EuroGP-2019. L. Sekanina, Ting Hu, Nuno Lourenco *Eds.*, LNCS 11451, 245–261, Leipzig. 24-26 April 2019. Springer. doi:10.1007/978-3-030-16670-0_16 *Preprint*

Evolving AVX512 Parallel C Code using GP

William B. Langdon and Ronny Lorenz

CREST, Computer Science, UCL, London, WC1E 6BT, UK
Institute for Theoretical Chemistry, University of Vienna, 1090 Vienna, Austria

Abstract. Using 512 bit Advanced Vector Extensions, previous development history and Intel documentation, BNF grammar based genetic improvement automatically ports RNAfold to AVX, giving up to a 1.77 fold speed up. The evolved code pull request is an accepted GI software maintenance update to bioinformatics package ViennaRNA.

Keywords RNA secondary structure prediction, Genetic programming, GGGP, SIMD parallel computing, software engineering, RCS, SBSE

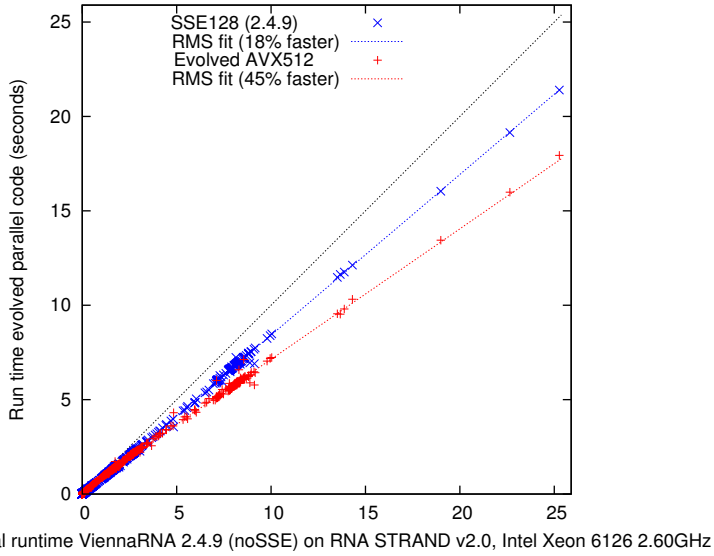

Fig. 1. Before and after GI elapsed time of RNAfold on 4663 RNA molecules.

1 Background: RNA, Genetic Improvement and RNAfold

RNA (like DNA) is a long chain biomolecule whose individual components (bases) have distinct side-binding affinities as well as forming strong links along the chain. Side bindings between elements of the chain cause RNA molecules to fold up in particular ways (known as their secondary structure). Giving them more or less stable shapes. The shapes of RNA molecules strongly influence their chemistry, including gene regulation. The ViennaRNA package [1], particularly

RNAfold, is often used to predict secondary structures from RNA base sequence data. We show genetic programming can automatically convert existing RNAfold code to new AVX 512 bit parallel instructions.

Unlike our earlier work with pknotsRG [2] initially there wasn't a parallel version of RNAfold. However in [3] we used grow and graft genetic programming to create a parallel vector (SSE128) based implementation for ViennaRNA release 2.3.0. This was adopted and has been shipped with the ViennaRNA package since release 2.4.7.

At the time we were frustrated in our attempt to go as far as the full 512 bit Advanced Vector Extensions (AVX512) as they were only supported in specialised (GPU like) hardware accelerators, i.e. the Intel Phi cards. (ViennaRNA also includes our GPU version of RNAfold [4].) Since then AVX has been made available in certain top end Intel CPU chips and so we renewed our attempt to automatically port RNAfold to AVX512 hardware.

As before we use our grammar based GP system [5]. The BNF grammar is automatically created from our SSE128 code (itself based on ViennaRNA release 2.3.0), the complete revision control system (RCS) history of the manual code we generated for it and the Intel documentation (`https://software.intel.com/sites/landingpage/IntrinsicsGuide/#` downloaded 26 Jan 2017). At 6122 rules (although only 631 are used) this is perhaps the largest GP grammar.

1.1 RNA_STRAND

We test our evolved version of RNAfold on more than four thousand curated RNA molecules given by RNA_STRAND [7]. All 4666 RNA molecular base sequence were downloaded from http://www.rnasoft.ca/strand/download/

1.2 GGGP Genetic Improvement System

We use genetic improvement [8,9,5,10] as part of our Grow and Graft GP (GGGP) [11] approach to optimising program sources. Previously [3] we had profiled RNAfold using GNU gcov. It indicated almost all the execution time was taken by a small fraction of function E_ml_stems_fast in multibranch_loops.c Just four lines of C code inside nested for loops (given in Figure 2) were responsible, since they were executed billions of times. Previously [3], RNAfold was sped up by porting them to 128 bit SSE parallel Intel instructions. Here genetic programming ports them to 512 bit operations.

```
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);
  }
}
```

Fig. 2. Original RNAfold 2.3.0 code in function E_ml_stems_fast()

Table 1. GGGP to improve SSE version of RNAfold

Representation:	variable list of replacements, deletions and insertions into BNF grammar (6122 rules)
Fitness:	Compile (GCC 7.3.1 -O2, -DNDEBUG -march=native -mtune=native) to modified object code, run on 10 000 random test cases. Two objectives: number of tests past and elapsed time (see Section 2.10).
Population:	5000, panmictic, elite 10, generational.
Parameters:	Initial population of random single mutants. 50% truncation selection. 50% two point crossover, 50% mutation. No size limit. Stop after 50 generations.

1.3 Parallel SSE 128-bit and AVX 512-bit Vector Instructions

At the end of the second millennium Intel started to increase the parallelism of its flagship 8080 series of processors. The most successful approach remains to integrate a few CPU cores onto single silicon chips. However at about the same time, Intel extended the instruction set with SIMD (single instruction multiple data) vector operations. Initially these allowed up to four 32 bit operations in parallel but the SSE instruction set has been progressively extended and now some processors support AVX (Advanced Vector Extensions) of 512 bits (e.g. sixteen 32 bit int operations in parallel).

During the original manual phase, intermediate versions of the manually written SSE code were held in a revision control system (RCS), see reference [3, Figs. 2 and 3]. The appendix (Figure 9) shows the initial (SSE 128) seed code. Additionally the whole of the Intel SSE/AVX library was available to genetic programming via the automatically created grammar, as an extended code base [12]. 5694 rules were derived from the SSE/AVX documentation. A further 168 were derived from RCS. Finally 141 came from the manually written SSE code Figure 9 (i.e. the usual GI source seed code [5]). Of these 68 are fixed and provide the framework within which GI operates on the remaining 73.

The next section (2) deals with setting up the grammar (2.1) and type restrictions (2.2–2.5) before dealing with evolutionary parameters, such as population size (2.6, see also Table 1), creating the initial population (2.7) crossover and mutation (2.8) and fitness and selection (2.9–2.12). Some sections, e.g. 2.5, are very detailed and included for completeness.

Section 3 gives the results. Whilst Section 4 discusses, although it was not needed here, if a Pareto approach might been better and suggests that perhaps, despite the noisy fitness function, elitism is not in fact necessary. Finally we conclude (Section 5) that the evolved AVX 512 code is more than six times faster than the sequential code.

2 The Grammar Based Genetic Programming System

For our genetic programming [13,14,15] system we used GISMOE [5,16,17,18]. GISMOE (Figure 5) creates a BNF grammar, which represents the original program's source code and legitimate changes to it. It then creates and evolves a population of changes to the BNF. Each modified BNF is expanded to give a modified (i.e. mutated) C source file, which is then compiled (Section 2.9) and tested (Section 2.10).

```
const int end = 1 + stop - k;
int       cnt;
__m128i   inf = _mm_set1_epi32(INF);

for (cnt = 0; cnt < end - 3; cnt += 4) {
  __m128i   a     = _mm_loadu_si128((__m128i *)&fmi_tmp[k + cnt]);
  __m128i   b     = _mm_loadu_si128((__m128i *)&fm[k1j + cnt]);
  __m128i   c     = _mm_add_epi32(a, b);
  __m128i   mask1 = _mm_cmplt_epi32(a, inf);
  __m128i   mask2 = _mm_cmplt_epi32(b, inf);
  __m128i   res   = _mm_or_si128(_mm_and_si128(mask1, c),
                                 _mm_andnot_si128(mask1, a));

  res = _mm_or_si128(_mm_and_si128(mask2, res),
                     _mm_andnot_si128(mask2, b));
  const int en = horizontal_min_Vec4i(res);
  decomp = MIN2(decomp, en);
}

for (; cnt < end; cnt++) {
  if ((fmi[k + cnt] != INF) && (fm[k1j + cnt] != INF)) {
    const int en = fmi[k + cnt] + fm[k1j + cnt];
    decomp = MIN2(decomp, en);
  }
}

k   += cnt;
k1j += cnt;
```

Fig. 3. Start point for genetic improvement (ViennaRNA patch 2.4.9). Notice SSE 128 code already added [3], cf. Figure 2.

2.1 Representation: Variable length Genome

Each variable length linear GP individual is an ordered list of changes to the original BNF grammar. Each generation, mutation can append a new change to the list or two point crossover can create a new list from randomly chosen parts of two parent lists [19, p18]. (Section 3.2 contains an example individual.)

2.2 BNF Generic Vector Types veci (m128i, m256i or m512i)

Since we want evolution to have access to the new vector instructions (i.e. 256 and 512 bit) but these are not used in the existing code, we automatically convert BNF grammar rules for existing 128 bit SSE code to a generic vector type "<veci_". (See also next section.) This can be either 4 int, 8 int or 16 int. There are three special meta mutations vecsize=4, vecsize=8 and vecsize=16, which convert veci in BNF rules to m128i, m256i or m512i types. An individual can contain multiple vecsize= meta-mutations. Variable length linear GP individuals are interpreted from left to right. Therefore veci is converted to whichever m*nnn*i type is active. (By default the original code, i.e. m128i, is active.)

Everyone in the initial population must be unique, therefore there are initially only three individuals contain vecsize= metamutations. However during evolution their number climbs rapidly to fill the whole population and many individuals contain more than one. However the three sizes are approximately equally prevalent.

```
      const int end = 1 + stop - k;
      int       cnt;
      __m128i   inf = _mm_set1_epi32(INF);

#ifdef AVX512
      for(cnt = 0; cnt < end - (16- 1); cnt += 16)  {
        __m512i   a     = _mm512_loadu_si512((__m512i*)&fmi[k + cnt]);
        __m512i   b     = _mm512_loadu_si512((__m512i*)&fm[k1j + cnt]);
        __m512i   c     = _mm512_add_epi32(a, b);
        __m512i   min1  = _mm512_shuffle_epi32(c, _MM_SHUFFLE(0,0,3,2));
        __m512i   min2  = c;
        __m512i   min3  = _mm512_shuffle_epi32(min2, _MM_SHUFFLE(0,0,0,1));
        __m512i   min4  = _mm512_min_epi32(min2,min3);
        en =  _mm512_reduce_min_epi32(min4);
        decomp = MIN2(decomp, en);
      }
#endif /*AVX512*/
      //second for loop used by both 128 bit SSE and 512 bit AVX code
      for (; cnt < end; cnt++) {
        if ((fmi[k + cnt] != INF) && (fm[k1j + cnt] != INF)) {
          const int en = fmi[k + cnt] + fm[k1j + cnt];
          decomp = MIN2(decomp, en);
        }
      }

      k   += cnt;
      k1j += cnt;
```

Fig. 4. AVX 512 code automatically evolved from Figure 3

2.3 veci Mutations: Switching between m128i, m256i or m512i

Half mutations are randomly chosen to be vecsize= meta mutations (see previous section). The size, 4, 8 or 16, is chosen uniformly at random.

2.4 BNF Grammar Type Matching

The input code (appendix Figure 9) leads to 12 BNF Grammar types:

39 “”	statement
2 for1	first part of for loop
2 for2	second part
2 for3	last part of for loop
2 inti	int
1 int(veci)	int function with generic input
8 const03	0, 1, 2 or 3
2 unsigned-char(const03,const03,const03,const03)	function with 4 inputs
9 veci	generic vector
2 veci(veci-const*)	function pointer input, return vector
2 veci(veci,int)	function with vector and int arguments
3 veci(veci,veci)	function with two vector inputs

The first four types have been repeatedly used by GISMOE [5,20]. There are 39 simple lines (i.e. type “”). These can be deleted, replaced by another line, swapped or inserted before another line. All genetic operations are between BNF grammar rules of the same type. The two for loops in the input code (appendix Figure 9) each give rise to three rules (of types: for1, for2 and for3).

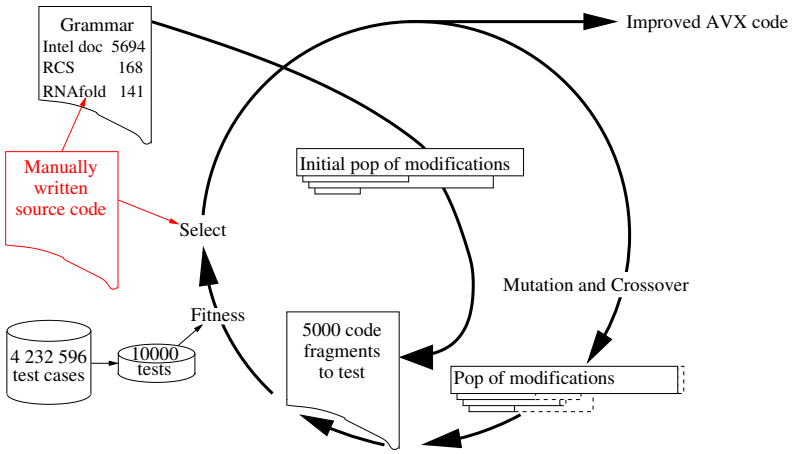

Fig. 5. Genetic Improvement evolutionary cycle starts with the original C code (red, see appendix Figure 9). The grammar tries to ensure many mutants compile, run, and terminate in ≤ 2 seconds. Fitness is given by run time and comparing with the original code's answer.

2.5 Vector Type Matching Rules

The remaining types (i.e. "inti" to the end of the table in Section 2.4) are new and relate to expressions rather than complete lines of code. They can be deleted, replaced and exchanged. Rather than leave a hole in the source code, each type has a delete operation. For example: deleting a constant actually causes it to be replaced by zero, deleting a vector causes it to be replaced by a default vector "a", deleting a pointer to vector means replacing it with a pointer to "a" and deleting functions causes them to be replaced by their first argument. Notice 5 types include "veci" and so expand to 15 possibilities, making 22 types in total.

The Intel SIMD intrinsics documentation defines 5694 library functions. The documentation includes their input arguments' types and return type. This gives 1337 types. However to be used the types must match rules given by the input code (appendix Figure 9). There are several aspects of matching. Firstly, the input code uses generic veci types (Section 2.2 above). Secondly, the Intel intrinsics use multiple types to represent pointers and integers. For example, the type int(veci) matches functions which return int, mmask8, mmask16, mmask32 or unsigned-int and (depending on vecsize, Section 2.2) takes vector inputs of type m128i, m256i or m512i. The possible function type matches for this RNAfold experiment are given in Table 2.

When a new mutant code change is made, the Intel intrinsics whose return type matches the current vecsize are eligible and one of them (depending upon mutant type) maybe chosen. However if there is a type error during fitness evaluation when the mutated C code is generated, there is a fixup process which selects the Intel intrinsic of the right type whose name matches as closely as possible. Also there are a few veci(veci,int) and veci(veci,veci) types where veci takes dif-

Table 2. Each box of three columns lists the number of Intel intrinsics which match the generic type above the table. The columns are (left to right) for 128, 256 and 512 bit vectors. I.e. 4, 8 and 16 int parallel operations.

int(veci)

int(m128i)	3	int(m256i)	2	int(m512i)	7
mmask8(m128i)	3	mmask8(m256i)	2	mmask8(m512i)	1
mmask16(m128i)	1	mmask16(m256i)	1	mmask16(m512i)	1
mmask32(m128i)	0	mmask32(m256i)	1	mmask32(m512i)	1
unsigned(m128i)	0	unsigned(m256i)	0	unsigned(m512i)	2

veci(veci-const*)

m128i(m128i*)	1	m256i(m256i*)	0	m512i(m512i*)	0
m128i(m128i-const*)	4	m256i(m256i-const*)	4	m512i(m512i-const*)	0
m128i(void-const*)	6	m256i(void-const*)	0	m512i(void-const*)	5

veci(veci,int)

m128i(m128i,int)	16	m256i(m256i,int)	8	m512i(m512i,int)	6
m128i(m256i,int)	2	m256i(m128i,int)	0	m512i(m128i,int)	0
m128i(m512i,int)	2	m256i(m512i,int)	2	m512i(m256i,int)	0
m128i(m128i,const-int)	4	m256i(m128i,const-int)	0	m512i(m128i,const-int)	0
m128i(m256i,const-int)	2	m256i(m256i,const-int)	13	m512i(m256i,const-int)	0
m128i(m512i,const-int)	0	m256i(m512i,const-int)	0	m512i(m512i,const-int)	3
m128i(m128i,unsigned)	1	m256i(m256i,unsigned)	1	m512i(m512i,unsigned)	9
				m512i(m512i,MM-*)	4

veci(veci,veci)

m128i(m128i,m128i)	136	m256i(m256i,m256i)	118	m512i(m512i,m512i)	110
		m256i(m128i,m128i)	2	m512i(m512i,m128i)	9
		m256i(m256i,m128i)	9		

Total usable intrinsics 502 + 5 (see end of Section 2.5) = 507
All these available intrinsic functions are used by evolution.

ferent values. Since vecsize can only take one value at a time, these are resolved by the fixup process. E.g., m128i(m256i,int) (3rd box in Table 2) may be replaced by one of the m128i(m128i,int), m256i(m256i,int), m512i(m512i,const-int), etc. functions. Which one depends upon the current vecsize but is fixed so that the mapping from genotype to phenotype is deterministic.

In addition to the 502 intrinsics in Table 2, during evolution (Section 3), the fixup process substitutes other rules from the input (17), the history (10), an additional 5 Intel intrinsics (not in the table) or either stub. (Two stub functions, _mm_cvtsi256_si32 _mm_cvtsi512_si32, are missing from the Intel intrinsics documentation. They were defined by hand in C code to correspond to _mm_cvtsi128_si32.)

2.6 Population Size

In order to give a reasonable chance of including most of the eligible Intel SIMD intrinsics, the population size was increased dramatically to 5000 mutants. The initial generation (Section 3) used 396 intrinsics directly.

2.7 Initial Population

The initial population of 5000 unique individuals is created using mutation.

2.8 Genetic Operations: Mutation and Crossover

Half the new members of each generation are created by mutation and half by two point crossover. Excluding the small elite (Section 2.12), every member of the generation is unique. Both mutation and crossover attempt to avoid repeating the same changes by removing such duplications from the new child's genome before applying unique requirements. Similarly both apply limited scoping checks to the genome [11,21]. If any of these checks fail, the change is discarded and a new random change is attempted.

As mentioned above (Section 2.3) half the time mutation attempts to set vecsize. (However the uniqueness requirement limits the extent to which vecsize floods the population.) If chosen, vecsize is inserted at a random point in the individual's genome. Whereas other mutations are appended to it.

If not a vecsize mutation, one of the 73 variable BNF grammar rules from the input code is chosen at random to be the target of a change. There are four changes: delete, replace, insert and swap. Any grammar rule can be deleted but for replacement, insert and swap, the replacement grammar rule must be of the same type.

2.9 Compiling gcc `-O2 -DNDEBUG -march=native -mtune=native`

Both reference (see Section 2.11) and evolved code were compiled with the same compiler (GCC 7.3.1) and options (`-O2 -DNDEBUG -fmax-errors=1` and `-march=native -mtune=native`). Notice we use the same default optimisations (`-O2 -DNDEBUG`) as the ViennaRNA release kit. The gcc command line option `-DNDEBUG` is used to disable some runtime checks in the supporting code. It does not directly affect the evolved code. Since any compilation error is regarded as fatal, `-fmax-errors=1` is used to terminate failed compilations as quickly and tersely as possible.

By default the compiler will not generate either SSE or AVX instructions. Therefore, `-march=native -mtune=native` gcc command line options are used to generate optimised SSE and AVX instructions.

As before [3], we prevent some semantically equivalent mutants by insisting that the compiled object code is not identical to that of the un-mutated code.

2.10 Fitness Function: Run 10 000 times, Elapsed time & Accuracy

To get a realistic fitness function for training the GI, we started by profiling the RNAfold code on a real RNA molecule. This logged all 4 232 596 calls to the modular decomposition code in order. Each call was converted into a test case, giving the inputs to the code and the required output (i.e. the test oracle). Due to the way RNAfold uses dynamic programming, these test cases are also roughly ordered by difficulty and run time.

To both decrease the chance of over fitting and to reduced run time only a fraction of the test cases are used. To give a good spread of test case difficulty and runtime the 4 232 596 are divided into five equal sequences. Each generation

a sequence of 2000 tests is chosen uniformly at random from each group. Making a total of 10 000 test cases.

Assuming the mutant compiled ok, it is run and the number of tests passed and time taken, for both the original code and the mutant code, is recorded. The whole process is subject to a total CPU limit of two seconds. (Normally it will take about 0.2 seconds.)

2.11 Selection for Speed and Correctness

As is often the case, we have two aspects of fitness. 1) the code should be accurate. 2) it should be fast. These are decided by testing (see previous section). Instead of combing these in a fixed way our selection is similar to VEGA [22]. (See also Section 4.) All the individuals which completed the tests (i.e. did not abort during testing) and passed at least one test are sorted into two lists by speed and accuracy. One list is sorted first by number of tests passed (ties are resolved by relative elapsed time). The second is sorted by relative elapsed time. Ties are unlikely but for symmetry are resolved by number of tests passed. (To compensate for short term fluctuations in computer speed we use the ratio of run time of the mutant and that of the reference code measured in the same exe image within a few milliseconds of each other. Indeed to obtain stable results both reference and mutated code are run eleven times and the first quartile, i.e. 3rd fastest, run time is used.) Mutants are taken alternatively from the two lists. When a mutant has already been selected, because it came earlier in the other list, it is not selected again. In effect that list loses a turn and selection goes immediately to the other list.

As mentioned in Section 2.8, two children are created in the next generation per selected parent. However if less than half the population are selected to be parents the next generation is made up to the full population size by creating random individuals.

It seems this selection does reduced the tendency of the population to converge but also the population seems to diverge into two separate camps (as reported for VEGA [23]), see Figure 7.

2.12 10 Member Elite

Up to the first ten members of the list sorted by tests passed (see previous section) are automatically carried unchanged directly into the next generation [24, p 101]. Normally they will also be parents of children (created with additional mutations or by crossover).

Since it is easy to retain the best of each generation, we had not been overly worried by evolution's tendency to lose (even the best) individuals from the population. Nonetheless, due to the noisy fitness function, we tried elitism here and it did indeed stabilise the best (see Figures 6 and 7). Nonetheless having the best of run individual in the last generation does not seem a big gain and in practise we were still free to chose the best mutant from any generation (Section 3.2).

3 Results

The next section describes the evolutionary processes, before Section 3.2 describes why we chose the best mutant from generation 34. Section 3.3 explains how its genome translates into coding changes. Whilst Section 3.4 shows it gives a six fold speed up. Then Section 3.5 measures it in situ, i.e. within RNAfold, and shows it speeds up the whole of RNAfold by 45% on real RNA molecules.

3.1 Evolving the Population

Several hundred individuals which pass all their tests and are more than 10% faster evolved (Figure 6 red ×). But most of population evolve to be either much faster and fail many tests or pass all their tests but give no speed up Figure 7. Figure 7 highlights new code which passes all its tests and is at least 40% faster (×) or is more than 60% faster (+).

During evolution 13% of mutants fail to compile, 0.3% compile ok but their object code is identical to the seed code's (end of Section 2.9), 1.5% fail at runtime (e.g. segfault or CPU time limit exceeded) and 85% run all ten thousand tests.

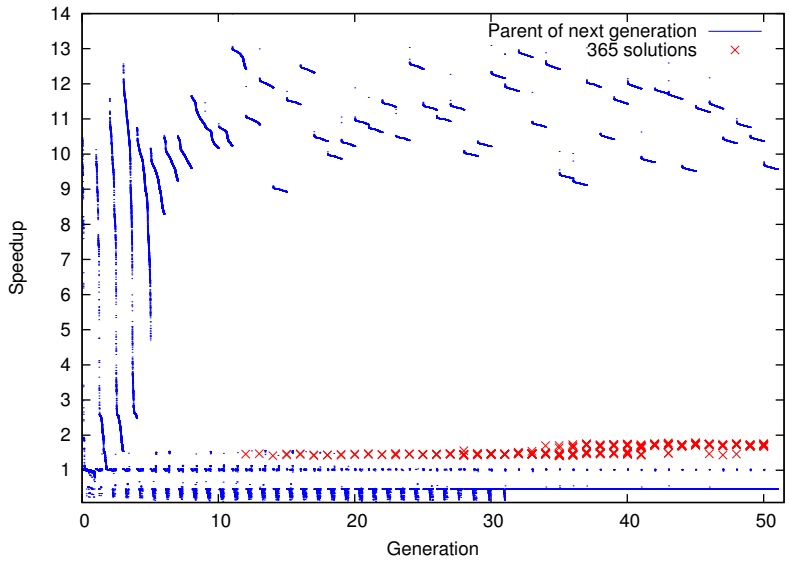

Fig. 6. Evolution of breeding population with elitism. Red × evolved individual passes all tests and speed up exceeds 10%. With our two criteria selection (Section 2.11, approximately half each breeding population is faster than the original code but fails one or more tests (see also Figure 7).

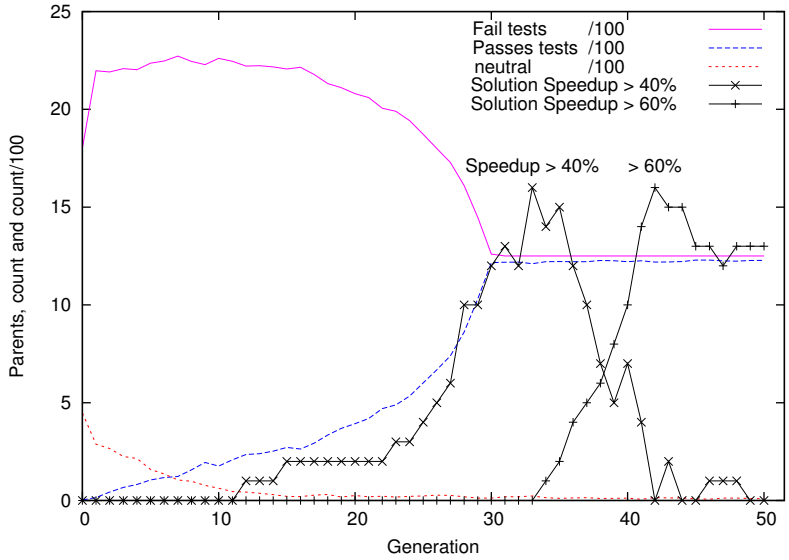

Fig. 7. Evolution of breeding population (2500). Solutions black lines (×, +). Initially most parents fail tests but are faster (solid purple), until gen 30. when the parents which pass all the tests but give no speed up (blue dashed) has grown to almost half (1216). Initially 448 parents (red dashed) pass all 10 000 tests but give no speed up. Gen 50 ≈50% are >10 fold faster but fail many tests (Fig. 6).

3.2 Choosing the Winner

We choose the best individual in generation 34 since it was the first to be more than 60% faster (actually 69.9%) and pass all 10 000 of the tests used in its generation. This evolved individual contains 8 mutations:

vecsize=16
<_modular_decomposition.c_100>x<_modular_decomposition.c_77>
<veci_2modular_decomposition.c_110>
<int(veci)_1modular_decomposition.c_116><int(m512i)_IntrinsicsGuide.txt_8506>
<veci(veci,int)_1modular_decomposition.c_112><m256i(m512i,int)_IntrinsicsGuide.txt_4614>
<_modular_decomposition.c_97>x<_modular_decomposition.c_84>
<const03_5modular_decomposition.c_114>
<veci(veci,veci)_1modular_decomposition.c_113>

3.3 Explaining the Wining Evolved Program

The evolved individual (8 mutations) can be reduced to three, since:

- Swapping lines 100 and 77 makes no difference.
- Similarly swapping lines 97 and 84 makes no difference.
- Deleting rule <veci_2modular_decomposition.c_110> again makes no difference (since delete substitutes a default variable for the deleted one, which turns out to be identical).

– Similarly <const03_5modular_decomposition.c_114> replaces a zero with another zero.
– The mutation which changes the right hand side of line 112 also makes no difference since the variable it writes to (min1) is no longer used. (Originally it was set on line 112 and used on line 113.)

The three remaining essential changes that are left are: vecsize=16 <int(veci)_1modular_decomposition.c_116><int(m512i)_IntrinsicsGuide.txt_8506> <veci(veci,veci)_1modular_decomposition.c_113>

– vecsize=16 converts all generic SSE code to 512 bit code.
– Line 8506 of the Intel documentation IntrinsicsGuide.txt is _mm512_reduce_min_epi32, so <int(veci)_1modular_decomposition.c_116><int(m512i)_IntrinsicsGuide.txt_8506> causes en = _mm_cvtsi128_si32(min4) to be replaced with en = _mm512_reduce_min_epi32(min4). This is a key step, since it causes the output of the code (en) to be set to the minimum of sixteen (32 bit) int values held in 512 bit vector min4.
– The remaining mutation, increases efficiency by allowing the optimising compiler (note -O2) to avoid calculating a result by spotting it is being written into a variable which is not used.

Other efficiency gains might be possible by removing other unneeded operations. (Conceivable the compiler has already splotted them without needing explicit changes to the source code.) As is usual in GI practise [5], only the critical changes were retained. The evolved code gives exactly the same answers as the original sequential code.

3.4 Improved Performance inside RNAfold

To estimate the total improvement in performance we ran RNAfold on a long real sequence eleven times: for no SSE, for the 2.4.9 SSE128 released code and for the new evolved AVX512 code. To allow a regression plot (see Figure 8) each of these were repeated with the critical code repeated from one to ten times. The gnuplot (version 4.6.2) fit function was used to calculate the RMS least errors linear fit (Figure 8).

In summary, on long sequences, the SSE128 code in release 2.4.9 is 3.428 ± 0.002 times faster than the nonSSE code and the newly evolved AVX512 code is 1.774 ± 0.003 faster than the released SSE128 code. I.e. 6.083 ± 0.002 times faster than the non SSE release 2.4.9 code.

3.5 Performance on Hold Out Data

The new mutant AVX 512 code was inserted into the current release (2.4.9) of RNAfold by hand and tested on the whole of RNA_STRAND (see Figure 1). In all 4 666 cases it gives identical results. Even on the five corrupt fasta strings in RNA_STRAND v2.0 where RNAfold fails, the evolved and the released code give the same error messages. It is on average 45% faster. (Remember this includes the whole code, not just the dynamic programming inner loop.)

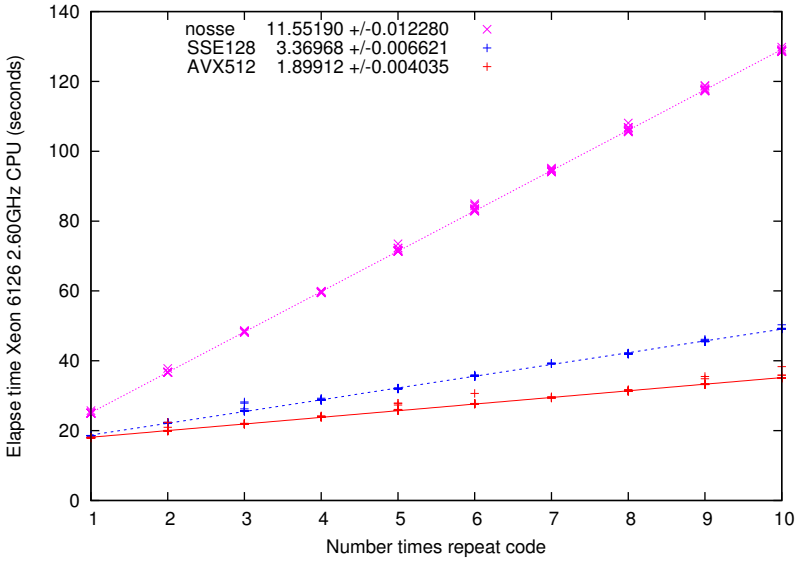

Fig. 8. Estimating true runtime with linear regression. Running multiple times (x-axis) allows the individual elapsed time to be estimated from the gradient. CRW_00528 (4382 bases) using newly evolved AVX512 (lowest +) code, the release 2.4.9 code configured with –enable-sse (+) and without (nosse × top).

4 Discussion: Population Convergence

The (VEGA [22] like) selection by two individual objectives (Section 2.11) avoids arbitrary weighting of the objectives but it appears (as suggested by Goldberg [23]) that it leads to the population dividing between the two objectives, so some programs are very fast but pass few tests and others pass all their tests but give little speed-up (Figure 7). Whereas an approach like NSGA II [25] could potentially lead to more intermediate mutants (i.e. give a speed up but fail some tests) [26,27]. It may be having the larger population allows sufficient diversity in the two sub-populations to allow evolution to progress without the need for more refined selection techniques.

It appears that the use of an elite group ([24, p 101] Section 2.12) does stably allow its preservation from one generation to the next. However the number of good fast mutants does not grow exponentially but quickly stabilises at three or four more than the size of the elite. As mentioned in Section 2.12, it may be these dozen or so high fitness individuals are not essential to allow evolution to progress.

5 Conclusions

Using a standard computer under a standard operating system (Linux) without specialised customisation to either, we have demonstrated that evolution can optimise a critical function written in C. Automatically creating AVX instructions, it gives almost a doubling in speed (1.71 fold) on top of hand written SSE instructions (6.1× the sequential code).

The new code has been included in ViennaRNA since release 2.4.11.

Acknowledgements

Funded by EPSRC grant EP/M025853/1.
GP code is available via the author's home page in `ftp/gp-code/rnafoldAVX.tar.gz`

References

1. Lorenz, R., Bernhart, S.H., Höner zu Siederdissen, C., Tafer, H., Flamm, C., Stadler, P.F., Hofacker, I.L.: ViennaRNA package 2.0. AMB 6(1) (2011), `http://dx.doi.org/doi:10.1186/1748-7188-6-26`
2. Langdon, W.B., Harman, M.: Grow and graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In: Langdon, W.B., et al. (eds.) Genetic Improvement 2015 Workshop. pp. 805–810. ACM, Madrid (11-15 Jul 2015), `http://dx.doi.org/doi:10.1145/2739482.2768418`
3. Langdon, W.B., Lorenz, R.: Improving SSE parallel code with grow and graft genetic programming. In: Petke, J., et al. (eds.) GI-2017. pp. 1537–1538. ACM, Berlin (15-19 Jul 2017), `http://dx.doi.org/doi:10.1145/3067695.3082524`
4. Langdon, W.B., Lorenz, R.: CUDA RNAfold. Tech. Rep. RN/18/02, Computer Science, University College, London, Gower Street, London (27 March 2018), `http://dx.doi.org/doi:10.1101/298885`
5. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Trans. EC 19(1), 118–135 (Feb 2015), `http://dx.doi.org/doi:10.1109/TEVC.2013.2281544`
6. Langdon, W.B., Petke, J., Lorenz, R.: Evolving better RNAfold structure prediction. In: Castelli, M., et al. (eds.) EuroGP 2018. LNCS 10781, pp. 220–236. Springer (4-6 Apr 2018), `http://dx.doi.org/doi:10.1007/978-3-319-77553-1_14`
7. Andronescu, M., Bereg, V., Hoos, H.H., Condon, A.: RNA STRAND: The RNA secondary structure and statistical analysis database. BMC Bioinformatics 9(1), 340 (2008), `http://dx.doi.org/doi:10.1186/1471-2105-9-340`
8. Langdon, W.B.: Genetic improvement of programs. In: Matousek, R. (ed.) 18th International Conference on Soft Computing, MENDEL 2012. Brno University of Technology, Brno, Czech Republic (27-29 Jun 2012), `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/Langdon_2012_mendel.pdf`, invited keynote
9. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: a comprehensive survey. IEEE Trans. EC 22(3), 415–432 (Jun 2018), `http://dx.doi.org/doi:10.1109/TEVC.2017.2693219`
10. Langdon, W.B.: Genetically improved software. In: Gandomi, A.H., et al. (eds.) Handbook of Genetic Programming Applications, chap. 8, pp. 181–220. Springer (2015), `http://dx.doi.org/doi:10.1007/978-3-319-20883-1_8`

11. Langdon, W.B., Lam, Brian Y.H., Petke, J., Harman, M.: Improving CUDA DNA analysis software with genetic programming. In: Silva, S., et al. (eds.) GECCO 2015. pp. 1063–1070. ACM, Madrid (11-15 Jul 2015), http://dx.doi.org/doi:10.1145/2739480.2754652
12. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Specialising software for different downstream applications using genetic improvement and code transplantation. IEEE Trans. SE 44(6), 574–594 (Jun 2018), http://dx.doi.org/doi:10.1109/TSE.2017.2702606
13. Koza, J.R.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT press (1992)
14. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction. Morgan Kaufmann, San Francisco, CA, USA (Jan 1998),
15. Poli, R., Langdon, W.B., McPhee, N.F.: 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)
16. Langdon, W.B.: Genetic improvement of software for multiple objectives. In: Labiche, Y., Barros, M. (eds.) SSBSE. LNCS 9275, pp. 12–28. Springer, Bergamo, Italy (Sep 5-7 2015), http://dx.doi.org/doi:10.1007/978-3-319-22183-0_2, invited keynote
17. Langdon, W.B.: Genetic improvement GISMOE blue software tool demo. Tech. Rep. RN/18/06, University College, London, (22 Sep 2018), http://www.cs.ucl.ac.uk/fileadmin/user_upload/blue.pdf
18. Lopez-Lopez, V.R., Trujillo, L., Legrand, P.: Novelty search for software improvement of a SLAM system. In: Alexander, B., et al. (eds.) 5th edition of GI @ GECCO 2018. pp. 1598–1605. ACM, Kyoto, Japan (15-19 Jul 2018), http://dx.doi.org/doi:10.1145/3205651.3208237
19. Langdon, W.B., Lam, Brian Y.H., Modat, M., Petke, J., Harman, M.: Genetic improvement of GPU software. Genetic Programming and Evolvable Machines 18(1), 5–44 (Mar 2017), http://dx.doi.org/doi:10.1007/s10710-016-9273-9
20. Langdon, W.B.: Evolving better RNAfold C source code. Tech. Rep. RN/17/08, University College, London, (2017), http://dx.doi.org/doi:10.1101/201640
21. Langdon, W.B., Lam, Brian Y.H.: Genetically improved BarraCUDA. BioData Mining 20(28) (2 Aug 2017), http://dx.doi.org/doi:10.1186/s13040-017-0149-1
22. Schaffer, J.D.: Multiple objective optimization with vector evaluated genetic algorithms. In: Grefenstette, J.J. (ed.) Proceedings of an International Conference on Genetic Algorithms and the Applications. pp. 93–100. Carnegie-Mellon University, Pittsburgh, PA, USA (24-26 July 1985), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/icga1985/icga85_schaffer.pdf
23. Goldberg, D.E.: Genetic Algorithms in Search Optimization and Machine Learning. Addison-Wesley (1989)
24. De Jong, K.A.: An Analysis of the Behavior of a Class of Genetic Adaptive Systems. Ph.D. thesis, Computer and Communications Sciences, Michigan, USA (1975)
25. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. EC 6(2), 182–197 (Apr 2002), http://dx.doi.org/doi:10.1109/4235.996017
26. Langdon, W.B.: Genetic Programming and Data Structures. Kluwer, Boston (1998), http://dx.doi.org/doi:10.1007/978-1-4615-5731-9
27. Knieper, T., Defo, B., Kaufmann, P., Platzner, M.: On robust evolution of digital hardware. In: Hinchey, M., et al. (eds.). IFIP, vol. 268, pp. 213–222. Springer, Milan (Sep 8-9 2008), http://dx.doi.org/doi:10.1007/978-0-387-09655-1_19

```
  int
modular_decomposition(const int i, const int ij, const int j, const int turn, const int fmi[(2913+1)], const int fm[5000000]) {
  int k = 0;
  int k1j = 0;
  int stop = 0;
  int end = 0;
  int en = 0;
  int decomp = INF;
  k +=  i;
  k += turn;
  k += 1;
  k +=  1;
  k += -1;
  k +=  2;
  k += -2;
  k +=  3;
  k += -3;
  k1j += ij;
  k1j += turn;
  k1j += 2;
  k1j +=  1;
  k1j += -1;
  k1j +=  2;
  k1j += -2;
  k1j +=  3;
  k1j += -3;
  stop += j;
  stop += -turn;
  stop += -2;
  stop +=  1;
  stop += -1;
  stop +=  2;
  stop += -2;
  stop +=  3;
  stop += -3;
  {
  end += 1;
  end += stop;
  end += -k;
  end +=  1;
  end += -1;
  end +=  2;
  end += -2;
  end +=  3;
  end += -3;
  int i;
  for(i=0;i<end-3;i+=4) {
    __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);
    __m128i min1 = _mm_shuffle_epi32(c, _MM_SHUFFLE(0,0,3,2));
    __m128i min2 = _mm_min_epi32(c,min1);
    __m128i min3 = _mm_shuffle_epi32(min2, _MM_SHUFFLE(0,0,0,1));
    __m128i min4 = _mm_min_epi32(min2,min3);
      en = _mm_cvtsi128_si32(min4);
    decomp = MIN2(decomp, en);
  }
  for(;i<end;i++) {
      en = fmi[k +i]+fm[k1j+i];
    decomp = MIN2(decomp, en);
  }
  }
  return decomp;
}
```

Fig. 9. Starting point for evolution. C code derived from ViennaRNA (Figures 2 and 3). Pairs of lines of code were manually added before the for loops to provide feed stock for evolution [2] but appear not to have been helpful. Instead successful changes (Figure 4) come from the for loops themsleves.