Improving SSE Parallel Code with Grow and Graft Genetic Programming

William B. Langdon and Ronny Lorenz
Department of Computer Science, University College London, Gower Street, WC1E 6BT, UK.
Theoretical Biochemistry, Theoretical Chemistry, University of Vienna

ABSTRACT

RNAfold predicts the secondary structure of RNA molecules from their base sequence. We apply a mixture of manual and automated genetic improvements to its C source. GI gives a 1.6% improvement to parallel SSE4.1 code. The automatic programming evolutionary system has access to Intel library code and previous revisions. On 4 666 curated structures from RNA_STRAND, GGGP gives a combined speed up of 31.9%, with no loss of accuracy (GI code run 1.4 10^{11} times).

ACM Reference format:
William B. Langdon and Ronny Lorenz. 2017. Improving SSE Parallel Code with Grow and Graft Genetic Programming. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 2 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082524

1 INTRODUCTION

In contrast to proteins, RNA structures are mostly investigated at the level of their secondary structure, i.e. their base pairings. The most widely used computer programs that predict secondary structures from sequence data are probably those within the ViennaRNA Package [1], principally RNAfold. Unlike our earlier work with pknotsRG [2] there is no pre-existing parallel version of RNAfold.

Excluding documentation, help files, examples and test cases, version 2.3.0 of ViennaRNA package consists of about 400 C/C++ source files containing about 170 000 lines of code. RNAfold itself, excluding include files, is made from six C sources files (7 100 lines of code).

We use genetic improvement [3] as part of our Grow and Graft GP (GGGP) [4] approach to optimising program sources. Almost all the performance gain was found in the (≈ 3 day) manual pre-evolution phase. We profiled RNAfold using GNU gcov on a large real RNA molecule (CRW_01456 3 435 characters). The whole of RNA_STRAND v2.0 [5] was downloaded from http://www.rnasoft.ca/strand/download/RNA_STRAND_data.tar.gz.) gcov indicated almost all the execution time was taken by a small fraction of function E_ml_stems_fast in multibranch_loops.c where 4 lines inside nested for loops (Figure 2) were executed billions of times.

GECCO '17 Companion, Berlin, Germany
 978-1-4503-4939-0/17/07...$15.00
DOI: http://dx.doi.org/10.1145/3067695.3082524

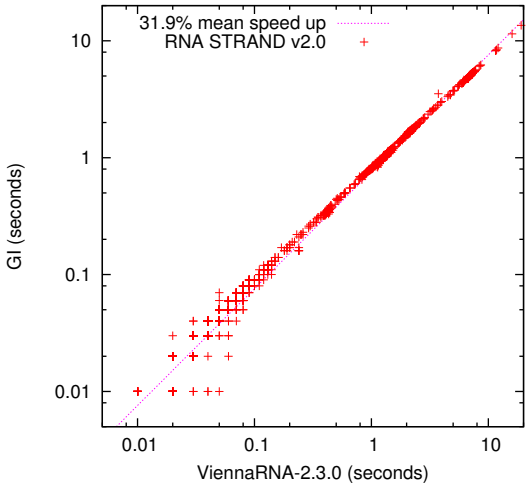

Figure 1: Elapsed time of GIed RNAfold v. original. Data below 0.01 seconds not plotted. Note log scales. The GI code always produced identical answers.

The original intention was to use GGGP to generate a CUDA version of RNAfold but having determined that the main computational bottle neck was quite small but inside a heavily nested routine which might itself be called more than four million times, it was decided to attempt something new and use GGGP with Intel SIMD vector SSE operations.

1.1 Parallel SSE vector instructions

Towards the end of Moore's law as commonly applied to CPU clock speeds, Intel started to increase the parallelism of its flagship 8080 series of processors. The most famous and successful approach has been to put multiple CPU cores onto single silicon chips. However at about the same time, Intel extended the instruction set to support single instruction multiple data (SIMD) vector operations. Many of these allow four or more 32 bit operations to be performed in parallel. The SSE instruction set has been progressively extended and now accelerators, such as the Intel Xeon Phi, support vectors of 512 bits.

It had been hoped to use GI to automatically extend the manually written 128 bit SIMD version of the heavily used code to 512 bit vectors. Although a grammar [6] was successfully created from the Intel SSE documentation (rather late in the day) it was realised we had no access to a Xeon Phi and none of the available computers supported 256 or 512 bit vector operations. Hence GI was only applied to the new 128 bit code (Figure 3).

Table 1: GGGP to improve SSE version of RNAfold

Representation:	variable list of replacements, deletions and insertions into BNF grammar (1030 rules)
Fitness:	Compile (GCC 5.4.0) to modified object code, run on 10 000 random test cases.
Population:	50, panmictic, non-elitist, generational.
Parameters:	Initial population of random single mutants. 50% truncation selection. 50% two point crossover, 50% mutation.

```
for (decomp=INF, k=i + 1 + turn; k<=stop; k++, k1j++){
  if((fmi[k] != INF ) && (fm[k1j] != INF)){
    en = fmi[k] + fm[k1j];
    decomp = MIN2(decomp, en);
  }
}
```

Figure 2: Original code, in E_ml_stems_fast()

During the manual phase, intermediate versions of the new manually written code were held in a revision control system (RCS). The whole of the 128 bit SSE library and the earlier manual revisions were available, via the automatically created grammar, as an extended code base [7]. 789 rules were derived from the SSE documentation. A further 168 were derived from RCS. Finally 141 came from the manually written SSE code (i.e. the usual GI source seed code [6], Figure 3). Of these 68 are fixed and provide the framework within which GI operates on the remaining 73. Notice, the manual SSE code (Figure 3) does not check for INF*inity*.

The best individual evolved over night (generation 147) contains five mutations to the hand written code. Two of them involved importing from the RCS code base. (None used the Intel documentation code base.) As is usual in GI practise [6], only the critical changes were retained (shown in blue in Figure 3). In fact, the change imported from RCS, which changes the first `for` loop's upper bound, is all that is needed. It means the first loop may be execute once more (during which it will process 4 array indexes in parallel). This allows the compiler -O2 optimisation to remove entirely the second loop containing non-parallel instructions. The GI code never accesses data outside the bounds of the arrays. In real usage, it gives exactly the same answers.

2 EXPERIMENT

In the GP run (see Table 1) each mutant is compiled and run on 10 000 of the available 4 232 596 test cases. Each of its answers is compared with that of the original code and how long it took is recorded. To reduce noise, this is repeated 103 times and the 1^{st} quartile time is taken. To give a spread of easy to hard tests, each generation the 4 232 596 test cases are divided equally into 5 and a sequence of 2 000 is chosen at random from each fifth. Every mutant in that generation is tested on the same $5 \times 2\,000$ examples.

To avoid infinite loops, a CPU limit of 15 seconds was imposed (on our 3.20GHz i5 CPU). Each generation the mutants which were run and terminated ok are sorted by the number of tests they passed and their runtime (minimised). The top half the population (25) are selected to have 2 children each in the next generation.

```
#include <smmintrin.h>
int horizontal_min_Vec4i(__m128i x) {
 __m128i min1=_mm_shuffle_epi32(x,SH..(0,0,3,2));
 __m128i min2=_mm_min_epi32(x,min1);
 __m128i min3=_mm_shuffle_epi32(min2,SH..(0,0,0,1))
 __m128i min4=_mm_min_epi32(min2,min3);
 return _mm_cvtsi128_si32(min4);
}
int modular_decomposition(int i, int ij, int j,
                          int turn, int* fmi, int* fm) {
 int k   =  i + turn + 1;
 int k1j = ij + turn + 2;
 const int stop = j - 2 - turn;
 int decomp = INF;
{const int end = 1 + stop - k;
 int i;
 for(i=0;i<end;i+=4){ //was for(i=0;i<end-3;i+=4) {
  //if((a[i] != INF ) && (b[i] != INF)){
  __m128i a =_mm_loadu_si128((__m128i*)&fmi[k +i]);
  __m128i b =_mm_loadu_si128((__m128i*)&fm[k1j+i]);
  __m128i c =_mm_add_epi32(a,b);
  const int en = horizontal_min_Vec4i(c);
  decomp = MIN2(decomp, en);
 }
 for(;i<end;i++) {
  const int en = fmi[k +i]+fm[k1j+i];
  decomp = MIN2(decomp, en);
 }
 } return decomp;
}
```

Figure 3: Hand coded GGGP replacement code. Evolution (blue) gives 1.6% speedup.

During evolution 10% of mutants fail to compile, 1% compile ok but their object code is identical to the seed code's, 3% fail at runtime (e.g. segfault or CPU time limit exceeded) and 86% run all ten thousand tests.

3 CONCLUSIONS

We have demonstrated that evolution can optimise C code primarily composed of hand written SSE instructions. Our original plan to allow GGGP to expand it from 128 bit to 256 or 512 bit instructions was frustrated by the available hardware. Nonetheless GI found a small unexpected optimisation on top of handwritten code, using a standard desktop PC under a standard operating system (Ubuntu 16.04.1 LTS) without specialised customisation to either.

Acknowledgements

I am grateful for the assistance of Bobby R. Bruce and stackoverflow's Paul R. GP code in `rnafoldGI.tar.gz`

References

[1] Lorenz, R., et al.: ViennaRNA package 2.0. Alg Mol Biol **6**(1)
[2] Langdon, W.B., Harman, M.: Grow and graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. GI-2015
[3] Petke, J., et al., Genetic improvement of software: a survey. TEVC
[4] Langdon, W.B. et al., Improving CUDA DNA analysis software with genetic programming. In GECCO '15, Madrid, 1063–1070
[5] Andronescu, M., et al.: RNA STRAND. BMC Bioinf **9**(1) 340
[6] Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. TEVC **19**(1) (2015) 118–135
[7] Petke, J., et al., Specialising software for different downstream applications using GI and code transplantation. TSE (accepted).