Genetic Improvement of Genetic Programming

William B. Langdon
CREST, Computer Science, UCL,
London, WC1E 6BT, UK

***Abstract*—GISMOE BNF grammar based GI is applied to optimise run time of the tree interpreter in the fastest single computer floating point genetic programming system, GPavx. Up to two fold speed up is obtained. Performance varies with tree size. The GI version of Singleton's C++ GPquick is demonstrated on random trees of up to 79 million opcodes on Intel AVX512 SIMD parallel compute servers.**

***Index Terms*—Testing, GP, GI, Dogfooding, Advanced Vector eXtensions (AVX) Single Instruction Multiple Data (SIMD), SBSE**

I. Introduction

We apply our Genetic Improvement (GI) system to our new parallel version of Genetic Programming (GP) [1]. Such "eating your own dogfood" [2] is rare and this is the first time GI has been applied to speed up GP. In the process evolution overturned more than 25 years of established wisdom.

Recently we manually extended Andy Singleton's GPquick system [3] for parallel operation on a new generation of Intel AVX 512 multi-core servers [1]. Our goal was to study evolution [4] over far longer than usual GP runs. Indeed some runs reached a million generations and continued to find fitness improvements many thousands of generations after GP runs are usually terminated [5]. In the process trees with hundreds of millions of nodes were evolved. With run time in weeks, we were keen to improve GPavx's efficiency. As expected, the new interpreter was the performance bottleneck. Excluding a few specialised or image processing benchmarks running on graphics hardware, at up to 149 billion GP operations per second ($149\ 10^9$ GPop/s), GPavx is the fastest single computer GP system [6, Tab. 3]. Nevertheless we set out to see if genetic improvement could speed it up.

Following the background (next section), Sections III and IV described our grammar system and fitness function. Section V describes four progressive experiments starting with the simplest part of the interpreter and ending with running six different conditional compilation options for the whole tree GP AVX interpreter. Sections VI and VII deal with new post-evolution processing and validation. Section VIII discusses the background of earlier attempts to speed up GP before we conclude (in Section IX) that evolution can over come considerable noise, has upset established wisdom and can indeed speed up the fastest GP system on the planet.

II. Background

A. Intel AVX512 float vector[16] Parallel Instructions

The AVX instruction set was originally introduced into the Intel x86 family of processors via specialist accelerator Xeon Phi cards in response to increasing use of nVidia GPUs for general purpose computing on GPUs (GPGPU). The AVX extensions give the ability to perform a single operation on a small vector of data simultaneously. This is referred to as single instruction multiple data (SIMD) parallel processing. The AVX512 extension provides operations to manipulate integer, floating point and double precision numbers in (typically cache aligned) vectors of 512 bits (e.g. 16×32 bit float).

B. GPquick: C++ Genetic Programming System 1993–2018

GPQUICK [3] is an implementation of Koza's genetic programming system [7] written in C++. Singleton appears to have used implementation tricks suggested by Keith and Martin [8]. In particular they recommend using a Pre-fix Jump Table (PJT). When interpreting the GP tree, the PJT allows the direct dispatch of the code responsible for implementing each function or leaf without use of switch/case or if/else statements [8, page 300]. In the more than a quarter of a century since 1994, GPQUICK has been maintained (including porting to new hardware, e.g. Dec Alpha, CUDA [9] and of course Intel) but the assumption has always been retained that the jump table is the right way to go. In Sections V-D and V-E we will show that, for big trees, evolution can find more efficient switch/case approaches.

C. Background: Genetic Improvement

Genetic Improvement [10]–[16] has been widely applied, including automatically fixing bugs [17], [18], reducing energy consumption [19] improving predictions [20] and reducing run time [10], [21]. Lopez-Lopez et al. [22] applied our GISMO GI framework [10] to Gagné's Beagle Puppy [23], an open Evolutionary Computing framework which includes GP and emphasises ease with which programmers can adapt it to new EC ideas and mutation operators, rather than raw GP speed. Lopez-Lopez et al. [22] do not report speed, instead they concentrate upon trying to evolving a GP algorithm which gives better symbolic regression models. In the following sections we describe using GISMOE to speed up the world's fastest GP system without changing in anyway its output.

III. Representation and Genetic Operators

For our Genetic Improvement system we used GISMOE [10], [22], [24]. (Other GI systems include GIN [25] and PyGGI [26].) GISMOE is a grammar based GP system in which the source code to be evolved is first automatically converted into a BNF grammar specific to that code. Variable rules within the grammar can then be mutated. The individuals in

the population are variable length lists of grammar mutations. GISMOE allows variable BNF rules to be: deleted, replaced by another rule of the same type, swapped with another rule of the same type, and another rule of the same type to be inserted. (Table III has some examples.) We also include grammar rules for the whole of the Intel AVX library. These are typed by method signature to ensure existing AVX code can be replaced by compatible AVX library intrinsics. The are also special vecsize mutations which convert between AVX-128, 256 and 512 bit instructions [24].

To evaluate an individual each of the grammar mutations it contains is applied (in left right order) to the grammar, giving a new grammar. The new grammar is inverted to give C++ code, which the fitness function will attempt to compile and run. The grammar will ensure that the new source code is syntactically valid C++ code but does not ensure it will compile. Compilation errors are usually due to moving variables out of scope. The fraction of the initial random population which compiles varies between 37% (in Section V-C) to 100% (in Section V-A).

If an individual is selected to be a parent of the next generation: 1) a child is created by appending a random mutation to the parent and 2) a single child can be created by two point cross over between it and another parent [24]. In some runs, the initial population contains few individuals which run ok and yield correct answers, in which case the missing children are created at random (as in the initial population). This injection of random individuals continues until the number of error-free individuals is sufficient to fill the next generation.

A. BNF Grammar Special Cases: Blank Lines, vzeroupper

The GP interpreter is 59 lines of code (excluding header files), split over six opcodes ($+ - \times / x$ and ERC constants). Thus each part is small, leading to a compact BNF, which might be less evolvable. To allow evolution additional flexibility, blank lines were added after each statement. This makes it easy to insert code both before and *after* existing statements.

Whilst trying to understand why AVX code might be slow, various online posts were consulted. One on stackoverflow suggested vzeroupper might resolve known problems with Intel's skylake processor. Since our cluster uses skylake processors, we added a read only rule to the grammars allowing evolution to insert the assembler statement `__asm__ __volatile__( "vzeroupper" : : : );`

IV. Fitness Function

A. Random Trees as Random Test Cases

The mutated code is run on a random test case. Assuming it runs without error and gives the same answer as the released code, its fitness is the difference between its run time and that of the released code. To be eligible to be selected as a parent in the next generation, the mutated code must run without error and produce the correct answer. Even slower mutants can be parents but of course must still compete on fitness (run time) with the rest of the population.

TABLE I
Default Genetic Improvement Parameters.

Representation:	variable list of replacements, deletions, swaps and insertions into BNF grammar comprised of Eval() C++ code specific rules, plus 5694 rules for the Intel Intrinsics library [24].
Fitness:	compile with g++ 9.2.0 -O3 -DNDEBUG -pthread -march=skylake-avx512. Run on random tree changed every generation. log distribution tree size (half lie below 1000 and half above). Fitness is difference in quartiles of elapsed time between original and evolved C++ code (Section IV).
Population:	100 (500 in Section V-C), panmictic, no elite, generational.
Parameters:	Initial population of random single mutants. Best half selected to be parents. 50% two point crossover, 50% mutation. No size limit. Stop after 100 generations.

We typically run overnight, < 12 hours, so to keep GI run time manageable, the test cases used in both pass 1 and 2 (Section IV-D, next page) were kept small. (They are between 10^2 and 10^4 nodes, *geometric* mean 1000, see Table I). Section IV-E, next page, adds a third pass with random trees of about 20 million opcodes. A high resolution (64 bit nanosecond) clock provided by the server CPU is used. To avoid overfitting, both the size and contents of the test case to be evaluated were changed every generation.

B. `for` Loops Not Mutated to Avoid Infinite Loops

The evolved code to be evaluated contains no branches but contains `for` loops. It turns out the consteval and diveval `for` loops are identical and so there is no virtue in mutations which interchange their components. Except for special AVX vec processing [24], this leaves only delete mutation and since, in this case, deletion will only remove the loop or cause it to loop indefinitely, it was decided to prevent non-vec `for` mutations.

C. Reducing Fitness Noise with Robust Average Runtime

Program runtime is notoriously noisy. (Even the unix perf stat count of instructions:u shows some variation.) This is compounded by 1) running on servers within an active cluster, in which both the number and the nature of the other jobs varies continuously. 2) The cluster servers use active power management to independently vary each core's CPU clock frequency (between 1.00–3.00 GHz, nominally the server is 2.3 GHz). Indeed frequency changes can happen while genetic programming is running on the core. Also jobs on the server can be moved between cores.

In view of this noise, the reference code and the evolved code were run multiple times alternately as close together as possible. Since run time suffers from long tails, and we only care for the difference, we took the average of the multiple runs to be the first quartile of all the times. (This effectively discards the longer half of the data and then takes the median of the short half.) For short runs, we take the first quartile of 11 runs, i.e. the third, see Figure 1.

In Section V-E, to save run time, with trees of more than a million nodes, we rely on the longer run time to average out enough of the noise and take fitness as the difference in run time between one run of the reference code and one run of the mutated code.

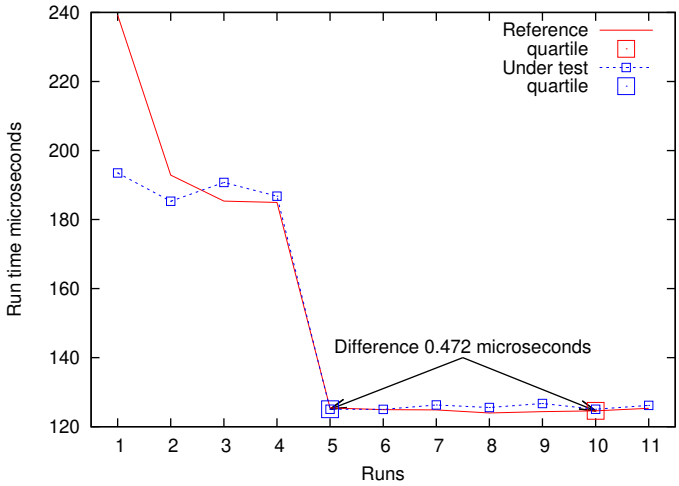

Fig. 1. Example of using first quartile □ of 11 run times to give robust estimate of difference between reference code (solid line) and evolved code (dashed line). In this example, elapse times from run 5 onwards are almost identical (quartile difference 0.472 μS). Note mean as an average is susceptible to outliers, e.g. run 1, and would have given a difference of 3.603 μS.

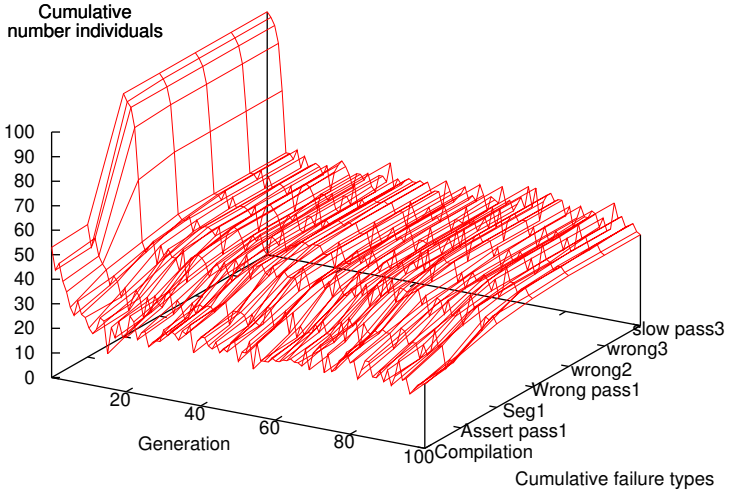

Fig. 2. Example of evolution of errors in 010 run Section V-E (other experiments are similar). For simplicity plot shows cumulative error during fitness evaluation. Since there are very few pass 2 and pass 3 errors, lines after pass1 are almost parallel to the y-axis. There are never compilation failures or segmentation faults in pass 2 or 3. In this run all compilable mutants produced changes in object code (both pass 1 and 2) and there were no incorrect answers in pass 2 or 3. On average 57% of the population (100) evaluated large (20 million) random trees correctly without slow down.

D. Two or Three Passes: Array Index and Addressing Errors

To detect array out of bounds addressing errors, in the first pass run time assert statements were automatically inserted before each array and stack pointer access ("Assert pass1" in Figure 2). To speed up GISMOE run time, in pass 1, the GNU C++ compiler (version 9.2.0) was used without optimisation. The first pass detects all compilation errors and almost all run time errors (Figure 2), however the additional checks mean it cannot be used to reliably estimate run time.

Instead mutants which compile, run ok and yield the correct answer, are recompiled using all the optimisation switches used by the original GPavx distribution kit[1]. This includes -DNDEBUG, which removes all assert statements. The optimised code is then run on the test case for this generation and (assuming it calculates the right answers) its (1st quartile) run time relative to the distributed code is used as the fitness of the mutant (previous Section, IV-C).

E. Pass 3 Fitness Testing on Big (20 million opcode) GP Trees

In the last set of experiments, Section V-E, a third pass was added, which uses far bigger trees (20 million nodes bigger). This was because our goal is running extended evolution with humongously large trees (Section I) and so we want evolution to optimise the interpreter for very big trees. To reduce fitness evaluation time, we do not test slow mutants (i.e. speedup < 1.0 in pass 2) a third time. Runtime is now in the region of a second, rather than microseconds, so we expect that noise to be a small fraction of runtime. Therefore, again to reduce fitness evaluation time, in the third pass we only run the reference code and the mutant code once each, rather than 11 times. Note in the later runs, a single mutant can have multiple fitnesses, e.g. from pass 2 and from pass 3. Since pass 3 fitness values are typically many thousands of times bigger than those from pass 2, mutants which run pass 3, dominate the population.

[1]http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GPavx.tar.gz

```
OPDEF(ConstEval) {
  assert(EvalSP <
&evalstack[nthreads*MAXTESTCASES*evalstacksize]);
  const float val = GETVAL;
  for(int i=0;i<MAXTESTCASES;i+=8) {
    //https://software.intel.com/en-us/node/524140
    //No corresponding Intel AVX instruction.
    _mm256_store_ps(&EvalSP[i], _mm256_set1_ps(val));
  }
  inc_EvalSP;
}
```

Fig. 3. Fragment of original AVX GPquick interpreter (evaluation of constants). Note use of older 256 bit instructions rather than AVX-512.

V. Evolution of Faster GP Tree Interpreters

To gain confidence, before attempting to optimise the whole AVX GP interpreter, we started with the simplest part, evaluation of constants (next section). Since this was successful, in Section V-C we next attempted to improve the code to evaluate protected division which is the largest of the eval functions. Section V-D and Section V-E describe 1+6 approaches to optimising the whole AVX interpreter.

A. Interpreting Constants

Figure 3 shows the original C++ code for interpreting constant opcodes. OPDEF is the GPquick macro for defining part of the interpreter. Similarly GETVAL is a macro for converting the current opcode into the corresponding floating point constant (ERC) value.

Before evaluation the address of each OPDEF function is loaded into the PJT jump table (Section II-B). We have two jump tables. The first for the original reference code and a second for our evolved code.

In most GP systems, including serial versions of GPquick, the GP tree is interpreted once per test case, i.e. repeatedly. If the GP tree does not fit into L1 cache, this may be expensive.

However GPavx uses a single pass through the tree to allow the interpreter to take advantage of the SIMD parallel operations. The interpreter AVX code performs each operation in the GP tree once on all the test cases before moving onto the next node of the tree. Thus traversing the tree only once.

The original AVX code uses an explicit stack. To support parallel operation, each consteval pushes a vector MAXTESTCASES=48 floats onto the stack, whilst functions (internal nodes) pop two such vectors. (GPavx supports only functions with two arguments.) They then use SIMD operations to do the required calculation and then push MAXTESTCASES results back onto the explicit stack.

The original code (Figure 3) uses various AVX instructions. As before [24] the BNF grammar is automatically generalised by replacing specific AVX instructions with `vec` variants. Evolution can tune these to 128, 256 or 512 bit variants. Notice the original code used 256 bit variants, but evolution is able to replace these with more recent 512 bit versions.

Each generation the fitness function generates a series of random ConstEval opcodes (each corresponding to a single node tree) and runs both the original and each mutated ConstEval interpreter on them. Each should push the corresponding constant onto the stack. The fitness function checks that the stack contains the right contents (including its depth) and records the difference in run time (1st quartile, Section IV-C). Notice that the stack is the same depth as the number of random operations (between 100 and 10 000) but contains 48 floating point values at each level.

As with Sections V-C to V-E, evolution is run with the default parameters given in Table I. For ConstEval the population is 100 and there are 15 grammar rules specific to ConstEval. After 100 generations, the best in population reports a speed up of 21%. Figure 4 reports actual speed for 100 runs for various numbers of random opcode. This was sufficiently encouraging to continue to the largest of the AVX interpreter's functions, protected division, Section V-C.

B. Estimating Intel L1 Cache Performance

ConstEval is the simplest of our test cases and therefore liable to get the most out of the L1 cache. When loading 151 random test cases onto the stack, the best mutant gets on average about 8.01e+09 GP operation per second, Figure 4. Because of the AVX operations, we are actually doing 48 GP operations together. It is to be hoped that the compiler will be able to keep all the highly access indexes and loop counters in registers, and thus only the stack, opcode, PTJ, ERC and evalstack memory will have to be read/written via the L1 cache per 48 GP unit. I.e. 1 cache line each, except for evalstack, where 48 floats occupy 3 cache lines. We estimate this at 7 = (1+1+1+1+3) cache lines, suggesting the maximum L1 cache data rate is $7 \times 64\,\text{bytes} \times 8.01\,10^9\,\text{GPOPs}^{-1}/48 = 70$ Gigabytes per second (i.e. 1200 million cache lines per second).

Given an estimate of peak cache performance we can place an upper bound on the performance of the best GP AVX interpreter. On an average GP operation, the 010 mutant (Section V-E) perhaps has to read the stack and an

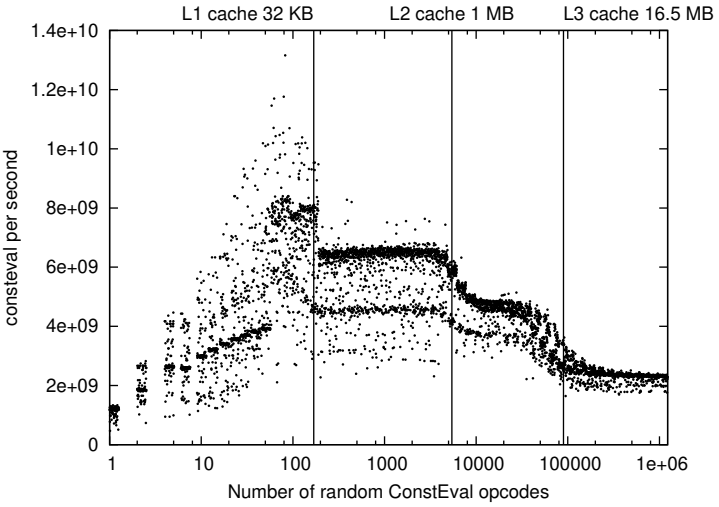

Fig. 4. Performance of simplified evolved GPquick interpreter for ERCs. Although trained on 100 . . . 10 000, the evolved code extrapolates to a million random one node trees. Vertical lines show impact of data caches. Notice variation across 100 runs of each test case. (Small horizontal displacement added to make overlapping data more visible.)

opcode. Then 50% of the time (a function eval) read the evalstack twice and write it once, 25% of the time (XEval) transfer 48 floats onto the evalstack, and 25% of the time (ConstEval) read an ERC and write 48 floats onto the evalstack. Expressing this in terms of cache lines, gives: $1 + 1{+}0.5{\times}3{\times}(2{+}1){+}0.25{\times}3{\times}2{+}0.25{\times}(1{+}3) = 9$ cache access per opcode. If our estimate of L1 cache bandwidth is accurate then this gives an upper limit of 48×1200 million/9 $= 6$ billion GP Operations per second (6 Giga GPOP/s) per Xeon 5118 core. For 48 cores this gives a total of 300 Giga GPOP/s. Worryingly the best results [5, Tab. 3] is already within a factor of two of this, suggesting major improvement is not possible by code improvement alone but more radical approaches are needed. Such as, since many GP operations have no side effects, avoiding re-evaluating parts of the trees which have not been modified, Section VIII-C.

C. Interpreting Protected Division

Protected division is more complicated that the three other binary functions which GPavx supports, as it must detect in parallel division by zero and generate 1.0f as the default value. Building on ConstEval (Section V-A) and since the AVX code contains no branches, the fitness function could be quite simple. It generated a long random list of three node random trees. Each containing protected division of two random constants. As with ConstEval each tree should leave its answer on the stack to be checked by the fitness function. As before the same fitness test case is used for the whole of each generation but replaced in the next generation. The number of random 3-trees is similarly chosen logarithmically from 100/3 to 10 000/3.

At 67 rules, the grammar is much bigger than consteval.bnf (note it also uses the newly evolved ConstEval_gp, Section V-A). Therefore the population size was increased from 100 to 500. Figure 5 shows the best speed up in generation 100

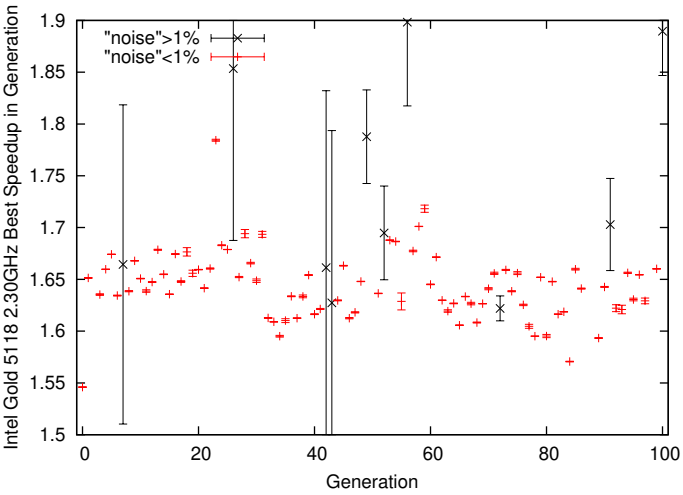

Fig. 5. Speed up of best of each generation (population 500). Combined diveval and consteval, Section V-C. Five noisy points above 1.8 omitted. Notice rapid initial rise in fitness but (perhaps due to the impact of noise) later variability. (We have 11 timings for the non-mutated reference code, Figure 1. Noise is estimated by $(4^{\text{th}} - 2^{\text{nd}})/3^{\text{rd}}$ fastest reference times.)

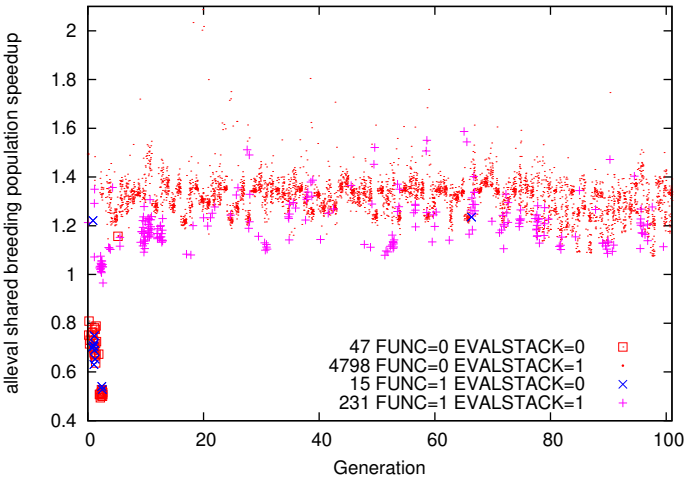

Fig. 6. Speed up of best 51 of each generation (population 100) alleval, Section V-D. Ten noisy points above 2.1 omitted. Notice despite testing all 4 possibilities every generation, the breeding population is quickly dominated by using switch/case statements and an explicit stack (4798 dots) in preference to the original PJT jump table or evaluating the GP tree three times.

was 89%. However Figure 5 suggests this was out of the usual run of the later populations, nevertheless we were encouraged to consider expanding to allow evolution to consider afresh the whole of the AVX GP tree interpreter.

D. Whole AVX GP Interpreter (Shared Population)

In the last experiments (this and the next section) we automatically generate BNF grammars covering the whole of the AVX GPquick interpreter. Each generation the mutants are tested on one complete randomly generated tree of between 100 and 10 000 nodes (as above). We return to a population of 100.

As mentioned in the previous section, a large fraction of the interpreter is devoted to protected division where evolution had already given some encouraging results. However we feared the remaining functions (XEval, AddEval, SubEval and MulEval) would give little additional room for evolution. Therefore we decided to allow it to revisit some previous manual design choices. Additional conditionally compiled code was added by hand to allow replacing the PJT jump table by a switch statement (FUNC=0) and to replace the explicit stack by using conventional return statements (EVALSTACK=0). EVALSTACK=0 makes the evaluation return type a 16 float vector, so the GP tree must be processed MAXTESTCASES/16 (i.e. 3) times. Conditional compilation of these two options give four possibilities. Here the BNF grammar (129 rules) contains conditional compilation, which cannot be mutated, but the code covered by it can be evolved.

We treat each mutant as if it existed in four independent environments: FUNC: with switch/case or PJT jump table, EVALSTACK: with implicit stack (3 passes) or with explicit stack. That is, each mutant is (conditionally) compiled four times for pass one (i.e. with asserts and no optimisation) run and its output is compared with that of the original code. As before, if it gives the correct answer, it is complied (with optimisation) and run again. Notice each mutant is compiled between 4 and 8 times and can have up to 4 fitness values. Figure 6 shows speed up of more than two fold were reported (albeit 1.5 fold is more typical).

In several cases we saw evolution mutating the order of case statements to place more frequently used opcodes nearer the switch statement. Potentially this reduces opcode dispatch overhead. Since the switch statement is all code and thus visible to the compiler (whereas the PJT jump table is less visible) possibly it is more open to compiler optimisation. We certainly see heavy use of compiler optimisation tricks such as inlining function calls and unrolling loops.

Figure 6 shows evolution quickly favoured one of the four environments. Indeed it appeared that fast mutants were not only successful but also "poisoning" their competitors by injecting compilation errors into conditionally compiled code they did not use. Thus in many cases children of successful mutants would fail to compile in at least two of the four environments. Therefore in the last experiments (next section) we used separate populations rather than allowing competing mutants to co-evolve.

E. Evolving whole AVX GP interpreter (6 Separate Grammars)

In the last experiment we added another design choice to evolution. In the previous section, deciding not to use the external stack required the tree to be processed 3 times therefore we manually added another conditionally compiled option, T48, which made each eval function in the AVX interpreter return all 48 (MAXTESTCASES) in one go. The interpreter's EVAL function is called recursively at the top of the GP tree and returns answers for all 48 test cases in one pass of the tree. Effectively using the system stack in place of the explicit EVALSTACK. Although T48 can be true or false, it is only valid when EVALSTACK=0 thus there are now 6 (rather than 8) environments to test mutants.

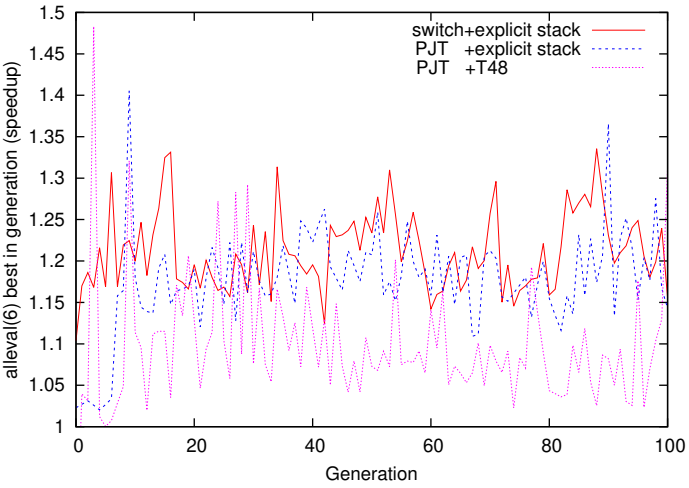

Fig. 7. Evolution of best in population speed up alleval(6) Section V-E. For clarity, 000, 001 and 100 not plotted as on average they give no speedup.

As noted in the previous section, attempting to co-evolve mutants in multiple environments seems to cause unnecessary competition. Therefore we decide to use separate runs and so separate populations for each environment. We automatically generated 6 BNF grammars, one for each valid setting of the three conditional compilation symbols, FUNC, EVALSTACK and T48, and ran each with a population of 100 for 100 generations. (The six grammars contain 000 49, 001 57, 010 86, 100 49, 101 57 and 110 86 BNF rules.)

We seem to be getting good speed up on relatively small trees ($10^2 - 10^4$ nodes) but want also to ensure the evolved AVX interpreter scales to humongous trees, so we added a third pass to fitness testing (see Section IV-E above). Good mutants are now also tested on a random tree which is 20 000 000 nodes longer than the one used in the first two passes. As in the previous section, mutants can have multiple fitness values, however good fitness on the 2 10^7 node GP trees quickly dominates selection, Figure 7.

Using EVALSTACK by itself (010, solid red in Figure 7) gave consistently the best results on the 20 million node GP trees, followed by the original environment (110, dashed blue) and then the jump table and implicit stack (101 FUNC=1 T48=1, dotted purple).

VI. Using Assembler Code to Remove Bloat

The best of the last generation mutants evolved in the last experiment covering the whole AVX GP interpreter (Figure 7) are somewhat bloated (column 3 of Table II). Previously, e.g. [10], we used fitness to thin the evolved mutants by removing genes which had no impact on fitness. As our fitness (runtime) is so noisy we sought something else and choose comparing the compiler's assembler output [27] to indicate which parts of the evolved mutants actually make a difference.

For each of the three promising cases (010, 110, 101) the best individual from generation 100 was used. g++ with -S and the usual command line options was used to compile again the evolved code, but -S causes the C++ compiler to generate an assembler .s text file. The mutant's .s file was saved. The

TABLE II
Using assembler code as a guide allowed noiseless removal of ineffective code from best of generation 100 GI individuals

Environment		Number of genes evolved	final
010	switch and explicit stack	16	6
110	PJT jump table and explicit stack	20	3
101	PJT jump table and eval in one pass (T48)	23	2

evolved mutant contains len genes, there are len mutants with one gene removed from it. These len mutants were created and compiled as far as generating their .s file. The len shorter mutants' .s files were each compared with the evolved mutant's .s file. If they were identical, the corresponding gene was removed from the evolved mutant. In some cases knocking out a gene caused the shorter mutant not to compile, so the gene was retained. However when all len short mutants had been tried and a new shorter mutant created, a second pass of gene removal was used. This resulted in no compilation errors and an even shorter mutant. Table II gives the number of genes before and after assembler code based clean up.

VII. Out of Sample Testing

The performance of the three cleaned up codes relative to the distributed original GPavx code for a wide range of random GP tree sizes is given in Figure 8. Table III gives performance for the three mutants during training (pass 2 and pass 3) as well as two unrelated random trees (sizes 995 and 20 056 365). Table III is broken into three section (one for each tidied mutant). After each row of performance data, comes the mutant itself. The mutant's genes are separated by white space. Each gene defines one BNF grammar change. This is followed by a text description of how the whole mutant works.

The estimate of noise given under "p2" is based on variation across the 11 runs used to measure fitness. Typically this takes a handful of microseconds during which neither the load on the server nor its clock speed changes. This allows us to get a good estimate of the speed of the mutant code being tested and that of the original code. However over a period of even a few minutes these two factors do change leading to larger estimates of noise given in columns "995 speedup" and "20 056 365 speedup". Also note that the p2 and p3 figures relate to the best in generation and so part of their high score could also be attributed to randomness due to noise and so we could interpret them as the luckiest members of their generation. Since Figure 8 (and "995" and "20 056 365" columns in Table III) show 100 widely spaced runs for each tree size they give a more representative measure of out of sample performance.

VIII. Discussion: Alternative Fast GPs

As well as specialised hardware accelerators such as graphics cards [9] and parallel computing [28], there are several software based approaches for fast GP. For example, avoiding trees by using linear GP [29] or Cartesian GP [30].

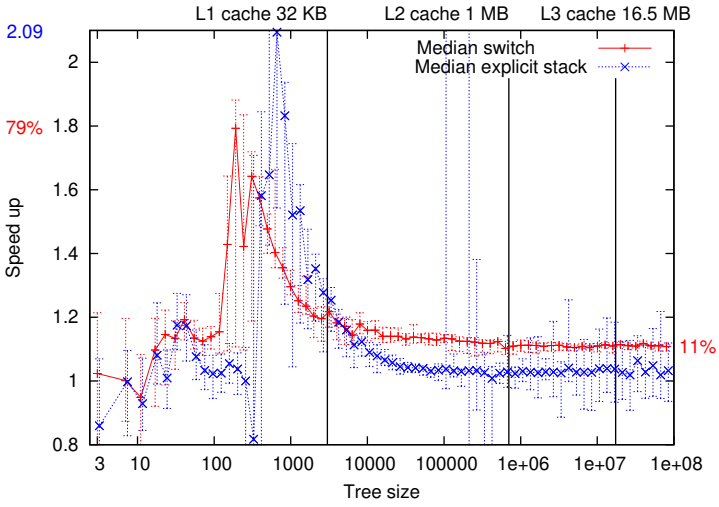

Fig. 8. Speed up (red +) of GI optimised GPquick interpreter using evolved switch statement on random binary trees on multi-user Intel Xeon Gold 5118 2.30GHz server (max 79%). GI optimised user stack (blue × max 2.1 fold). Cluster power management changes CPU clock frequency continuously. Error bars give interquartile range. Vertical bars show impact of data caches.

TABLE III
3 EVOLVED MUTANTS AFTER BLOAT REMOVAL (SECTIONS VI VII).

Environment		Speedup			
	size	p2	p3	995	20 056 365
010	6	40%±0.2%	16%	30%±4%	11%±0.7%
vecsize=16 <CASE_34>x<CASE_32> <CASE_34>x<CASE_33> <_139>x<_137> <CASE_35>x<CASE_34> <CASE_35>x<CASE_31>					
vecsize=16 ensures AVX512 SIMD instructions are used through out. <CASE_ mutants reorder case: in eval dispatch switch moving XEval closer to switch statement and placing MUL and DIV before ADD and SUB. Swapping lines 139 and 137 moves the assignment of e0 to later in Eval2. e0 is not used. Perhaps moving it makes it easier for the compiler to remove.					
110	3	23%±1%	14%	52%±35%	3%±9%
vecsize=16 <_138>+<_266> <_135>x<_139>					
vecsize=16 ensures AVX512 SIMD instructions are used through out. Insert assembler vzeroupper (line 266, Section III-A) between the two EVAL in Eval2(), line 138. Swapping lines 139 and 135 moves the assignment of e0 to later in Eval2. e0 is not used. Perhaps moving it makes it easier for the compiler to remove.					
101	2	31%±2%	31%	-23%±12%	-2%±2%
<_168> <_45><_261>					
Deleting line 168 removes redundant code to set variable to zero. Insert assembler vzeroupper (line 261) at the start of function ConstEval_t48					

A. Compiling to Machine Code instead of Interpreting

Fukunaga et al. [31] advocates a pre-pass through each GP tree which generates machine code directly from it. For each fitness test instead of interpreting the tree, the corresponding machine code is executed. They say the one off cost of generating the machine code quickly pays for itself. They compile for a RISC architecture (Sun UltraSparc 2) rather than AVX extensions for a Intel's x86, so generation of good machine code might be easier. Also, although they stress the speed of their compiler, they say the "maximum speedup ... when the number of test cases is 1000". Whereas (due to use of parallel instructions) we use only three ($3 \times 16 = 48$) test cases.

Non-avx GPquick's FASTEVAL conditional compilation is somewhat similar. With FASTEVAL there is a pre-pass in which tree opcodes are converted to function addresses (rather than machine code). This means the linear array of opcodes is replaced by a linear array of function addresses, thus eliminating the indirection step inherent in using the Prefix Jump Table (Section II-B). For AVX, as well as requiring multiple passes through the tree it would also mean (for 64 bit addressing) eight times as many bytes passing though the caches which suggests it will only help with many fitness tests and trees smaller than $\frac{1}{8}$ cache size. In principle the address array could be used throughout GPquick, i.e. by mutation and crossover operations, thus also eliminating the need for a pre-pass before interpreting each GP tree. However it would mean the memory required to store the population would be eight times as much and the current FASTEVAL code does not support this.

B. Evolving Machine Code

A very successful extension to linear GP is to replace interpreting the (linear) genome by machine code. That is, to define mutation and crossover operators which act on machine code. Nordin's original work was similarly based on the RISC architecture (Sun-4 workstation) but Nordin later showed that evolution of Intel x86 machine code is possible [32] indeed the commercial system Discipulus runs on Intel x86.

C. Interpreting Only Part of the Tree

We can avoid interpreting the whole tree and evaluate only the parts changed by evolution. Handley [33] suggested a very compact encoding in which the whole evolving population is stored as a single directed acyclic graph (DAG). It is often the case that GP primitives do not have side effects and use a static fitness function so trees within the population need only be re-evaluated if they are changed [7]. Indeed if intermediate subtree fitness values are cached within the DAG, only the part of the tree between the change and the root node needs to be re-evaluated. In typical GP usage this reduces the evaluation from O(n) to O($\sqrt{n}$), where n is the average size of the trees. Sutherland [34] implemented this in Java.

Tree interpreters for side-effect free problems, can recognise subtree results which mean a function argument can be ignored [7]. E.g. in `(MUL 0 subtree2)`, `subtree2` can be skipped. However, although large trees are often composed almost entirely of introns (dead code) [35], initial unpublished results suggest savings can be small.

IX. CONCLUSIONS

Even in a noisy multi user cluster environment, on a server which uses power management to dynamically throttle the CPU clock, genetic improvement (GI) found changes to the code of the fastest (single computer) genetic programming (GP) system which make the main GP interpreter engine up to twice as fast, see Figure 8. The switch/case based variant, it is up to 79% faster. On smaller trees (up to 1000 nodes) an average speed up of between 10% and 50% can be expected (red+ line in Figure 8). The GI code consistently gives an 11% improvement on very big trees. We showed g++ -S assembler output can winow chaff genes to noiselessly remove bloat.

It is established wisdom [8] that a jump table is the efficient way to code a C++ GP interpreter. This assumption was built into GPquick from the start and although many different versions of GPquick have been released this is the first work to challenge it. We show, with a modern C++ compiler, a switch statement decoding GP byte opcodes can yield a more efficient GP interpreter, when the PJT table is displaced from L1 cache.

It was thought, given the Intel x86 is a stack based machine, that the compiler would be happier with using the hardware supported implicit stack. However the most efficient mutant did not dispense with the explicit stack. This may be because the AVX 512 bit parallel SIMD vector instructions typically have to be aligned to 64 byte cache boundaries. Therefore it may be this alignment does fit well with other uses of the system stack. (Typically calling functions and passing arguments to them uses much less than 64 bytes).

It appears that the AVX interpreter is now limited by L1 cache bandwidth, rather than code execution time and further improvements will have to reduce data read/write rates.

Acknowledgements

I am grateful for the assistance of Dr. Aymeric Blot and my anonymous reviewers. EPSRC grant EP/P005888/1.

REFERENCES

[1] W. B. Langdon, "Parallel GPQUICK," in *GECCO 2019 Companion*, pp. 63–64. http://dx.doi.org/10.1145/3319619.3326770

[2] N. Alshahwan *et al.*, "Deploying search based software engineering with Sapienz at Facebook," in *SSBSE 2018*, pp. 3–45. http://dx.doi.org/10.1007/978-3-319-99241-9_1

[3] A. Singleton, "Genetic programming with C++," *BYTE*, pp. 171–176, Feb. 1994. http://www.assembla.com/wiki/show/andysgp/GPQuick_Article

[4] R. E. Lenski *et al.*, "Sustained fitness gains and variability in fitness trajectories in the long-term evolution experiment with Escherichia coli," *Proceedings of the Royal Society B*, vol. 282, no. 1821, 2015. http://dx.doi.org/10.1098/rspb.2015.2292

[5] W. B. Langdon and W. Banzhaf, "Continuous long-term evolution of genetic programming," in *ALIFE 2019*. MIT Press, pp. 388–395. http://dx.doi.org/10.1162/isal_a_00191

[6] W. B. Langdon, "Large scale bioinformatics data mining with parallel genetic programming on graphics processing units," in *Massively Parallel Evolutionary Computation on GPGPUs*, S. Tsutsui and P. Collet, Eds., 2013, pp. 311–347. http://dx.doi.org/10.1007/978-3-642-37959-8_15

[7] J. R. Koza, *Genetic Programming*. MIT Press, 1992. http://mitpress.mit.edu/books/genetic-programming

[8] M. J. Keith and M. C. Martin, "Genetic programming in C++: Implementation issues," in *AiGP*. MIT Press, 1994, pp. 285–310. http://cognet.mit.edu/sites/default/files/books/9780262277181/pdfs/9780262277181_chap13.pdf

[9] W. B. Langdon and W. Banzhaf, "A SIMD interpreter for genetic programming on GPU graphics cards," in *EuroGP 2008*, pp. 73–85. http://dx.doi.org/10.1007/978-3-540-78671-9_7

[10] W. B. Langdon and M. Harman, "Optimising existing software with genetic programming," *IEEE TEVC*, vol. 19, no. 1, pp. 118–135, 2015. http://dx.doi.org/10.1109/TEVC.2013.2281544

[11] M. Harman and J. Petke, "GI4GI: Improving genetic improvement fitness functions," in *GI 2015 Workshop*, pp. 793–794. http://dx.doi.org/10.1145/2739482.2768415

[12] J. Petke *et al.*, "Using genetic improvement and code transplants to specialise a C++ program to a problem class," in *EuroGP 2014*, pp. 137–149. http://dx.doi.org/10.1007/978-3-662-44303-3_12

[13] J. Petke, "Constraints: The future of combinatorial interaction testing," in *SBST 2015*, pp. 17–18. http://dx.doi.org/doi:10.1109/SBST.2015.11

[14] J. Petke *et al.*, "Genetic improvement of software: a comprehensive survey," *IEEE TEVC*, vol. 22, no. 3, pp. 415–432, 2018. http://dx.doi.org/doi:10.1109/TEVC.2017.2693219

[15] ——, "Specialising software for different downstream applications using genetic improvement and code transplantation," *IEEE TSE*, vol. 44, no. 6, pp. 574–594, 2018. http://dx.doi.org/10.1109/TSE.2017.2702606

[16] W. B. Langdon *et al.*, "Genetic improvement of GPU software," *GP & EM*, vol. 18, no. 1, pp. 5–44, 2017. http://dx.doi.org/10.1007/s10710-016-9273-9

[17] C. Le Goues *et al.*, "Automated program repair," *Comm. of the ACM*, vol. 62, no. 12, pp. 56–65, 2019. http://dx.doi.org/10.1145/3318162

[18] N. Alshahwan, "Industrial experience of genetic improvement in Facebook," in *GI-2019, ICSE workshops proceedings*, p. 1, keynote. http://dx.doi.org/10.1109/GI.2019.00010

[19] E. Schulte *et al.*, "Post-compiler software optimization for reducing energy," in *ASPLOS 2014*, pp. 639–652. http://dx.doi.org/10.1145/2541940.2541980

[20] W. B. Langdon *et al.*, "Evolving better RNAfold structure prediction," in *EuroGP 2018*, pp. 220–236. http://dx.doi.org/10.1007/978-3-319-77553-1_14

[21] Z. A. Kocsis *et al.*, "Automatic improvement of Apache Spark queries using semantics-preserving program reduction," in *GI 2016 Workshop*, pp. 1141–1146. http://dx.doi.org/10.1145/2908961.2931692

[22] V. R. Lopez-Lopez *et al.*, "Applying genetic improvement to a genetic programming library in C++," *Soft Computing*, vol. 23, no. 22, pp. 11 593–11 609, 2019. http://dx.doi.org/10.1007/s00500-018-03705-6

[23] C. Gagné and M. Parizeau, "Genericity in evolutionary computation software tools: Principles and case study," *IJAIT*, vol. 15, no. 2, pp. 173–194, 2006. http://dx.doi.org/10.1142/S021821300600262X

[24] W. B. Langdon and R. Lorenz, "Evolving AVX512 parallel C code using GP," in *EuroGP 2019*, pp. 245–261. http://dx.doi.org/10.1007/978-3-030-16670-0_16

[25] A. E. I. Brownlee *et al.*, "Gin: genetic improvement research made easy," in *GECCO '19*, M. Lopez-Ibanez *et al.*, Eds., Prague, Czech Republic, 2019, pp. 985–993. http://dx.doi.org/10.1145/3321707.3321841

[26] Gabin An *et al.*, "PyGGI 2.0: Language independent genetic improvement framework," in *ESEC/FSE 2019*, pp. 1100–1104. http://dx.doi.org/10.1145/3338906.3341184

[27] E. Schulte *et al.*, "Software mutational robustness," *GP & EM*, vol. 15, no. 3, pp. 281–312, 2014. http://dx.doi.org/10.1007/s10710-013-9195-8

[28] D. Andre and J. R. Koza, "Parallel genetic programming: A scalable implementation using the transputer network architecture," in *AiGP 2*. MIT Press, 1996, pp. 317–337. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6277532

[29] W. Banzhaf, "Genetic programming for pedestrians," in *ICGA 1993*. Morgan Kaufmann, p. 628. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/GenProg_forPed.ps.Z

[30] J. F. Miller, "Cartesian genetic programming: its status and future," *GP & EM*. http://dx.doi.org/10.1007/s10710-019-09360-6

[31] A. Fukunaga *et al.*, "A genome compiler for high performance genetic programming," in *GP 1998*. Morgan Kaufmann, pp. 86–94. http://metahack.org/gp98-compiler.pdf

[32] P. Nordin *et al.*, "Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover," in *AiGP 3*. MIT Press, 1999, pp. 275–299. http://www.cs.ucl.ac.uk/staff/W.Langdon/aigp3/ch12.pdf

[33] S. Handley, "On the use of a directed acyclic graph to represent a population of computer programs," in *WCCI 1994*, pp. 154–159. http://dx.doi.org/10.1109/ICEC.1994.350024

[34] N. F. McPhee *et al.*, "Sutherland: An extensible object-oriented software framework for evolutionary computation," in *GP 1998*. Morgan Kaufmann, p. 241. http://facultypages.morris.umn.edu/~mcphee/Research/Sutherland/sutherland_gp98_announcement.ps.gz

[35] W. B. Langdon *et al.*, "The evolution of size and shape," in *AiGP 3*. MIT Press, 1999, pp. 163–190. http://www.cs.ucl.ac.uk/staff/W.Langdon/aigp3/ch08.pdf

[36] Yue Jia *et al.*, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *ICSE 2015*. IEEE, pp. 540–550. http://dx.doi.org/10.1109/ICSE.2015.71

[37] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the push programming language," *GP & EM*, vol. 3, no. 1, pp. 7–40, 2002. http://dx.doi.org/10.1023/A:1014538503543