WCCI (CIGPU special session) 2010, Pilar Sobrevilla, *et al., Eds.*, Barcelona, 18-23 July, IEEE.

Evolving a CUDA Kernel from an nVidia Template

W. B. Langdon and M. Harman

***Abstract*— Rather than attempting to evolve a complete program from scratch we demonstrate genetic interface programming (GIP) by automatically generating a parallel CUDA kernel with identical functionality to existing highly optimised ancient sequential C code (gzip). Generic GPGPU nVidia kernel C++ code is converted into a BNF grammar. Strongly typed genetic programming uses the BNF to generate compilable and executable graphics card kernels. Their fitness is given by running the population on a GPU with randomised subsets of training data itself derived from gzip's SIR test suite. Back-to-back validation uses the original code as a test oracle.**

I. Introduction

The goal of genetic interface programming (GIP) is to evolve automatically small interfaces between or components of much larger systems which have already been created by traditional software engineering techniques. There are many areas of software where quite small software units are required to fit a well define interface. In many cases it will be simplest to write the code. However as systems get more complex finding a person with skills appropriate to the two or more potentially very large subsystems being interfaced becomes more difficult. This is especially true where the subsystems are written in different languages or by different suppliers or reside on different hosts.

It becomes still more difficult for the programmer when the requirements change or involve multiple competing interests. In a traditional server host there may be little incentive to produce compact code, whereas in a single user mobile device space and battery life might need to be juggled against less severe response time. In principle a multi-objective automated system could come up with a range of Pareto optimal implementations allowing the software engineer to choose the solution appropriate to the current requirement. Indeed when requirements change the software maintenance engineer might choose a different solution from the existing non-dominated set, or even re-run the evolution with new requirements.

As a very first demonstration of these ideas we have taken a well constrained problem: evolving an nVidia CUDA graphics card kernel to parallelise existing sequential code. Creating high quality GPGPU kernels is known to be difficult and there are, as yet, few experts in CUDA. In software engineering this is not uncommon. Indeed it has been known for systems to be constructed, not in the computer language of choice, but the language where there is a ready supply of skilled programmers.

The Software-artifact Infrastructure Repository (SIR) [Hutchins *et al.*, 1994] contains a version of gzip suitable for large scale regression testing and a test suite for it. gzip works by scanning the file to be compressed for sequences of bytes which occur multiple times and replacing them in the compressed file by a shorter code. The longer the replaced repeated sequences are the better the compression. Searching for the longest matches consumes most of the CPU time used by gzip. In principle, for every byte in the input file, gzip needs to find all the parts of the file which match it. It then remembers the longest match, which may then become one of the compression codes. gzip incorporates hashing and numerous heuristics and restricts the search window to limit the computational load. These have been tuned by hand. A possible future investigation would be to consider re-tuning gzip in the light of modern demands. However we have not tried to do this, instead we seek a CUDA C++ kernel which is not only compatible with existing versions of gzip and compressed files but insist the new graphics code yield identical outputs.

The next section describes how we start with code supplied by the manufacture nVidia for a totally different application and use it as a template for a gzip string matching CUDA kernel. Section III-A describes its conversion into a context free BNF grammar. Section III-B says how the existing SIR test suite for gzip was converted into a GP fitness function [Poli *et al.*, 2008]. The evolution and validation of the automatically created CUDA gzip kernel are described in Sections IV and V. We finish with a discussion including future work (Section VI) and our conclusions (Section VII).

II. Analysing Human Examples

We start, as a human novice might, by studying the available example codes. CUDA 2.3 comes with 67 examples coded in C++. `scan_naive_kernel.cu` was selected. The comments were stripped from it and it is was converted to take the right inputs and operate on the right types for a gzip string matching kernel. The human created starting point for the automatic generation of a CUDA kernel to replace the gzip `longest_match` routine is shown in Figure 1.

As a first step towards the evolution of CUDA code, the template kernel was deliberately kept simple. By removing direct access to `__shared__` data and threading information, thousands of GP individuals can be individually tested in parallel on the GPU without the risk of them interfering with each other. This does, however, restrict the types of kernels which can be evolved. Again it must be stressed that at this stage we seek only to show the approach is possible. As we said in the introduction, we do not yet try to demonstrate automatically generating efficient, high performance code or improved compression by finding more or better matches or other trade-offs between multiple objectives.

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

```
__device__ void scan_naive(int *g_odata, const uch *g_idata, const int strstart1,
const int strstart2)
{
    //extern  __shared__  float temp[];
    int thid = 0; //threadIdx.x;
    int pout = 0;
    int pin = 1;
    int offset = 0;
    int num_elements = 258;
    <3var> /*temp[pout*num_elements+thid]*/ = (thid > 0) ? g_idata[thid-1] : 0;
    for (offset = 1; offset < num_elements; offset *= 2)
    {
        pout = 1 - pout;
        pin  = 1 - pout;
        //temp[pout*num_elements+thid] = temp[pin*num_elements+thid];
        <3var> =  g_idata[strstart+pin*num_elements+thid];
        if (thid >= offset)
        <3var> += g_idata[strstart+pin*num_elements+thid - offset];
    }
    return <3var> ;
}
```

Fig. 1. Template supplied to GIP and from which GIP creates gzip C++ code. Based on scan_naive_kernel.cu supplied with CUDA 2.3 by nVidia. Simplified by removing direct access to thread index `threadIdx.x`, `__shared__` data `temp` (and hence `__syncthreads()`) and replacing function inputs and direct write to `g_odata[threadIdx.x]` with `return`. Also convert from `float` to `int` and `uch` data types. The three places where `temp` was written are replaced by writing to an `int` local variable (`<3var>`). Where elements of `temp` were read, `temp` is replaced by the only other array `g_idata`. `n` is renamed `num_elements`. To avoid complications between statements inside for loop and outside, the loop control variable `offset` is declared at the start of the function rather than in the `for` statement.

There are too many occurrences of the letter "n" in the grammar, therefore the variable `n` was renamed `num_inputs`. As is usual with GP [Langdon, 1998], values, including constants, can evolve. Thus the grammar provides the means for introducing a small number of `int` and `uch` constants which can be used directly or as part of evolved expressions. 0, 1 and 2 come from the original CUDA code, whereas 258 comes from gzip. A lot of care was taken about how the grammar deals with the three lines supplied by nVidia which update an element of the (now removed) `__shared__` array `temp`. However, this part of the grammar does not affect how the evolved solution work (cf. Figure 5). The interested reader can find details in the technical report: [Langdon and Harman, 2010].

III. Preparing Genetic Programming

In addition to the standard five steps described by [Koza, 1992], the strongly typed [Montana, 1995] grammar based GP [Langdon and Harrison, 2009] needs a BNF grammar. The next section describes the motivation behind and the mechanism for, the conversion of the human template (Figure 1) into a context free grammar (Figures 2 and 3). The complete grammar spans four pages so is given in full in a technical report [Langdon and Harman, 2010].

A. The Grammar

The automatic generation of CUDA Kernel's was in part motivated by last year's prize winning work [Weimer *et al.*, 2009; Forrest *et al.*, 2009] in which bug fixes were automatically evolved by genetic programming. And earlier work by [Arcuri and Yao, 2008]. Whereas in [Weimer *et al.*, 2009] the GP works by automatically reusing lines of human written code from elsewhere in the faulty code, we use a BNF grammar to describe a much smaller unit of human code. (I.e. the nVidia code used as the template described above in Section II and given in Figure 1.) The grammar also allows the GP much finer control. Instead of reusing whole statements, the grammar allows fragments of lines and even individual expressions to be be manipulated. At the same time the combination of grammar and strong typing ensures (unlike [Weimer *et al.*, 2009]) that each of the genetically manipulated individuals is legal CUDA C++ code. They compile and run and produce answers. There is of course, as usual, no guarantee that they do anything sensible.

We use gawk in conjunction with Unix shell scripts. Firstly, to help create the grammar and secondly, to do all the genetic manipulations (crossover and mutation) and C++ code generation [Langdon and Harrison, 2009].

As with our previous work on using GP to evolve variants of complete human written C code for mutation testing [Langdon *et al.*, 2010b], we start by converting the human written code (Figure 1) line-by-line into a BNF grammar (Figure 2). Next comes the grammar start rule (`<start>`) and the hierarchical sequence of rules which link it to the individual lines (i.e. the complete CUDA kernel, Figure 3). In the work on inserting mutations into existing code [Langdon *et al.*, 2010b], only small changes were permitted and the structure of the code, including the order of the lines of code was fixed to to be identical of the original human written code. However, since we seek to evolve new code, this new grammar allows the lines to be omitted, to be reversed and structures, like `for` and `if`, to be nested.

To ensure the new code compiles, some of the structure of the template (Figure 2) is enforced by the grammar. Although the C++ syntax allows fairly free ordering of variable declarations, variables must still be declared. To allow this constraint to be simply enforced by the grammar, the grammar is written so that all the variables are declared at the start of the CUDA kernel. Similarly, the grammar ensures that it ends with a `return` statement.

The right hand side of each BNF production can be either a binary alternative rule. (Meaning the grammar rule must be expanded into one of the two given alternatives.) Alternatively a rule is a single list of terminals and rules to be expanded, without alternatives. Two alternatives per rule allows each GP individual to be represented internally as a binary tree.

We adopt the convention that the alternative closer to the original code is usually the first alternative. The first alternative often consists mostly of terminals and so is the smaller grammar sub-tree. This is useful when the genetic operations start to run into space or depth limits. If such limits are reached, the genetic operations can chose the smaller option in an expansion without the need to investigate all options and back track to the smallest one. A typical example of this convention is rule `<line6e>`.

Grammar rule `<line6e>` deals with declaring `int pout`. `<line6e>` can either expand to give code identical to line 6 of the template or something similar, but slightly more complicated. In the second case, the initial value of `int pout` is given by an evolvable constant (rather than the fixed value 0) by rule `<intconst>`. By using `<intconst>` the grammar ensures the constant is of the same type as the variable.

Line 8 of the template is fairly complicated. (Full details are given in technical report [Langdon and Harman, 2010].) Briefly it may be omitted (i.e. replaced by `""`) used unchanged or modified. Rule `<line8.0.0>` breaks line 8 into its four components. Again, each of the components has alternatives which are the same as the template code and others which progressively generalise it. E.g. `<line8.2.1.2.2>` gives the most general form of the original code `thid > 0` as `<intvar> <compare> <intconst>`.

Line 10 contains a `for` loop. Again the grammar allows generalisations of the template. For example, the loop continuation test in line 10 uses <, rule `<line10.2>` says < may be used but also allows any comparison. As another example, the template uses `*= 2` in the iteration step but rule `<line10.4>` allows ++ etc.

Line 16 is a conditional statement. Again the grammar allows it to be included, omitted, modified and moved. The final part of the grammar contains most of the generalisation. The grammar still tends to favour expressions of the same type and rules like `<intspecialexpr>` encourage the reuse of expressions created by the authors of CUDA.

B. The Fitness Function

1) The Training Data: SIR supplies a test suite of 214 tests for gzip. We used 211 of version 1.4 of these. (The remaining three had proved to be unsuitable for using in automated scripts, e.g. due to non-determinism introduced by behaviour which depends on the detailed timing of events in the test script.) We extended our gzip test bed (used in mutation testing [Langdon *et al.*, 2010b]) to instrument all the inputs and outputs of gzip's `longest_match` routine. When running our SIR test suite `longest_match` is called 1 599 028 times and makes 42 472 851 searches.

gzip uses a sliding window of 64kb in which it searches for repeats of each string in the file to be compressed. Principally by using hashing, it greatly reduces the 2 147 450 880 potential matches. Nevertheless, for each byte in the file, `longest_match` still has to search every item on a particular hash chain. For pragmatic reasons gzip limits the length of matches to between three and 258 bytes.

The number of times `longest_match` is given a hash chain of a given length falls very rapidly with its length [Langdon and Harman, 2010, Fig. 9]. So, although the maximum length of the hash chain is 1588, the median is only eight and in $\approx$ 200 000 calls there is only one item on the hash chain to be checked. To reduce the volume of test data and give a more uniform spread of training data, for each of the 211 SIR tests only the first of each call of `longest_match` with a particular number of parallel searches was kept and thus available to train the GP. This yielded 29 315 examples.

In each generation 100 of the 29 315 examples were randomly chosen (without reselection). The fitness of every evolved CUDA kernel in the population is assessed on the same one hundred examples. Thus, the number of times each GP individual is called will be the same in a given generation but vary randomly between generation. Each time it is called it is given the address of the current string and that of a putative match. (Actually a point in the file with the same hash value.) The individual should return the number of bytes for which the two locations match. The distribution of correct answers for the whole of the training data is very non-uniform. Due to interaction with gzip's hashing algorithm there are only 407 strings of a single byte and none with two bytes. In contrast there are 100 333 searches where the strings don't match at all and 2 222 039 where only their first three bytes are identical. Again each generation is subject to different sampling noise.

2) Compiling and Linking the Evolving Kernels: Unlike our earlier experiments with evolving DNA regular expressions [Langdon *et al.*, 2010a], the time to compile the CUDA code dominates the fitness evaluation process. ([Harding and Banzhaf, 2009] describe a way of reducing the compilation overhead by running the compiler in parallel across a cluster of workstations.) There is some initial overhead with starting the compiler. It appears that the compilation time grows non-linearly as the volume of code increases. Initial experiments suggested that a good compromise between startup overhead

```
<line2>    ::=  "__device__ int kernelXXX(const uch *g_idata, const int strstart1,
               const int strstart2)\n"
<line3>    ::=  "{\n"
<line5>    ::=  "int thid = 0;\n"
<line6>    ::=  "int pout = 0;\n"
<line7>    ::=  "int pin = 1;\n"
<line71>   ::=  "int offset = 0;\n"
<line72>   ::=  "int num_elements = 258;\n"
<line8>    ::=  <3var> "= (thid > 0) ? G_idata(" <strstart> "thid-1) : 0;\n"
<line10>   ::=  "for (offset = 1; offset < num_elements; offset *= 2)\n"
<line11>   ::=  "{\n" "if(!ok()) break;\n"
<line12>   ::=  "pout = 1 - pout;\n"
<line13>   ::=  "pin = 1 - pout;\n"
<line15>   ::=  <3var> "= G_idata(" <strstart> "pin*num_elements+thid);\n"
<line16>   ::=  "if (thid >= offset)\n"
<line17>   ::=  <3var> "+= G_idata(" <strstart> "pin*num_elements+thid - offset);\n"
<line18>   ::=  "}\n"
<line20>   ::=  "return" <3var> ";\n"
<line21>   ::=  "}\n"
<3var>     ::=  "thid"
<strstart> ::= "strstart1+"|"strstart2+"
```

Fig. 2. First part of grammar used to evolve CUDA matches kernel for gzip. These rules allow little variation but reproduce the initial template, Figure 1. XXX is replaced by the GP individual's identification number. ok() is a macro (#define ok() ((nfor++<2000)? 1 : 0)) which prevents infinite loops. <3var> could be any of the three variables pout, num_elements or thid but only thid was implemented. The vertical bar | in the last line means rule <strstart> must be replaced by either "strstart1+" or "strstart2+". Other rules, e.g. <line20>, are expanded to a list of strings and/or rule names. Strings, e.g. "return", cannot be expanded further. Whereas rules, e.g. <3var>, must be expanded.

```
<start>       ::=  <line2> <line3> <line5> <line6e> <line7e> <line71e> <line72e>
                   <line8e> <line10-20> <line21>
<line10-20>   ::=  <line10-18> <line20e>
<line10-18>   ::=            ""|<line10-18a>
<line10-18a>  ::=  <line10e> <line11> <forbody> <line18>
<line12-17>   ::=  <line12-17a>|<line12-17b>
<line12-17a>  ::=  <line12-13> <line15-17>
<line12-17b>  ::=  <line15-17> <line12-13>
<line12-13>   ::=  <line12-13a>|<line12-13b>
<line12-13a>  ::=  <line12e> <line13e>
<line12-13b>  ::=  <line13e> <line12e>
<line15-17>   ::=  <line15-17a>|<line15-17b>
<line15-17a>  ::=  <line15e> <line16-17>
<line15-17b>  ::=  <line16-17> <line15e>
<line16-17>   ::=  <line16-17a>|<line17.0>
<line16-17a>  ::= <line16-17.1>|<line16-17.2>
<line16-17.1> ::=  <line16e> <line17.0>
<line16-17.2> ::=  <line16e> "{{" <forbody> "}}"
<forbody>     ::=   <line12-17>|<forbody.1>
<forbody.1>   ::=  <forbody.1a>|<forbody.1b>
<forbody.1a>  ::=  <line8e> <line12-17>
<forbody.1b>  ::=  <line12-17> <line8e>
```

Fig. 3. Second part of grammar. <start> is the start rule of the grammar. Each C++ gzip kernel is created by recursively expanding <start> until all the rules (denoted by < >) have been replaced by ordinary text. These rules allow lines of code (defined in Figure 2) to be used, excluded, reversed and to be nested. Note using the first of each binary option tends to give a GP individual closer to the template, Figure 1. The function header, int declarations (<line2> ... <line72e>) and the return statement (<line20>) are always included. The complete grammar is given in technical report [Langdon and Harman, 2010].

and non-linear increase with size was when about one hundred kernels were compiled together. Therefore, the gawk scripts which perform crossover and mutation also split the population into ten parcels each containing one hundred GP individuals.

Each parcel contains a 100-way switch statement and code which unpacks the arguments supplied by the host PC and directs the overall kernel to the right GP individual and returns its answer to the host PC. The ten parcels are compiled by the CUDA 2.3 nvcc compiler into ten object files which are linked into a single executable. (The compilation might be 18% faster if the switch statement was moved to the host C++ code. Similarly using the -O1 nvcc command line switch might also speed up nvcc by 16%. The times given below were obtained with default settings.)

3) Testing the Evolving Kernels: The executable runs on the host Linux computer. It reads files holding the test data. Using gzip code, it calculates the answers that should be returned. (This is done only once.)

For efficiency, the whole GP population is run on each test example and their answers saved. This allows the data needed for each training example, which typically includes 64kb of data from the file to be compressed, to be loaded only once for all 1000 members of the population.

As mentioned above, the population is compiled into ten object files. For the purposes of training the GP, each is a separate CUDA kernel. The host launches these ten kernels in series. Their answers are returned to the host before the next is launched. In principle, this could be done in parallel. Each of the ten kernels runs its one hundred GP individuals across the whole of the current hash chain in parallel. While the exact number of parallel execution threads running on the GPU will vary across the training examples, the mean hash chain length is 293, so on average each kernel is run $293 \times 100 = 29\,300$ times in parallel.

For each generation, the GPU will launch the 10 kernels for each of the 100 training examples. On average fitness testing takes 6.5 seconds per generation (which includes host file processing). In contrast generating and compiling the population takes 49.0 seconds (most of which is spent in the nvcc compiler) [Langdon and Harman, 2010, Fig. 11].

4) Error based Fitness and Penalties: For each test case the GPU will return $1000\times$the length of the hash chain answers. (I.e. up to 1 588 000 integers.) Each is one of the population's answer to the question how many bytes match between two points in the current 64kb window onto the file being compressed. As described in Section III-B.3, we already know gzip's answer to this question. This allows us to define an error based fitness function. Fitness is essentially the sum of the absolute error between the answer returned by each GP and that calculated by the gzip code.

Some values are much more common in the training data than others. A simple strategy for obtaining a good fitness is to choose one of these and to always return this constant value. To deter this and so limit the number of children allocated to constants a penalty of 10 000 times the number of training examples was introduced. E.g. in the initial generation the 100 fitness cases together cover 29 328 putative string matches. This means the minimum constant penalty is 293 280 000. (We take care to avoid `int` overflow.) The penalty is is doubled if the constant was 2, tripled if it was 1 and quadrupled if it was 0.

TABLE I

STRONGLY TYPED GENETIC PROGRAMMING PARAMETERS FOR AUTOMATICALLY CREATING C++ GZIP NVIDIA KERNELS.

Function:	Binary rules in the BNF grammar (Figures 2–3).
Terminals:	Grammar rules without alternatives (Figures 2–3). Array `g_idata` indexes are forced to their legal range by reducing modulo array size.
Fitness:	Sum of absolute errors between evolved answer and that given by gzip across pseudo random sample of 100 of 29 315 examples drawn from compressing files in the SIR test suite (Section III-B.1).
Selection:	Generational. 4 members tournaments. New sample of training data each generation.
Population:	1000
Initial pop:	Ramped half-and-half 2:6
Parameters:	50% subtree crossover, 50% subtree mutation. Crossover and mutation points are chosen uniformly (i.e. without a function bias [Koza, 1992]) treating each grammar rule as a distinct type and ensuring each offspring is genetically different from both its parents. Max depth 100, No size limit. `ok()` (Figure 2) limits all `for` loops to $\leq$ 2000 iterations.
Termination:	1 000 generations

In the initial population 70.7% of random CUDA kernels return a fixed value. As evolution progresses the fraction of such useless kernels in the population eventually settles near 6.9%. Most of these always return zero, despite the fact it carries the largest fitness penalty.

5) Debug: There were many troubles during development. In the end it was decided to sacrifice the last member of the GP population and replace it with a non-evolved code fragment which simply reflected the training data. This allowed the host code to verify that the training data had indeed been copied successfully to the GeForce 295 GTX nVidia GPU graphics card, the kernel had been run and had been able to read its inputs, and it had succeeded in returning its answer to the host. If any of these steps fail, the Unix process is aborted until the problem is manually resolved.

IV. EVOLVING SOLUTIONS

The details of our grammar based GP are given in Table I. The terminal and function sets are determined by the grammar described in Section III-A. The error based fitness function was described in the previous section.

The evolution of the population in the first run is plotted in Figure 4. The first solution was found in generation 50. Five runs out of ten reported solutions by generation 100.

Given the BNF grammar all that is needed to construct each program is the sequence of choices made at the binary alternative nodes. The rest of the parse tree and hence the program itself can be inferred from these. If we treat the parse tree as the important underlying skeleton of the evolved code and look at two statistics (the number of nodes in the tree and their maximum depth from the `<start>` node)

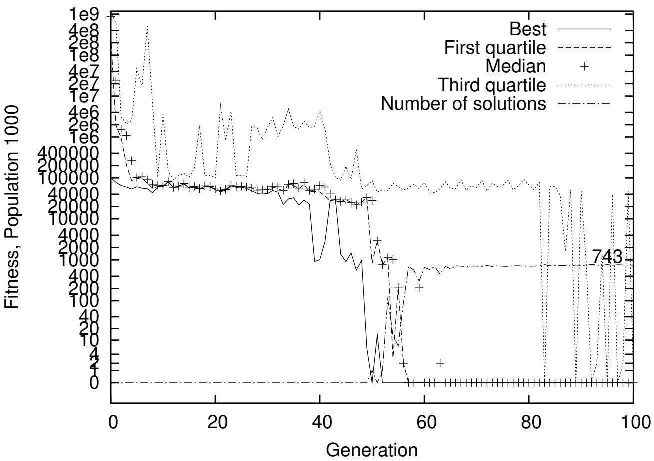

Fig. 4. Absolute error + constant penalty across $\approx$ 29 000 training examples (varies between generations). Note non-linear vertical scale. Programs which make no errors are first created in generation 50 (49 minutes) Typically after generation 57 about $\frac{3}{4}$ of the population pass all the tests.

we see the usual patterns of GP. The trees increase in size and depth. When we look at the evolution of both size and depth [Langdon and Harman, 2010, Fig. 15] we see the same behaviour as traditional GP benchmarks [Langdon and Poli, 2002, Chapter 11] [Langdon, 2000]. It is remarkable that a strongly typed grammar based GP which highly constrains evolution to ensure syntactically correct, executable C++ code yields such similar behaviour to tree GP where crossover may occur between any two points in the parent trees. This is reinforced when we look at the evolution of the distribution of the number of choice nodes in the individuals. Again we see the characteristic "humped" distribution with a long upper tail, with both the location of the hump and tails increasing over time. [Dignum and Poli, 2007] shows (for a totally different GP system) that this is a Lagrange distribution. Note even in generation 100, when most of the population are highly fit, mutation is producing small unfit children.

Circular lattices [Daida *et al.*, 2005] [Poli *et al.*, 2008, p 136] are a nice alternative way to display GP trees. They emphasise the shape of the tree rather than its contents. (Contrast Figures 6 and 7, which both refer to the underlying parse tree of the same evolved CUDA kernel.) The `<start>` rule is notionally the center of the circular display and we do indeed see its ten subsequent rules clustered around it. However, some of these (e.g. `<line2>`) are always immediately expanded into terminals of the grammar and cannot be expanded further. Other rules (e.g. `<line10-20>`) must be followed by other rules and large subtrees are often attached to them. Thus the rules of the BNF grammar give rise to asymmetric trees. However, this only exacerbates the natural tendency, we have already noted, for GP to evolve randomly shaped asymmetric trees [Langdon and Poli, 2002, Chapter 11] [Langdon, 2000].

Figure 4 shows after generation 57 the fitness of a large part of the population converges on the same fitness value. (I.e. fitness value zero. Zero is the best possible value since it means no errors have been reported.) However even after 1000 generations the population does not fully converge. The GP mutation and crossover operations remain disruptive and their active search means they continues to produce populations in which $\approx$ 23.7% of children make errors.

As with fitness, the population also converges in the sense that many parse trees have similar (rather than identical) shapes. Again, the strongly typed GP trees behave as would be expected of an ordinary tree based GP. (Animations displaying these three views of these evolving populations can be found on the GIP web pages.)

V. Validating The Solutions

The evolved kernel passes all the tests in generation 50. It does what it was evolved to do: run on the graphics card and provide answers that emulate those needed by gzip.

Notice that, although it skips the second byte, in its context in gzip and in particular the way gzip's hashing works, this is very unlikely to make any difference. Indeed when the evolved kernel is compiled as part of gzip's `longest_match` routine the new code produces identical answers. This was verified in a version of gzip containing both old and evolved code and compressing `gzip_1.4.tar` (19 322 880 bytes) which contains the complete release for SIR including the whole SIR test suite. However, it might, in principle, be possible to find an example where it fails.

On average, only every fifth generation contains training examples with strings which match for exactly one byte. The first such generation after the first solution has been evolved is generation 55. Generation 55 contains seven programs which pass all its tests. The last one was chosen as being the simplest to explain. It is given in Figure 5. The corresponding grammar parse tree is given in Figures 6 and 7. Apart from the *correct* initial value for the `for` loop control variable (and the irrelevant assignments to variable `thid`) it is identical to the kernel found five generations earlier. (The nvcc compiler automatically removes code which has no impact on the kernel's output.) The generation 55 kernel checks all the bytes in both strings. Like the solution evolved in generation 50 it succeeds on the whole of `gzip_1.4.tar` (19 322 880 bytes). It also passes contrived tests designed to check for one byte errors run outside gzip.

Obviously such naive kernels do not provide any speed up.

VI. Discussion

gzip is a venerable Unix utility which still betrays its roots in 16-bit micro computing. It has been highly optimised and, in particular, the string match finding code has been carefully contorted by skilled human programmers to get the best from traditional single CPU systems. It contains several heuristics which are designed to give a good tradeoff between execution time, and degree of compression obtained on typical ASCII Unix text files. Doubtless better tradeoffs could be obtained for other circumstances. For example, where the files to be compressed are specialised or when

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

Fig. 5. C++ code of the gzip CUDA kernel automatically generated by GIP.

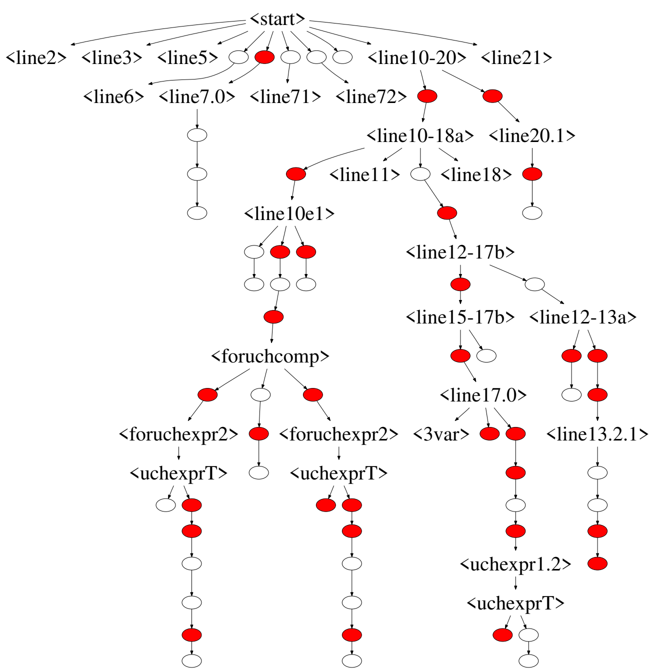

Fig. 6. Path through the grammar, c.f. Figures 2 and 3, taken by GIP to create the gzip CUDA kernel evolved in generation 55. Figure 5 gave the resulting C++ program. Ovals indicate binary decision rules. With shaded ovals the second option was used.

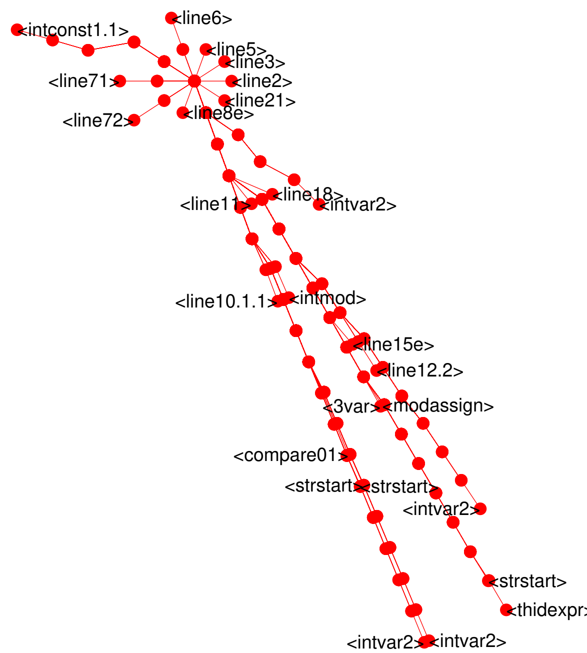

Fig. 7. Grammar expansion for the gzip CUDA C++ kernel evolved in generation 55. Identical to Figure 6 but displayed as a circular lattice [Poli *et al.*, 2008, p 136] [Daida *et al.*, 2005]. The last rule in each branch of the BNF grammar tree is labeled.

running on new hardware (such as remote robotic platforms) where energy consumption rather than compression speed is paramount. Relaxing limitations inherited from its 16-bit heritage might also allow different tradeoffs between memory used by gzip, CPU time and size of compressed output. All of these involve a small fraction of gzip which is heavily used. Existing profiling tools can automatically identify such regions in much bigger legacy systems.

Although a pure approach might require total automation, it may be that people will be less hesitant to adopt a pragmatic approach in which many novel "outside-the-box" tradeoffs between complex objectives are tried automatically. Inspecting these Pareto optimal evolved solutions might inspire human engineers to create and deploy a trusted interface incorporating multiple complex design compromises which they might not have tried had they not been suggested by evolution.

We have used gzip as a demonstration system for an approach which seeks not to re-engineer complete systems but to automatically generate code for tiny fraction of them. Given an existing system and its test suite, we have the bare bones to evolve a replacement for part of it. The existing system can be used as the "Gold standard" for the desired functionality. There is no need to create a formal specification. Each trial replacement can be simply compared with the existing system. We have taken the approach of requiring the replacement code to run on new hardware but there might be other, indeed multiple, objectives for the new plugin.

For efficiency, gzip deliberately relaxes its own objective of finding the longest match between strings in a file. This means sometimes it uses a match even though it is not the longest. However, although the search may have taken a long time it is relatively quick to verify the reported match is indeed a match. Indeed the debug version of gzip, has such a check permanently enabled. It is not uncommon for operations which are expensive in the forward direction to have a (relatively) cheap inverse check, or sanity check. E.g. after performing statistical calculations (mean, standard deviation) on large numbers, it is cheap to double check that their variance is non-negative.

We have adapted [Weimer *et al.*, 2009]'s approach, and used the manufacturer's supplied code as a template to guide the evolutionary process. We can see that in the general case there will always be the existing system to use as a guide. Whilst nVidia supplied only 67 modestly sized codes the volume of code typically available is much bigger. Even so such code may not be too large to prevent automatic analysis. Indeed this may be the next aspect of GIP we seek to demonstrate.

Whilst the grammar is firmly based on the nVidia supplied example, we have incorporated little information from gzip. Another approach to be tried in the future might be to investigate more automated ways of combining information gleaned from multiple sources.

VII. CONCLUSIONS

Automatic programming has been a goal of Computer Science for many years. With millions of people employed programming computers this still seems very remote. Nevertheless by pragmatically adopting less lofty ambitions, progress is being made in automating software engineering and particularly software maintenance [Arcuri and Yao, 2008; Weimer *et al.*, 2009]. The long-term goal of GIP is to automate the construction of small quantities of high value code (rather than to create whole systems). Particularly either code which is critical to the efficiency of the whole program or is critical to the success of a new system by linking together traditional codes. We have started with a small but achievable goal, where we have taken a crucial part of a venerable C program and GIP has ported it to new hardware.

Whilst automatic bug fixing has been demonstrated on a number of real C and C++ codes, this is the first time computational intelligence has been used to automatically create an nVidia CUDA kernel which runs in parallel on a state of the art graphics card as part of a legacy system. Section II described the context free grammar created from example code supplied by nVidia, whilst Section III explained the strongly typed grammar based genetic programming (GP) system. The results in Section IV show it is possible to evolve C++ code. The automatically created kernels have been tested (Section V) back-to-back with the original code many millions of times without error.

ACKNOWLEDGMENT

I would like to thank Simon Harding of Memorial University and Gernot Ziegler of nVidia.

REFERENCES

[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*, Hong Kong, 1-6 June 2008. IEEE Press.

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

[Dignum and Poli, 2007] Stephen Dignum and Riccardo Poli. Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In Dirk Thierens *et al.*, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1588–1595, London, 7-11 July 2007. ACM 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.

[Harding and Banzhaf, 2009] Simon L. Harding and Wolfgang Banzhaf. Distributed genetic programming on GPUs using CUDA. In Ignacio Hidalgo, Francisco Fernandez, and Juan Lanchares, editors, *Workshop on Parallel Architectures and Bioinspired Algorithms*, Raleigh, USA, September 13 2009.

[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.

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

[Langdon and Harman, 2010] W. B. Langdon and M. Harman. Evolving gzip matches kernel from an nVidia CUDA template. Technical Report TR-10-02, Department of Computer Science, King's College London, London, WC2R 2LS, UK, February 2010.

[Langdon and Harrison, 2009] W. B. Langdon and A. P. Harrison. Evolving DNA motifs to predict GeneChip probe performance. *Algorithms in Molecular Biology*, 4(6), 19 March 2009.

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

[Langdon *et al.*, 2010a] W. B. Langdon, Olivia Sanchez Graillet, and A. P. Harrison. Automated DNA motif discovery. arXiv, 30 January 2010.

[Langdon *et al.*, 2010b] William B. Langdon, Mark Harman, and Yue Jia. Efficient multi objective MC, GA, GP search and higher order mutation testing. *Journal of Systems and Software*. Submitted.

[Langdon, 1998] William B. Langdon. *Genetic Programming and Data Structures*. Kluwer, 1998.

[Langdon, 2000] William B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119, April 2000.

[Montana, 1995] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

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

[Weimer *et al.*, 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.