open source every day programs.) Here we work with the real hardware, rather than simulators, however real power measurements are not readily available with all our GTX and Tesla cards.

Many computers, including GPUs, and especially in mobile devices, now have variable power consumption. Thus reducing execution time can lead to a proportionate reduction in energy consumption and hence increase in battery life, since as soon as the computation is done the computer can revert to its low power idle hibernating state. [Yao *et al.*, 1995; Han *et al.*, 2010; Radulescu *et al.*, 2014] consider other ways of tuning of the processor's clock speed (which might be combined with software improvements).

Another promising extension is the combined optimisation for multiple functional and non-functional properties [Colmenar *et al.*, 2011]. Initial experiments hinted that NSGA-II [Deb *et al.*, 2002; Langdon *et al.*, 2010] finds it hard to maintain a complete Pareto front when one objective is much easier than the others. Thus a population may evolve to contain many fast programs which have lost important functionality while slower functional program are lost from the population. Newer multi-objective GAs or alternative fitness function scalings may address this.

The newer versions of CUDA also include additional tools (e.g. CUDA race check) which might be included as part of fitness testing.

The supplied kernel code contains several hundred lines of code. It may be that this only just contains enough variation for GP's cut-and-past operations (Section 8.14.3). We had intended to allow GP to also use code taken from the copious examples supplied by nVidia with CUDA (see Section 8.5) but so far this has not been tried.

nVidia and other manufactures are continuing to increase the performance, economy and functionality of their parallel hardware. There are also other highly parallel and low power chips with diverse architectures (e.g. multicore CPUs, FPGAs, Intel Xeon Phi, mobile and RFID devices [Andreopoulos, 2013]). These trends suggest the need for software to be ported [Langdon and Harman, 2010] or to adapt to new parallel architectures will continue to increase.

One of the great success for modular system design has been the ability to keep software running whilst the underlying hardware platforms have gone through several generations of upgrades. In some cases this has been achieved by freezing the software, even to the extent of preserving binaries for years. In practise this is not sufficient and software that is in use is under continual and very expensive maintenance. There is a universal need for software to adapt.

8.18 Conclusions

We have reviewed published work on using genetic programming on software. Initially we showed examples where genetic programming was able to evolve real software from scratch. In some cases, e.g. by automatically creating bespoke applications tailored to particular tasks, the GP generated code improves on generic human written code.

Even now, code evolved from scratch tends to be small. The GGGP (grow and graft) system, described in Section 8.6, is a potential way around the problem. GGGP still evolves small new components but also uses GP to graft them into much bigger human written codes, thus create large hybrid software.

Similarly the CUDA gzip example (Section 8.3) showed small but valuable units of code can be effectively automatically ported by evolving new code to match the functionality of the existing code, even if it is written in a different language or executes on different hardware. Indeed auto bugfixing (Section 8.2.4), Bowtie2, NiftyReg and StereoCamera also use the existing code as the de facto specification of the functionality of the to be evolved software.

The work on miniSAT (Section 8.5) shows GP can potentially scavenge not just code from the program it is improving but code from multiple programs by multiple authors. This GP plastic surgery [Barr *et al.*, 2014] created in a few hours an award winning version of miniSAT tailored to solving an import software engineering problem, for which it was better than generic versions of miniSAT which has been optimised by leading SAT solving experts for years.

Another promising area is evolving software to meet multiple conflicting requirements. Indeed GP's potential ability to present the software designer with a Pareto trade-off front of different measures of code performance [Harman *et al.*, 2012], may be one avenue that leads most quickly to the wide spread adoption of genetic programming for software improvement. One can imagine a system which shows a range of programs with different speed versus memory requirements, which invites the software designer to choose a suitable trade-off *before* any manual coding starts. Few, if any projects, once the location of their implementation on the trade off space is known, i.e. coding is almost complete, can afford to reject their initial design choices and start again from scratch. Instead typically only relatively small performance changes can be made within the straight jacket of the original design. Further the GP system could consider not only conventional alternatives (e.g. speed v. memory) but also aspects required by mobile computing, such as network bandwidth, power, battery life, and even quality of solutions. It would be very useful to be able to see credible results of design decisions before implementation starts, even if the machine generated code is totally discarded and the designer insists on human coding.

GI is definitely a growth area with the first international event on Genetic Improvement, GI-2015, being held in Madrid along side the main evolutionary computation conference, GECCO, and GI papers being well represented in the SSBSE software engineering conference series as well as gaining best paper prizes in the top software engineering conference and human competitive awards.

Mostly we have described in detail an application of our BNF grammar based GP system, in which a population of code patches is automatically evolved to create new versions of parallel code to run on graphics hardware. The evolving versions are continuously compared with the original, which is treated as the de facto specification, by running regression tests on both and frequently changing the example test used. The fitness function penalises deviation from the original but rewards faster execution. The GP evolves code which exploits the abilities of the hardware the code will run on. The StereoCamera system was specifically written by nVidia's image processing expert to show off their hardware and yet GP is able to improve the code for hardware which had not even been designed when it was originally written yielding almost a seven fold speed up in the graphics kernel.

Sources and Datasets

Le Goues' bug fixing system (Section 8.2.4) is available on line: `http://genprog.cs.virginia.edu/` The grammar based genetic programming systems for gzip (Section 8.3), Bowtie2 (Section 8.4), StereoCamera (Section 8.7 onwards) and 3D Brain scan registration (NiftyReg Section 8.7 page 190) are available on line via `ftp.cs.ucl.ac.uk`. (For the MiniSAT genetic improvement code, Section 8.5, please contact Dr. Petke directly.) The StereoCamera code is in file `genetic/gp-code/StereoCamera_1_1.tar.gz` and training images are in `StereoImages.tar.gz` The new code is available in `StereoCamera_v1_1c.zip`.

Acknowledgements

I am grateful for the assistance of njuffa, Istvan Reguly, vyas of nVidia, Ted Baker, and Allan MacKinnon.

GPUs were given by nVidia. Funded by EPSRC grant EP/I033688/1.

References

Ackling *et al.*, 2011. Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In Natalio Krasnogor et al., editors, *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1427–1434, Dublin, Ireland, 12-16 July 2011. ACM.

Altschul *et al.*, 1997. Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped BLAST and PSI-BLAST a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

Andreopoulos, 2013. Yiannis Andreopoulos. Error tolerant multimedia stream processing: There's plenty of room at the top (of the system stack). *IEEE Transactions on Multimedia*, 15(2):291–303, Feb 2013. Invited Paper.

Archanjo and Von Zuben, 2012. Gabriel A. Archanjo and Fernando J. Von Zuben. Genetic programming for automating the development of data management algorithms in information technology systems. *Advances in Software Engineering*, 2012.

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.

Arcuri and Yao, 2014. Andrea Arcuri and Xin Yao. Co-evolutionary automatic programming for software development. *Information Sciences*, 259:412–432, 2014.

Banzhaf *et al.*, 1998. Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, January 1998.

Barr *et al.*, 2014. Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In Alessandro Orso, Margaret-Anne Storey, and Shing-Chi Cheung, editors, *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, Hong Kong, 16-12 Nov 2014. ACM.

Berarducci *et al.*, 2004. Patrick Berarducci, Demetrius Jordan, David Martin, and Jennifer Seitzer. GEVOSH: Using grammatical evolution to generate hashing functions. In R. Poli et al., editors, *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, 26-30 June 2004.

Bradbury and Jalbert, 2010. Jeremy S. Bradbury and Kevin Jalbert. Automatic repair of concurrency bugs. In Massimiliano Di Penta et al., editors, *Proceedings of the 2nd International Symposium on Search Based Software Engineering (SSBSE '10)*, Benevento, Italy, 7-9 September 2010. Fast abstract.

Branke *et al.*, 2006. Jurgen Branke, Pablo Funes, and Frederik Thiele. Evolutionary design of en-route caching strategies. *Applied Soft Computing*, 7(3):890–898, June 2006.

Burke *et al.*, 2013. Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and Rong Qu. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, December 2013.

Buxton *et al.*, 2001. B. F. Buxton, W. B. Langdon, and S. J. Barrett. Data fusion by intelligent classifier combination. *Measurement and Control*, 34(8):229–234, October 2001.

Cadar *et al.*, 2010. Cristian Cadar, Peter Pietzuch, and Alexander L. Wolf. Multiplicity computing: a vision of software engineering for next-generation computing platform applications. In Kevin Sullivan, editor, *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 81–86, Santa Fe, New Mexico, USA, 7-11 November 2010. ACM.

Cody-Kenny and Barrett, 2013. Brendan Cody-Kenny and Stephen Barrett. The emergence of useful bias in self-focusing genetic programming for software optimisation. In Guenther Ruhe and Yuanyuan Zhang, editors, *Symposium on Search-Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 306–311, Leningrad, August 24-26 2013. Springer. Graduate Student Track.

Colmenar *et al.*, 2011. J. Manuel Colmenar, Jose L. Risco-Martin, David Atienza, and J. Ignacio Hidalgo. Multi-objective optimization of dynamic memory managers using grammatical evolution. In Natalio Krasnogor et al., editors, *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1819–1826, Dublin, Ireland, 12-16 July 2011. ACM.

Cotillon *et al.*, 2012. Alban Cotillon, Philip Valencia, and Raja Jurdak. Android genetic programming framework. In Alberto Moraglio et al., editors, *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012*, volume 7244 of *LNCS*, pages 13–24, Malaga, Spain, 11-13 April 2012. Springer Verlag.

Darwin, 1859. Charles Darwin. *The Origin of Species*. John Murray, penguin classics, 1985 edition, 1859.

Deb *et al.*, 2002. K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr 2002.

DeMillo and Offutt, 1991. Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

Estebanez *et al.*, . Cesar Estebanez, Yago Saez, Gustavo Recio, and Pedro Isasi. Automatic design of noncryptographic hash functions using genetic programming. *Computational Intelligence*. Early View (Online Version of Record published before inclusion in an issue).

Feldt, 1998. Robert Feldt. Generating diverse software versions with genetic programming: an experimental study. *IEE Proceedings - Software Engineering*, 145(6):228–236, December 1998. Special issue on Dependable Computing Systems.

Feldt, 1999. Robert Feldt. Genetic programming as an explorative tool in early software development phases. In Conor Ryan and Jim Buckley, editors, *Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering*, pages 11–20, University of Limerick, Ireland, 12-14 April 1999. Limerick University Press.

Forrest *et al.*, 2009. Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In Guenther Raidl et al., editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954, Montreal, 8-12 July 2009. ACM. Best paper.

Fredericks and Cheng, 2013. Erik M. Fredericks and Betty H. C. Cheng. Exploring automated software composition with genetic programming. In Christian Blum et al., editors, *GECCO '13 Companion: Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, pages 1733–1734, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

Freitas, 1997. Alex A. Freitas. A genetic programming framework for two data mining tasks: Classification and generalized rule induction. In John R. Koza et al., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 96–101, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

Gabel and Su, 2010. Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

Goldberg, 1989. David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

Han *et al.*, 2010. Xin Han, Tak-Wah Lam, Lap-Kei Lee, Isaac K.K. To, and Prudence W.H. Wong. Deadline scheduling and power management for speed bounded processors. *Theoretical Computer Science*, 411(40-42):3587–3600, 2010.

Handley, 1994. S. Handley. On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 154–159, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

Haraldsson and Woodward, 2014. Saemundur O. Haraldsson and John R. Woodward. Automated design of algorithms and genetic improvement: contrast and commonalities. In John Woodward et al., editors, *GECCO 2014 4th workshop on evolutionary computation for the automated design of algorithms*, pages 1373–1380, Vancouver, BC, Canada, 12-16 July 2014. ACM.

Harman *et al.*, 2012. 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.*, 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.

Harman, 2011. Mark Harman. Software engineering meets evolutionary computation. *Computer*, 44(10):31–39, October 2011. Cover feature.

Holland, 1992. John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, 1992. First Published by University of Michigan Press 1975.

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.

Imamura and Foster, 2001. Kosuke Imamura and James A. Foster. Fault-tolerant computing with N-version genetic programming. In Lee Spector et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, page 178, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

Imamura *et al.*, 2003. Kosuke Imamura, Terence Soule, Robert B. Heckendorn, and James A. Foster. Behavioral diversity and a probabilistically optimal GP ensemble. *Genetic Programming and Evolvable Machines*, 4(3):235–253, September 2003.

Jacob, 2001. Christian Jacob. *Illustrating Evolutionary Computation with Mathematica*. Morgan Kaufmann, 2001.

Karasek *et al.*, 2011. Jan Karasek, Radim Burget, and Ondrej Morsky. Towards an automatic design of non-cryptographic hash function. In *34th International Conference on Telecommunications and Signal Processing (TSP 2011)*, pages 19–23, Budapest, 18-20 August 2011.

Katz and Peled, 2013. Gal Katz and Doron Peled. Synthesizing, correcting and improving code, using model checking-based genetic programming. In Valeria Bertacco and Axel Legay, editors, *Proceedings of the 9th International Haifa Verification Conference (HVC 2013)*, volume 8244 of *Lecture Notes in Computer Science*, pages 246–261, Haifa, Israel, November 5-7 2013. Springer. Keynote Presentation.

Kessentini *et al.*, 2011. Marouane Kessentini, Wael Kessentini, Houari Sahraoui, Mounir Boukadoum, and Ali Ouni. Design defects detection and correction by example. In *19th IEEE International Conference on Program Comprehension (ICPC 2011)*, pages 81–90, Kingston, Canada, 22-24 June 2011.

Kim *et al.*, 2013. Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering (ICSE 2013)*, pages 802–811, San Francisco, USA, 18-26 May 2013.

Kordon, 2010. Arthur K. Kordon. *Applying Computational Intelligence How to Create Value*. Springer, 2010.

Kovacic and Sarler, 2014. Miha Kovacic and Bozidar Sarler. Genetic programming prediction of the natural gas consumption in a steel plant. *Energy*, 66(1):273–284, 1 March 2014.

Koza, 1992. John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT press, 1992.

Langdon and Barrett, 2004. W. B. Langdon and S. J. Barrett. Genetic programming in data mining for drug discovery. In Ashish Ghosh and Lakhmi C. Jain, editors, *Evolutionary Computing in Data Mining*, volume 163 of *Studies in Fuzziness and Soft Computing*, chapter 10, pages 211–235. Springer, 2004.

Langdon and Buxton, 2001. W. B. Langdon and B. F. Buxton. Genetic programming for combining classifiers. In Lee Spector et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 66–73, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

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, 2015. 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 Nordin, 2000. W. B. Langdon and J. P. Nordin. Seeding GP populations. In Riccardo Poli et al., editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 304–315, Edinburgh, 15-16 April 2000. Springer-Verlag.

Langdon and Poli, 2005. William B. Langdon and Riccardo Poli. Evolving problems to learn about particle swarm and other optimisers. In David Corne et al., editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 81–88, Edinburgh, UK, 2-5 September 2005. IEEE Press.

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

Langdon, 2004. W. B. Langdon. Global distributed evolution of L-systems fractals. In Maarten Keijzer et al., editors, *Genetic Programming, Proceedings of EuroGP'2004*, volume 3003 of *LNCS*, pages 349–358, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag.

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, 2011. W. B. Langdon. Graphics processing units and genetic programming: An overview. *Soft Computing*, 15:1657–1669, August 2011.

Langdon, 2012. W.B. Langdon. Creating and debugging performance CUDA C. In Francisco Fernandez de Vega, Jose Ignacio Hidalgo Perez, and Juan Lanchares, editors, *Parallel Architectures and Bioinspired Algorithms*, volume 415 of *Studies in Computational Intelligence*, chapter 1, pages 7–50. Springer, 2012.

Langdon, 2014. William B. Langdon. Genetic improvement of programs. In Franz Winkler et al., editors, *16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2014)*, pages 14–19, Timisoara, 22-25 September 2014. IEEE. Keynote.

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, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, January-February 2012.

Lohn and Hornby, 2006. Jason D. Lohn and Gregory S. Hornby. Evolvable hardware using evolutionary computation to design and optimize hardware systems. *IEEE Computational Intelligence Magazine*, 1(1):19–27, February 2006.

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.

Mahajan and Ali, 2008. Anjali Mahajan and M S Ali. Superblock scheduling using genetic programming for embedded systems. In *7th IEEE International Conference on Cognitive Informatics, ICCI 2008*, pages 261–266, August 2008.

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.

Moore, 1965. Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 19 1965.

Nguyen *et al.*, 2013. Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: program repair via semantic analysis. In Betty H. C. Cheng and Klaus Pohl, editors, *35th International Conference on Software Engineering (ICSE 2013)*, pages 772–781, San Francisco, USA, May 18-26 2013. IEEE.

O'Neill and Ryan, 1999. Michael O'Neill and Conor Ryan. Automatic generation of caching algorithms. In Kaisa Miettinen et al., editors, *Evolutionary Algorithms in Engineering and Computer Science*, pages 127–134, Jyväskylä, Finland, 30 May - 3 June 1999. John Wiley & Sons.

O'Neill and Ryan, 2001. Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, August 2001.

O'Neill and Ryan, 2003. Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.

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.

Pappa *et al.*, 2014. Gisele L. Pappa, Gabriela Ochoa, Matthew R. Hyde, Alex A. Freitas, John Woodward, and Jerry Swan. Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 15(1):3–35, March 2014.

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.

Podgornik *et al.*, 2011. Bojan Podgornik, Vojteh Leskovsek, Miha Kovacic, and Josef Vizintin. Analysis and prediction of residual stresses in nitrided tool steel. *Materials Science Forum*, 681, Residual Stresses VIII:352–357, March 2011.

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).

Radulescu *et al.*, 2014. Vlad Radulescu, Stefan Andrei, and Albert M. K. Cheng. A heuristic-based approach for reducing the power consumption of real-time embedded systems. In Franz Winkler, editor, *16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2014)*, Timisoara, 22-25 September 2014. Pre-proceedings.

Reynolds, 2011. Craig Reynolds. Interactive evolution of camouflage. *Artificial Life*, 17(2):123–136, Spring 2011.

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.

Romero *et al.*, 2013. Juan Romero, Penousal Machado, and Adrian Carballal. Guest editorial: special issue on biologically inspired music, sound, art and design. *Genetic Programming and Evolvable Machines*, 14(3):281–286, September 2013. Special issue on biologically inspired music, sound, art and design.

Ryan, 1999. Conor Ryan. *Automatic Re-engineering of Software Using Genetic Programming*, volume 2 of *Genetic Programming*. Kluwer Academic Publishers, 1 November 1999.

Schulte *et al.*, 2010. Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 313–316, Antwerp, 20-24 September 2010. 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.

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.

Stam, 2008. Joe Stam. Stereo imaging with CUDA. Technical report, nVidia, V 0.2 3 Jan 2008.

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.

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.

Weimer, 2013. Westley Weimer. Advances in automated program repair and a call to arms. In Guenther Ruhe and Yuanyuan Zhang, editors, *Symposium on Search-Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 1–3, Leningrad, August 24-26 2013. Springer. Invited keynote.

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.

Wilkerson and Tauritz, 2010. Josh L. Wilkerson and Daniel Tauritz. Coevolutionary automated software correction. In Juergen Branke et al., editors, *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1391–1392, Portland, Oregon, USA, 7-11 July 2010. ACM.

Xiao *et al.*, 2012. Liyuan Xiao, Carl K. Chang, Hen-I Yang, Kai-Shin Lu, and Hsin yi Jiang. Automated web service composition using genetic programming. In *36th Annual IEEE Computer Software and Applications Conference Workshops (COMPSACW 2012)*, pages 7–12, Izmir, 16-20 July 2012.

Yao *et al.*, 1995. Frances Yao, Alan Demers, and Scott Shenker. A scheduling model for reduced cpu energy. In *36th Annual Symposium on Foundations of Computer Science*, pages 374–382. IEEE, Oct 1995.

Yoo, 2012. Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. In Gordon Fraser et al., editors, *4th Symposium on Search Based Software Engineering*, volume 7515 of *Lecture Notes in Computer Science*, pages 244–258, Riva del Garda, Italy, September 28-30 2012. Springer.

Zhu and Kulkarni, 2013. Ling Zhu and Sandeep Kulkarni. Synthesizing round based fault-tolerant programs using genetic programming. In Teruo Higashino et al., editors, *Proceedings of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2013)*, volume 8255 of *Lecture Notes in Computer Science*, pages 370–372, Osaka, Japan, November 13-16 2013. Springer.

8.19 StereoKernel tuned for K20c Tesla

In addition to the complete Stereo Camera system `StereoCamera_v1_1c.zip` contains the following CUDA kernel. The modifications to the openVidia CUDA Stereo Camera code distributed by SourceForge are also described in Section 8.16 (pages 205 to 207).

```
/*******
stereoKernel
Now for the main stereo kernel:  There are four parameters:
disparityPixel points to memory containing the disparity value (d)
for each pixel.
width & height are the image width & height, and out_pitch specifies
the pitch of the output data in words (i.e. the number of floats
between the start of one row and the start of the next.).
disparityMinSSD removed by GP
*********/

__attribute__((global)) void stereoKernel(
  // pointer to the output memory for the disparity map
  float * __restrict__ disparityPixel,
  // the pitch (in pixels) of the output memory for the disparity map
  const size_t out_pitch,
  const int width,
  const int height,
  unsigned int * __restrict__ timer,   //For GP timing only
  int * __restrict__ sm_id             //For GP timing only
)
{
FIXED_init_timings(timer,sm_id);       //For GP timing only
extern __attribute__((shared)) float disparityPixel_S[];

int* const disparityMinSSD = (int*)&disparityPixel_S[ROWSperTHREAD*BLOCK_W];
 // column squared difference functions
int* const col_ssd = &disparityMinSSD[ROWSperTHREAD*BLOCK_W];
float d; // disparity value
float d0,d1;
float dmin;

int diff;    // difference temporary value
int ssd;     // total SSD for a kernel
float x_tex; // texture coordinates for image lookup
float y_tex;
int row;     // the current row in the rolling window
int i;       // for index variable
const int dthreadIdx = threadIdx.x % BLOCK_W;
const int dblockIdx  = threadIdx.x / BLOCK_W;

//bugfix force subsequent calculations to be signed
const int X = (__mul24(blockIdx.x,(BLOCK_W-2*RADIUS_H)) + dthreadIdx);
const int ssdIdx = threadIdx.x;
int* const reduce_ssd = &col_ssd[(BLOCK_W )*dperblock-BLOCK_W];
const int Y = (__mul24(blockIdx.y,ROWSperTHREAD));
```

```
//int extra_read_val = 0; no longer used
//if(dthreadIdx < (2*RADIUS_H)) extra_read_val = BLOCK_W + ssdIdx;

// initialize the memory used for the disparity and the disparity difference
//Uses first group of threads to initialise shared memory
if(threadIdx.x<BLOCK_W-2*RADIUS_H)
if(dblockIdx==0)
if(X<width )
{
  for(i = 0;i<ROWSperTHREAD && Y+i < height;i++)
  {
    // initialize to -1 indicating no match
    disparityPixel_S[i*BLOCK_W +threadIdx.x] = -1.0f;
    //ssd += col_ssd[i+threadIdx.x];
    disparityMinSSD[i*BLOCK_W +threadIdx.x] = MIN_SSD;
  }
}
__syncthreads();

x_tex = X - RADIUS_H;
for(d0 = STEREO_MIND;d0 <= STEREO_MAXD;d0 += STEREO_DISP_STEP*dperblock)
{
  d = d0 + STEREO_DISP_STEP*dblockIdx;
  col_ssd[ssdIdx] = 0;

  // do the first row
  y_tex = Y - RADIUS_V;
  for(i = 0;i <= 2*RADIUS_V;i++)
  {
    diff = readLeft(x_tex,y_tex) - readRight(x_tex-d,y_tex);
    col_ssd[ssdIdx] += SQ(diff);
    y_tex += 1.0f;
  }
  __syncthreads();

  // now accumulate the total
  if(dthreadIdx<BLOCK_W-2*RADIUS_H)
  if(X < width && Y < height)
  {
    ssd = 0;
    for(i = 0;i<=(2*RADIUS_H);i++)
    {
      ssd += col_ssd[i+ssdIdx];
    }
  }
  if(dblockIdx!=0) reduce_ssd[threadIdx.x] = ssd;
  __syncthreads();

  //Use first group of threads to set ssd to smallest SSD for d1<d0+dperblock
  if(threadIdx.x<BLOCK_W-2*RADIUS_H)
  if(X < width && Y < height)
  {
    dmin = d;
```

```
    d1 = d + STEREO_DISP_STEP;
    for(i = threadIdx.x+BLOCK_W;i < blockDim.x;i += BLOCK_W) {
      if(d1 <= STEREO_MAXD && reduce_ssd[i] < ssd) {
        ssd = reduce_ssd[i];
        dmin = d1;
      }
      d1 += STEREO_DISP_STEP;
    }
    //if ssd is smaller update both shared data arrays
    if( ssd < disparityMinSSD[0*BLOCK_W +threadIdx.x])
    {
     disparityPixel_S[0*BLOCK_W +threadIdx.x] = dmin;
     disparityMinSSD[0*BLOCK_W +threadIdx.x] = ssd;
    }
  }
  __syncthreads();

  // now do the remaining rows
  y_tex = Y - RADIUS_V; // this is the row we will remove
  #pragma unroll 11
  for(row = 1;row < ROWSperTHREAD && (row+Y < (height+RADIUS_V));row++)
  {
    // subtract the value of the first row from column sums
    diff = readLeft(x_tex,y_tex) - readRight(x_tex-d,y_tex);
    col_ssd[ssdIdx] -= SQ(diff);

    // add in the value from the next row down
    diff =  readLeft(x_tex,  y_tex + (float)(2*RADIUS_V)+1.0f) -
           readRight(x_tex-d,y_tex + (float)(2*RADIUS_V)+1.0f);
    col_ssd[ssdIdx] += SQ(diff);
    y_tex += 1.0f;
    __syncthreads();

    if(dthreadIdx<BLOCK_W-2*RADIUS_H)
    if(X<width && (Y+row) < height)
    {
      ssd = 0;
      for(i = 0;i<=(2*RADIUS_H);i++)
      {
        ssd += col_ssd[i+ssdIdx];
      }
    }
    if(dblockIdx!=0) reduce_ssd[threadIdx.x] = ssd;
    __syncthreads();

    //Use 1st group threads to set ssd/dmin to smallest SSD for d1<d0+dperblock
    if(threadIdx.x<BLOCK_W-2*RADIUS_H)
    if(dblockIdx==0)
    {
      dmin = d;
      d1 = d + STEREO_DISP_STEP;
      for(i = threadIdx.x+BLOCK_W;i < blockDim.x;i += BLOCK_W) {
        if(d1 <= STEREO_MAXD && reduce_ssd[i] < ssd) {
          ssd = reduce_ssd[i];
```

```
          dmin = d1;
        }
        d1 += STEREO_DISP_STEP;
      }
      //if smaller SSD found update shared memory
      if(ssd < disparityMinSSD[row*BLOCK_W +threadIdx.x])
      {
        disparityPixel_S[row*BLOCK_W +threadIdx.x] = dmin;
        disparityMinSSD[row*BLOCK_W +threadIdx.x] = ssd;
      }
    }//endif first group of thread
  }// for row loop
}// for d0 loop

//Write answer in shared memory to global memory
if(threadIdx.x<BLOCK_W-2*RADIUS_H)
if(dblockIdx==0)
if(X < width) {
#pragma unroll 3
  for(row = 0;row < ROWSperTHREAD && (row+Y < height);row++)
  {
    disparityPixel[__mul24((Y+row),out_pitch)+X] =
                disparityPixel_S[row*BLOCK_W +threadIdx.x];
  }
}
FIXED_report_timings(timer,sm_id);      //For GP timing only
}
```