Grow and Graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation

William B. Langdon, Mark Harman
Dept. of Computer Science, University College London Gower Street, WC1E 6BT, UK
W.Langdon@cs.ucl.ac.uk

ABSTRACT

Grow and graft genetic programming greatly improves GPGPU dynamic programming software for predicting the minimum binding energy for folding of RNA molecules. The parallel code inserted into the existing CUDA version of pknots was "grown" using a BNF grammar. On an nVidia Tesla K40 GPU GGGP gives a speed up of up to 10 000 times.

Keywords

SBSE; Medicine; Bioinformatics; Software engineering; Genetic Improvement

1. INTRODUCTION

Unlike protein folding, computer programs have had considerable success at predicting the three dimensional structure of RNA using the sequence of bases along the RNA molecule. These programs are based on finding the thermodynamically most stable structure, i.e. the one with the lowest free energy. Since guaranteeing to find the minimum energy requires considering all possible configurations, popular algorithms use Biologically feasible short cuts to reduce both time and space complexity. There are a number of algorithms which restrict their search to RNA structures no more complex than pseudo knots and yet have been shown to yield the correct structure for the vast majority of Biologically interesting RNA molecules [14, 15]. One such program is pknotsRG which uses dynamic programming.

pknotsRG [15] implements Reeder and Giegerich's algorithm for pseudo knot RNA energy folding. Although more recent programs are available, it has a number of advantages for our purposes:

- It is freely available. (See http://bibiserv.techfak.uni-bielefeld.de/adp/cuda.html)
- It is written in C and CUDA

GECCO'15 Companion, July 11 - 15, 2015, Madrid, Spain.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3488-4/15/07 $15.00.
DOI: http://dx.doi.org/10.1145/2739482.2768418 .

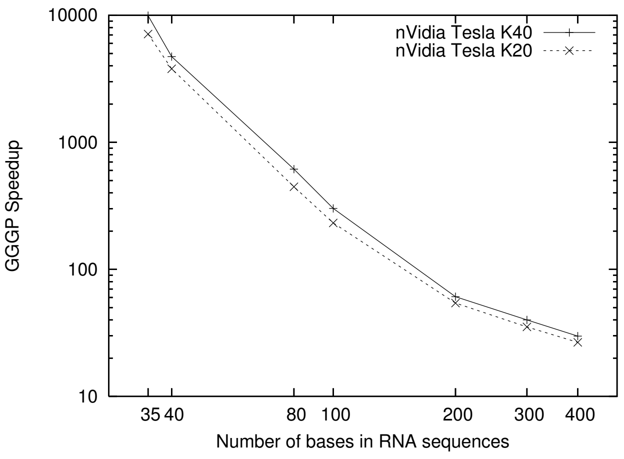

Figure 1: Ratio between original speed of CUDA version of pknotsRG and CUDA version after grow and graft change to allow processing multiple sequences in parallel.

- The same program is available both as a traditional C program and also as a parallel version running on graphics card GPU hardware.
- Although not as big as some programs we have successfully enhanced [9], at 11 000 lines of code, it is definitely non-trivial. (About one thousand lines of code relate to the graphics hardware.)

The RG pseudoknot Dynamic Programming algorithm is matrix based, with the key matrix being $(n+1)$ by $(n+1)$. Where n is the length of the string used to describe the RNA molecule's sequence of bases. (Although the pknotsRG actually only uses the lower triangle part of the matrix and does not store the other half.) With modern GPUs housing thousands of processors, a single RNA molecule of a few hundred bases does not represent sufficient computational load to effectively use all the available parallelism. However (as shown in Figure 1) enormous speedup are available using nVidia Tesla GPUs by processing strings representing different RNA molecules in parallel. (Figure 2 shows the wide range of RNA sizes typical of modest sized modern Bioinformatics datasets.)

Our goal was to demonstrate growing a small fragment of CUDA code and then grafting it into the existing pknotsRG code. Notice, as with our previous work [1], we give evolution a series of hints based on engineering knowledge. Firstly

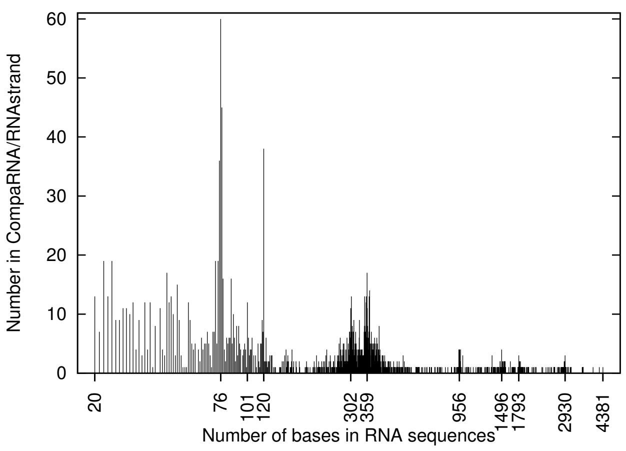

Figure 2: Distribution of lengths of RNA molecules. (From the CompaRNA version of RNAStrand which excludes RNA molecules with less than 20 bases.) Total 1987.

we strongly suspected that the performance bottleneck was due to processing each RNA's matrix in series, rather than all of them in parallel. (A single K40 Tesla GPU can readily process 200 000 such matrices. Table 1 summarises the capabilities of the two GPUs used.) Secondly we identified where in the existing parallel CUDA code we wanted the new code to be grafted in. In [1] the automated grafting process had more work to do in the sense the location was only given to the nearest module in Pidgin (a two hundred thousand line C program). In pknotsRG it seemed clear that the ideal location for the new code was at the start of the existing CUDA kernel and little benefit was anticipated in allowing the optimisation process to consider alternatives.

The next section gives full details of our grow and graft genetic programming (GGGP) system, including both the initial flawed experiments and the successful ones. Since solutions were found immediately in the 2nd approach, the busy reader may wish to skip past details of crossover and selection used to create subsequent generations and go directly to Section 3 which describes the results. Section 4 includes discussion of problems over come and the AI, Engineering and research implications.

2. THE GRAMMAR BASED GI SYSTEM

Much of the original code for pknotsRG was in fact automatically generated by the Algebraic Dynamic Programming (ADP) compiler [16]. The ADP compiler created the C and CUDA code from rules used to define the binding energies of interactions between RNA bases. As our previous Genetic Improvement (GI) work [9, 6, 8, 11, 10, 4] the downloaded code was automatically converted into a BNF grammar.

Originally it was intended to use the grammar rules from all the CUDA code as feedstock for the newly grown code [12]. However this was not tried, as it appeared most such rules contained variables that would be out of scope in the new location and hence lead to compilation errors. Instead, we attempted, only a little "plastic surgery" in which code is taken from elsewhere and reused in the graft. The only plastic surgery used was to gather constants (rather than lines of code) and make them available when growing the graft (see Section 2.3).

```
<gggpint_Kkernel_bnf.cu_57><gggpint_Kkernel_bnf.cu_128>
```

Figure 4: Example line replacement mutation. The format `<rule1><rule2>` causes rule1 to be replaced by rule2. So here line 57 is replaced by a copy of line 128. The type of each rule is given by the first part of its name. Here both rules are of type `gggpint`.

Initially a 206 rule grammar was automatically created from a modified form of the outer most function, calc_all, of the CUDA kernel. For reasons described in Section 2.1, this was further reduced to a grammar of 111 rules. In each case genetic programming [13] uses the grammar to control the mutations it can make. The grammar ensures that after each mutation the new code is syntactically correct. The grammar not only ensures the correct placement of CUDA language keywords, brackets, semi-colons, etc. but, in this case, also ensures that the mutated code compiles. As the evolving code does not include loops or recursion, it is easy to guarantee that it will terminate, and a fitness value can be assigned to each member of the population. (Section 2.6 will describe our use of CUDA MEMCHECK to handle evolved grafts which misbehave during run time fitness testing.)

2.1 Seed to be Grown and then Grafted

The outermost pknotsRG calc_all kernel calls seven other __device__ functions with the same twelve parameters. These include nine pointers to arrays. calc_all is designed to run one step of the dynamic programming algorithm on the triangular matrix for one RNA molecule. At each step calculations are performed in a diagonal line across the matrix, so at each step from 1 to $n + 1$ elements of the matrix are involved in the calculations. These can be done in parallel. Even for large RNA molecules from RNAstrand (Section 3.1) the amount of work the kernel does in parallel is below the number of threads needed to keep a K40 busy. Hence to exploit the massive parallelism available in GPUs the idea was to plant some seed code into the existing calc_all to allow it to process multiple RNA sequences simultaneously.

The seed code intercepts calc_all's eleven existing parameters. It provides a gap for evolution to fill in. Into this gap evolution puts code to update the twelve parameters passed to the seven __device__ functions. The seven __device__ functions are not changed.

In the first attempt to evolve a solution, the whole of calc_all including the manually created initial seed was converted to a BNF grammar and evolved. After various teething problems and bug fixes it was realised that the initial seed was fatally restricted and could not, as posed, be evolved into a solution. In the second attempt it was realised that there was no need to evolve the rest of calc_all and evolution should concentrate upon growing the graft (i.e. on the seed code). Therefore only the seed code was converted into a BNF grammar (see Figure 3). This reduced it from 206 rules of ten types to 111 rules of 4 types and lead *immediately* to a solution which was grafted into pknotsRG.

2.2 Variable Code: Grammar Types

Each line of the source code is represented by one or more rules in the BNF grammar. Evolution changes the seed code. Each variable rule belongs to one of ten (reduced to four) types. Mutations only copy code between rules of the same type.

Table 1: GPU Hardware. Each GPU chip contains 13 or 15 identical independent multiprocessors (MP, column 4). Each MP contains 192 stream processors. (Total given in column 6). The next two columns give sizes of the L1 and L2 caches used. ECC enabled.

GPU	Introduced	compute level	MP	total cores	Clock	L1 cache	L2 cache	Memory	Bandwidth
Tesla K20	2012	3.5		$13 \times 192 = 2496$	0.71 GHz	48 KB	1.25 MB	5 GB	140 GB/s
Tesla K40	2013	3.5		$15 \times 192 = 2880$	0.88 GHz	48 KB	1.50 MB	11 GB	180 GB/s

```
int i;
int j;
const int gggp_thread = blockIdx.x*blockDim.x+threadIdx.x;
//const int gggp_x = tree1(gggp_thread,gggp_nrna,diag,d_n,d_max_pknot_length);
//const int gggp_y = tree2(gggp_thread,gggp_nrna,diag,d_n,d_max_pknot_length,gggp_x);
//const int gggp_I = tree3(gggp_thread,gggp_nrna,diag,d_n,d_max_pknot_length,gggp_x,gggp_y);
const int gggp_x =  d_max_pknot_length + 1;
const int gggp_y =  gggp_x - diag + 0;
const int gggp_I =  gggp_thread/gggp_x;
const int gggp_I1 = (d_max_pknot_length+1)*gggp_I;
const int gggp_IJ = ((d_max_pknot_length*(d_max_pknot_length+1))/2+
                     d_max_pknot_length+1)*gggp_I;
i = gggp_thread- gggp_y*gggp_I;
j = i+diag;

printf("diag %d thread %d gggp_I %d gggp_I1 %d gggp_IJ %d i %d j %d\n",
       diag, gggp_thread, gggp_I,  gggp_I1,   gggp_IJ,    i,   j);
```

Figure 3: Manually written seed in 2nd experiments. Reformatted for clarity. The tree1,2,3 comments indicate an s-expression using the variables listed as arguments should be evolved. The grammar did not support this directly. Instead the evolvable code above was used. Fitness is based on `gggp_I`, `gggp_I1`, `gggp_IJ`, `i` and `j`. (`diag` is included with `printf` as a debug aid.)

Ten of the variable rules were simple lines, mostly calls to __device__ functions. They could be exchanged with each other, or indeed inserted before other statements. However as they all referred to code outside the seed, they were all removed in the second version. Similarly the three IF rules disappeared in the second version. The 8 (later 6) constants are represented by constant integer type rules (see Section 2.3).

Originally each of C types had a corresponding BNF rule type (there were four of these) but this reduces to just gggpint for the `int` C type in the second version. That is, in the final grammar there are 17 gggpint rules.

In all cases the code compiled.

2.3 Integer constants

All the integers used in the source and Makefile (see Figure 5), plus INT_MAX (a total of 86) were extracted and converted into BNF typed grammar rules like those used to represent mutable code. They are given the same type as `int` constants used in the seed code, which means any constant in the seed code can be replaced with any constant (or INT_MAX) taken from anywhere in the original program.

2.4 Initial Population

The initial population of 300 unique individuals is created using mutation.

2.5 Genetic Operations: Mutation and Crossover

Mutation and crossover were used in the first experiments. In the second a solution was found in the first generation so there was no need to breed a second.

In the first experiments, each member of the breeding pool, i.e. the top 150 individuals from the previous generation, is allocated two children, one to be created by mutation and one via crossover (between the selected parent and another randomly selected from the breeding pool). If less than 150 individuals in the previous generation were selected, the missing children are created at random in the same way as the initial population was created.

New code mutations are made by appending another code mutation. There are three types of code mutation: delete a line, copy and replace the target line and copy and insert before the target line. (Figure 4 gives an example of a copy and replace mutation.

Insert mutations are denoted by a `+` between the rule names. For example, `<_Kkernel_bnf.cu_161>+<_Kkernel_bnf.cu_170>` which causes line 170 `__syncthreads();` to be inserted before line 161 `calc_knot(`$\cdots$`);`

Crossover acts on the genetic representation giving a new child by two point crossover like Koza's sub-tree crossover [3]. Notice crossover only selects genes from its two parents, it does not create new ones. If crossover is unable create a new unique offspring after a small number of tries, the child is created by mutating the parent instead.

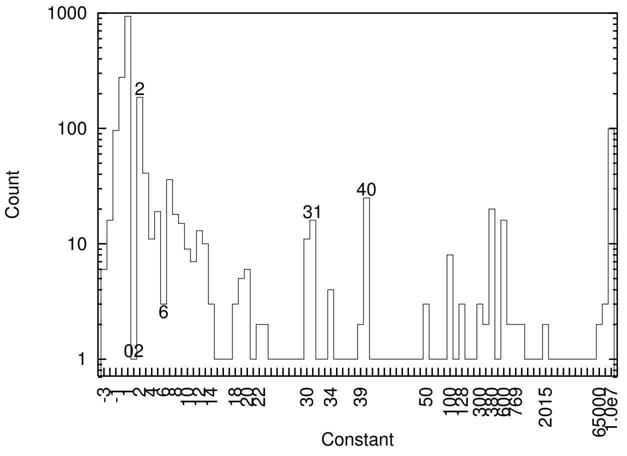

Figure 5: Integer constants in the C/CUDA sources of pknotsRG. To reduced clutter on the x-axis only selected numbers given.

2.6 Ensuring Mutated Code Runs

Much of the CUDA code uses pointers and macros to access arrays. Whilst it was possible to annotated these with `assert` statements to ensure array indices did not exceed their array's bounds this was not sufficient to catch all pointer errors and so during fitness testing evolved code was run inside the nVidia CUDA MEMCHECK tool. MEMCHECK imposes a considerable run time penalty. To reduce run time, the evolving graft was only tested on a small number of strings (actually five) and these were quite short (35 bases). Thirty five randomly chosen bases being enough so that there is some interesting self-binding for the fitness function to check for. Also the CUDA block size was reduced from 128 to 1.

In the event of a mutated program triggering a run time assert (e.g. by violating an array bound check) or MEMCHECK detecting a memory access violation, testing of the current individual is aborted. Its fitness is set from the answers it returned before the error. The GPU is reset and fitness testing of the next member of the population is started.

2.7 Fitness Function

It is well known that compiling the population in one go is typically more efficient than compiling each member of the population individually [2]. Therefore all 300 GP individuals are converted into a single file which is compiled and linked with the rest of pknotsRG to give a single image which runs the 300 kernels one at a time.

To reduce Genetic Improvement (GI) run time, for fitness testing the evolved code is tested on short RNA strands. Random RNA strings of length 35 typically yield ten non-zero intermediate energy values. To show the GI working with multiple RNA molecules fitness testing processes five in parallel. For simplicity these are all copies of the same string. (In holdout and benchmark testing, multiple RNA base sequences were used.) In the first experiments, the evolved code was tested inside the complete calc_all kernel against answers produced by unmodified code. However in the second, this was short cut and values produced by the GI graft were checked directly (see following and `printf` in Figure 3).

Table 2: GGGP to improve pknotsRG

Representation:
variable list of replacements, deletions and insertions into BNF grammar
Fitness:
Compile modified code. Random RNA string of 35 bases.
Population:
Panmictic, non-elitist, generational. 300 members.
Parameters:
Initial population of random single mutants. 50% truncation selection. 50% two point crossover, 50% mutation. No size limit.

The first experiments had a complex scheme of rewards for running tests without aborting, returning non-zero values and penalties for giving default answers. In the second experiments a simpler fitness function was used: As usual the calc_all kernel is called $n + 1$ times (i.e. 36). Once for each position of the diagonal. After each, the five values (see `gggp_I` etc., in `printf` in Figure 3) of the five evolved variables is compared with its ideal answer. The difference between them is normalised by the maximum correct value of that variable. These are all summed. E.g. if i is 40 but should be 15 then $|40 - 15|/35$ is added to the sum of errors. The goal is to minimise this sum of 36×5 differences.

2.8 Selection

At the end of each generation the kernels are sorted by their fitness and the best half (i.e. the top 150) are selected to be parents of the next generation.

3. RESULTS

The two experiments differed in terms of the BNF grammar and the fitness function but otherwise the same genetic improvement system was used (see Table 2).

In both experiments the GI system was based on evolving lines of code. Although, for simplicity (not shown in Figure 3) each component of the expressions to be evolved was placed on a line by itself. This quick work around to allow easy use of the existing line based scheme unfortunately lead to an opaque system where it was not obvious that it lacked sufficient flexibility and intermediate values to allow the desired functionality to evolve. Comments in Figure 3 give hints about how a multiple tree based system [5] might have been used instead.

Given the improved representation a suitable program (given in Figure 4) was found almost immediately. (Figure 4 shows line 57, `gggp_x` in the assignment of `const int gggp_I`, see Figure 3, is replaced by a copy of line 128, `gggp_y`.) The modified seed was grafted into pknotsRG and was found to give enormous speedups when multiple RNA molecules were processed in parallel (see Figure 6). In almost all cases the new parallel code produces identical answers to the original CPU version and in all cases the difference in energy predictions between the CPU version and the GGGP CUDA version is less than 1%.

3.1 Real RNA Molecules: RNAstrand

Reeder and Giegerich [14] have previously shown that performance of their dynamic programming algorithm on random RNA base sequences is close to that on real RNA se-

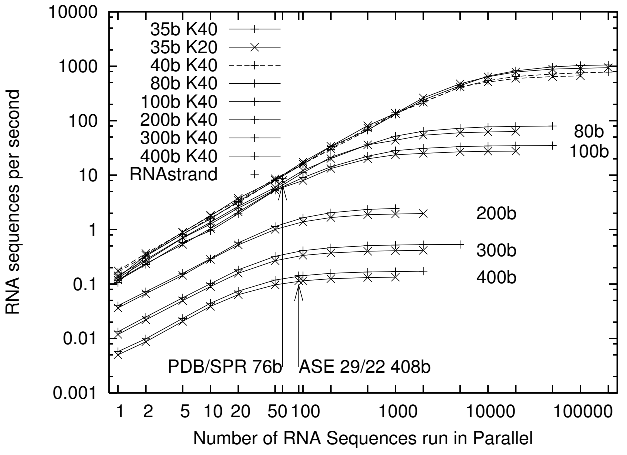

Figure 6: Speed at which new GGGP pknotsRG predicts the binding energy of random RNA molecules on nVidia Tesla K40 (+) and K20 (×) GPUs. Depending on molecule size, the Tesla can process hundreds or thousand of molecules before saturating. The saturation speed was estimated from the performance on the largest RNA molecules to give speed up ratios plotted in Figure 1 and used in Figure 7. The lengths (other than 35 bases) were chosen to allow easy comparison with [14, Table 4] (see also Figure 8). As expected [14], performance on 60 and 90 Biological RNA sequences (76 and 408 bases, arrowed) closely follows that of random sequences.

quences. Nevertheless we down loaded real RNA sequences whose structure has been determined from RNAstrand[1] at the International Institute of Molecular and Cell Biology in Warsaw. We selected two collections of RNA sequences: 1) 90 sequences of 408 bases and 2) 76 sequences of 90 bases and ran pknotsRG on them. In all cases the GGGP GPU code produced identical energy predictions to those given by the original CPU version of pknots. We ran the GGGP version on both Tesla K20 (×) and K40 (+) and got run times similar to those we would expect on random sequences (see Figure 6).

4. DISCUSSION

It is difficult to convert C code to CUDA and have the parallel code both produce the right answers and take advantage of the GPU hardware. The downloaded CUDA code allocated one thread per matrix diagonal element. I.e. on average $n/2$. As we can see from the saturation curves in Figure 6 in excess of 100 000 active threads may be needed to fully load a modern Tesla class GPU. Here the grown and grafted code has enabled all of the existing CUDA code to be used on multiple RNA molecules in parallel and achieved enormous speedups as a result. However the bulk of the CUDA code is as the CPU version. We have not permitted evolution to improve it. We anticipate further gains are possible here but suspect that we have already harvested the "low hanging fruit".

[1]http://iimcb.genesilico.pl/comparna/site_media/entire_datasets/rnastrand.zip 163.3MB down loaded 3 Apr 2015

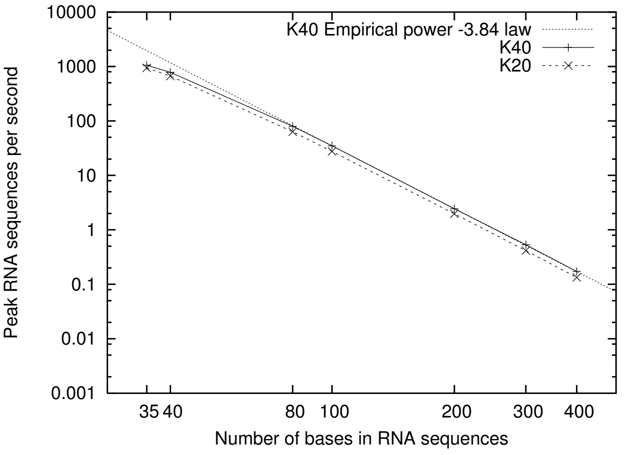

Figure 7: Maximum speed of GGGP pknotsRG on K40 (+) and K20 (×) parallel hardware, empirically scales as $n^{-3.84}$ for both Tesla (where n is the size of the RNA molecules). As expected this is close to $n^{-3.87}$ observed with the original CPU code on a modern 2.67 GHz Intel server.

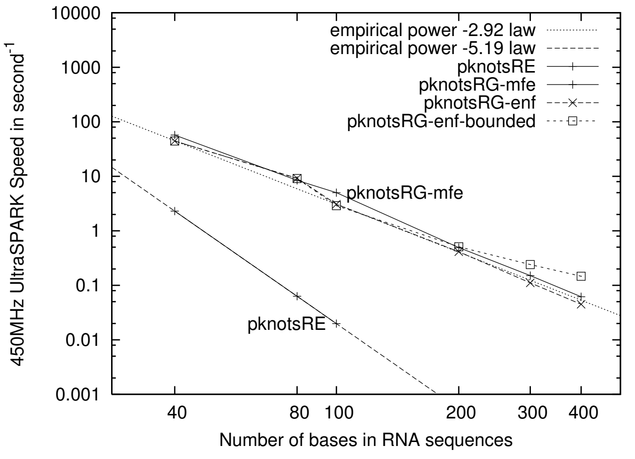

Figure 8: Scaling laws for pknotsRG and earlier pseudo knot RNA folding algorithms. Data from [14, Table 4].

From an artificial intelligence purist point of view the outcome could be taken as disappointing. The seed from which the CUDA code grew and the environment where it was grafted both took human intervention. However that is not the goal of GGGP [1], which seeks to exploit the synergy between human programmers and heuristic optimisation techniques and thereby lift programming up a level of abstraction. GGGP can best be viewed as a stepping stone towards programming as saying (e.g. via a high level description of testing or a fitness function) what needs to be done and letting the computer (compiler) get on and find a program that implements this. In GGGP testing remains key but also it requires the programmer to give hints about components that the automatic procedure might use (here the 12 inputs to the calc_all CUDA kernel) and approximately where to graft in the newly evolved code. (Here the start of calc_all.)

From an engineering point of view a factor of ten thousand fold speed up is not to be sneezed at. From a research view point we need to investigate the differences between the system that evolved but failed to find a solution and the simpler one, with the missing variable `diag` included which gave a general solution almost immediately.

5. CONCLUSIONS

pknotsRG [15] is unusual in real applications in having both CPU and CUDA versions derived from a common stock. With default parameters, in typical short RNA sequences, the performance of the downloaded CUDA version was disappointing. And yet we have shown a small graft of automatically grown code can give a four orders of magnitude speedup without loss of accuracy.

Acknowledgements

Teslas donated by nVidia. Funded by EPSRC grant EP/I033688/1.

6. REFERENCES

[1] Harman, M., Jia, Y., and Langdon, W. B. Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In *Proceedings of the 6th International Symposium, on Search-Based Software Engineering, SSBSE 2014* (Fortaleza, Brazil, 26-29 Aug. 2014), C. Le Goues and S. Yoo, Eds., vol. 8636 of *LNCS*, Springer, pp. 247–252. Winner SSBSE 2014 Challange Track.

[2] Harris, C. *An investigation into the Application of Genetic Programming techniques to Signal Analysis and Feature Detection*. PhD thesis, University College, London, UK, 26 Sept. 1997.

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

[4] Langdon, W. B. Genetically improved software. In *Handbook of Genetic Programming Applications*, A. H. Gandomi, A. H. Alavi, and C. Ryan, Eds. Springer. Forthcoming.

[5] Langdon, W. B. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, vol. 1 of *Genetic Programming*. Kluwer, Boston, 1998.

[6] Langdon, W. B. Performance of genetic programming optimised Bowtie2 on genome comparison and analytic testing (GCAT) benchmarks. *BioData Mining 8*, 1 (8 Jan. 2015).

[7] Langdon, W. B., and Harman, M. Evolving a CUDA kernel from an nVidia template. In *2010 IEEE World Congress on Computational Intelligence* (Barcelona, 18-23 July 2010), P. Sobrevilla, Ed., IEEE, pp. 2376–2383.

[8] Langdon, W. B., and Harman, M. Genetically improved CUDA C++ software. In *17th European Conference on Genetic Programming* (Granada, Spain, 23-25 Apr. 2014), M. Nicolau, K. Krawiec, M. I. Heywood, M. Castelli, P. Garcia-Sanchez, J. J. Merelo, V. M. Rivas Santos, and K. Sim, Eds., vol. 8599 of *LNCS*, Springer, pp. 87–99.

[9] Langdon, W. B., and Harman, M. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation 19*, 1 (Feb. 2015), 118–135.

[10] Langdon, W. B., Lam, B. Y. H., Petke, J., and Harman, M. Improving CUDA DNA analysis software with genetic programming. In *GECCO '15: Proceeding of the Seventeenth annual conference on genetic and evolutionary computation conference* (Madrid, 11-15 July 2015), ACM.

[11] Langdon, W. B., Modat, M., Petke, J., and Harman, M. Improving 3D medical image registration CUDA software with genetic programming. In *GECCO '14: Proceeding of the sixteenth annual conference on genetic and evolutionary computation conference* (Vancouver, BC, Canada, 12-15 July 2014), C. Igel, D. V. Arnold, C. Gagne, E. Popovici, A. Auger, J. Bacardit, D. Brockhoff, S. Cagnoni, K. Deb, B. Doerr, J. Foster, T. Glasmachers, E. Hart, M. I. Heywood, H. Iba, C. Jacob, T. Jansen, Y. Jin, M. Kessentini, J. D. Knowles, W. B. Langdon, P. Larranaga, S. Luke, G. Luque, J. A. W. McCall, M. A. Montes de Oca, A. Motsinger-Reif, Y. S. Ong, M. Palmer, K. E. Parsopoulos, G. Raidl, S. Risi, G. Ruhe, T. Schaul, T. Schmickl, B. Sendhoff, K. O. Stanley, T. Stuetzle, D. Thierens, J. Togelius, C. Witt, and C. Zarges, Eds., ACM, pp. 951–958.

[12] Petke, J., Harman, M., Langdon, W. B., and Weimer, W. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In *17th European Conference on Genetic Programming* (Granada, Spain, 23-25 Apr. 2014), M. Nicolau, K. Krawiec, M. I. Heywood, M. Castelli, P. Garcia-Sanchez, J. J. Merelo, V. M. Rivas Santos, and K. Sim, Eds., vol. 8599 of *LNCS*, Springer, pp. 137–149.

[13] Poli, R., Langdon, W. B., and 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).

[14] Reeder, J., and Giegerich, R. Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics. *BMC Bioinformatics 5*, 1 (2004), 104.

[15] Reeder, J., Steffen, P., and Giegerich, R. pknotsRG: RNA pseudoknot folding including near-optimal structures and sliding windows. *Nucleic Acids Research 35*, suppl 2 (2007), W320–W324.

[16] Steffen, P., and Giegerich, R. Table design in dynamic programming. *Information and Computation 204*, 9 (2006), 1325–1345.