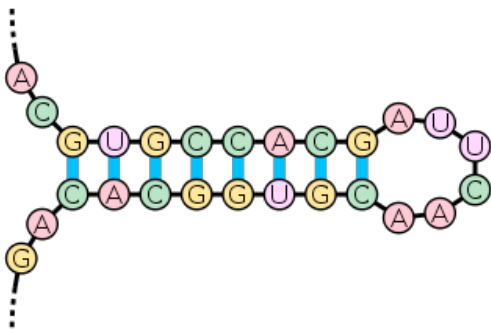




Infernal-GPU: CUDA-Accelerated RNA Alignment

Overview

A non-coding RNA (ncRNA) is a functional RNA molecule that is not translated into a protein. They are a diverse group of genes involved in a variety of cellular processes. Moreover, there are many types of such ncRNAs, which can be broadly classified into families. A particular ncRNA family might be identified on the basis of features that are conserved across a range of organisms. Unlike proteins, which are typically only conserved at the primary sequence level, functional RNAs also conserve secondary structure by means of compensatory mutations that preserve Watson-Crick base pairing. Therefore, it is quite common for structurally homologous ncRNAs to have markedly different primary sequence. As a result, programs that attempt to predict a multiple sequence alignment of ncRNAs must incorporate secondary structure into their model in order to achieve good results. *Infernal* [1], by Sean Eddy et al., is one such program. The Rfam database of RNA families [2] is based on *Infernal* in much the same way that the Pfam database of protein families is based on profile HMM software [3,4].



An example of an RNA stem-loop secondary structure.

Stochastic context-free grammars (SCFGs) are used in the software package *Infernal* to create models of RNA as both primary sequence and base-paired secondary structure. SCFGs are used because hidden Markov models (HMMs) are inadequate to capture the often long-distance pairwise correlations in RNA secondary structure. A particular SCFG architecture called a covariance model (CM) was developed specifically for the RNA similarity search problem [5].

In a covariance model, each single-stranded residue in the query RNA is represented by a state. States are arranged in a tree-like structure that mirrors the secondary structure of the RNA, along with additional states to model insertions and deletions (the consequences of evolution). The standard CM dynamic programming alignment algorithm works by calculating the probability that a substructure of the query rooted at state v aligns to a subsequence $i..j$ in the target sequence. The calculation is recursive, starting at the leaves of the CM (ends of hairpin loops) and subsequences of length 0, and working upward in larger substructures of the CM, and outward in longer and longer subsequences.

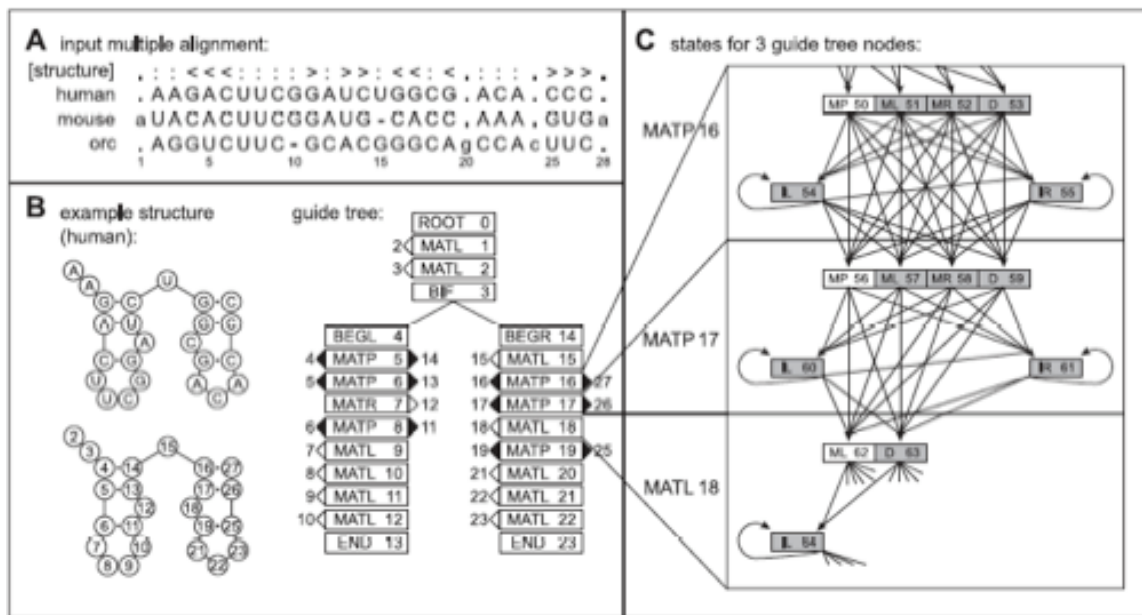


Figure 1. An Example RNA Family and Corresponding CM
 (A) A toy multiple alignment of three RNA sequences, with 28 total columns, 24 of which will be modeled as consensus positions. The [structure] line annotates the consensus secondary structure: angle brackets mark base pairs, colons mark consensus single-stranded positions, and periods mark "insert" columns that will not be considered part of the consensus model because more than half the sequences in these columns contain gaps.
 (B) The structure of one sequence from (A), the same structure with positions numbered according to alignment columns, and the guide tree of nodes corresponding to that structure, with alignment column indices assigned to nodes (for example, node 5, a MATP match-pair node, will model the consensus base pair between columns 4 and 14).
 (C) The state topology of three selected nodes of the CM, for two MATP nodes and one consensus "leftwise" single residue bulge node (MATL, "match-left"). The consensus pair and singlet states (two MPs and one ML) are white, and the insertion/deletion states are gray. State transitions are indicated by arrows.
 doi:10.1371/journal.pcbi.0030056.g001

To construct a CM for a query RNA, one must know (or predict) the secondary structure of the query RNA. An earlier software package from the Eddy lab called COVE [6] included a way to predict secondary structure within a CM framework, but this feature was not included in *Infernal* because more work is needed to make the method generally effective. Hence, *Infernal* requires that the secondary structure be provided (via the SS_cons lines of a stockholm multiple alignment). Another class of programs attempts to predict structure based on a multiple alignment (e.g., RNAalifold [7], CMfinder [8]), and yet another class of programs attempts to simultaneously predict structure and alignment (e.g., Stemloc [9], CONSAN [10]).

The most noteworthy limitation of SCFGs is their computational complexity. SCFG-based RNA analysis algorithms require time and memory proportional to at least L^3 (where L is the sequence length), because every possible pair of residues (L^2) must be tried against up to $L/2$ base-pairing states in the model (and in most RNA SCFGs, the time required more typically scales as L^4). The latest version of *Infernal* incorporates some heuristics [11] to ameliorate the situation, but the computational cost can still be considerable. Moreover, if the program were made to run faster, one may be able to reduce the use of heuristics in order to get more optimal results. This was my motivation for proposing to use CUDA to accelerate the programs that constitute *Infernal*. The following section describes my work towards this goal.

Implementation and Results

Infernal is a large and complex software package. After reading through the documentation and completing the tutorial (~100 pages), I had a good idea of the programs that would benefit most from speedup - namely, *cmcalibrate* and *cmsearch*. Both programs use the 'Inside' algorithm, which is nearly identical to the Cocke-Younger-Kasami (CYK) database search algorithm for CMs. The banded CYK version of the algorithm used by *Infernal* is as follows:

```

Initialization (impose bands): for  $j = 0$  to  $L$ ,  $v = M - 1$  down to  $0$ :
  for  $d = 0$  to  $(\text{dmin}(v) - 1)$ ,  $\alpha_v(j,d) = -\infty$ ;
  for  $d = \min((\text{dmax}(v) + 1), (j + 1))$  to  $W$   $\alpha_v(j,d) = -\infty$ ;

Initialization at  $d = 0$ : for  $j = 0$  to  $L$ ,  $v = M - 1$  down to  $0$ :
 $v = \text{end state } (E)$ :  $\alpha_v(j,0) = 0$ 
 $v = \text{bifurcation } (B)$ :  $\alpha_v(j,0) = \alpha_y(j,0) + \alpha_z(j,0)$ ;
 $v = \text{delete or start } (D,S)$ :  $\alpha_v(j,0) = \max_{y \in C_v} [\alpha_y(j,0) + \log t_v(y)]$ ;
else ( $v = P, L, R$ ):  $\alpha_v(j,0) = -\infty$ .

Recursion: for  $j = 1$  to  $L$ ,  $d = \max(1, \text{dmin}(v))$  to  $\min(\text{dmax}(v), j)$ ,  $v = M - 1$  down to  $0$ 
 $v = \text{end state } (E)$ :  $\alpha_v(j,d) = -\infty$ ;
 $v = \text{bifurcation } (B)$ :  $k_{\min} = \max(\text{dmin}(z), (d - \text{dmax}(y)))$ ,
 $k_{\max} = \min(\text{dmax}(z), (d - \text{dmin}(y)))$ ,
 $\alpha_v(j,d) = \max_{k_{\min} \leq k \leq k_{\max}} [\alpha_y(j - k, d - k) + \alpha_z(j, k)]$ ;
 $v = \text{delete or start } (D,S)$ :  $\alpha_v(j,d) = \max_{y \in C_v} [\alpha_y(j,d) + \log t_v(y)]$ ;
else ( $v = P, L, R$ ):  $\alpha_v(j,d) = \max_{y \in C_v} [\alpha_y(j - \Delta_v^R, d - (\Delta_v^L + \Delta_v^R)) + \log t_v(y)] + \log e_v(x_i, x_j)$ .

```

Using the GNU profiler (*gprof*), it was easy to determine that the most expensive function was the call to `FastIInsideScan`, ~25% of the overall runtime. This function in turn calls `ILogsum`, ~22% of the overall runtime. Therefore, this was the logical place to focus my efforts to speed up the program. `FastIInsideScan` is an optimized version of the Inside algorithm, and as such is broken up such that more commonly encountered cases are tested for first in various `case` statements. I identified 13 blocks of calls to `ILogsum`, and profiled the application to see which blocks were most expensive. The results are presented in the following table:

Self Seconds	Number of Calls	Block Number
76.71	21726867600	ILOGSUM09
44.75	4699345716	ILOGSUM08
23.77	18621678828	ILOGSUM04
22.11	12792346590	ILOGSUM03
21.61	18283754300	ILOGSUM06
12.00	10277078672	ILOGSUM10
10.07	3683305096	ILOGSUM05
7.97	1035019000	ILOGSUM07
5.80	5606151190	ILOGSUM11
5.14	349980292	ILOGSUM02
1.31	1081341872	ILOGSUM12
0.77	469160812	ILOGSUM01
0.29	264335068	ILOGSUM13

Based on this analysis, I decided to parallelize loop #9, which contained several `ILogsum` calls. If this were successful, I thought, I could move on to the other loops. Empirically, I determined that on average (for my test input, at least) this loop contained approximately 40-80 iterations, and since there were no dependencies between iterations, I could assign each one to a thread on the GPU. Next, I had to determine which data would be input to the kernel, and which data would be copied back as output. Aside from various simple integer offsets and counters, I determined that four arrays had to be copied to the GPU: `alpha`, `itsc`, `init_scAA`, and `ilogsum_lookup`. The output array would be called `sc_v`. The CUDA memory allocation and copying functions require knowing exactly how large each array is, so it took some tracing through the code to determine this. In addition, some of the arrays on the host were multidimensional, so the best way to store this on the GPU was unclear. After some research, it seemed simplest to map everything to 1-dimensional GPU arrays, so this is what I did. Computing indexes into these 1-D GPU arrays had to be done carefully, especially when the host array was 3-D. In my first naïve implementation, I copied the input arrays to the GPU with each kernel invocation, knowing full well that this was inefficient.

Testing was done to time the program and to make sure it produced correct results. All of my runs were done using a reduced version of an included sequence database. The actual `cmsearch` invocation was as follows:

```
cmsearch --fil-no-hmm --fil-no-qdb my.c.cm tosearch.chopped.db
```

The original program takes ~20 seconds to run and produces the following results:

>example

Plus strand results:

Query = 1 - 72, Target = 101 - 173
Score = 78.06, E = 1.873e-22, P = 2.906e-26, GC = 53

```
((((((((, , <<<< ____ . ____ >>>>, <<<<< _____ >>>>>, , , , , <<<<< _____
1
gCcgacAUaGcgcAgU.GGuAgcgCgccagccUgucAagcuggAGgUCCgggGUUCGAUu 59
GC:+A::UAGC:CAGU GG
AG:GCGCCAG:CUG+++A:CUGGAGGUCC:G:GUUCGAU
101
GCGGAUUUAGCUCAGUuGGGAGAGCGCCAGACUGAAGAUCUGGAGGUCCUGUGUUCGAUC 160

>>>>>))))))):
60 CcccGUgucgGca 72
C:C:G::U+:GCA
161 CACAGAAUUCGCA 173
```

Minus strand results:

Query = 1 - 72, Target = 15124 - 15083
Score = 12.93, E = 0.02895, P = 4.491e-06, GC = 48

```
((((((((, , ~~~~~, , , , , <<<<< _____ >>>>>))))))):
1 gCcgacAUa*[33]*AGgUCCgggGUUCGAUuCcccGUgucgGca 72
GCC: :AUA A G CC::GG UCG UCC::GU: :GGC+
15124 GCCAGUAUA*[ 5]*AUGCCCUAGGAUCG--UCCUAGUAAUGGCG 15083
```

For all of my testing, these are the results I refer to as “correct”. My first GPU implementation took ~20 minutes, or was ~60x slower than the original program. I determined that it should be safe to pre-load the input data on the GPU once, and avoid copying it before each kernel invocation. I was able to do this for all but one alpha row (arow2; still not sure why this gave problems), and doing this reduced the GPU version runtime to ~3 minutes, or ~9x slower. The basic reason for the slower time, even after eliminating wasteful memory transfers, is because there is not enough work buffered to make the overhead of each kernel invocation worth it. I came to this conclusion by using the CUDA profiler, and inspecting the `cputime` and `gputime` values associated with each kernel invocation. `cputime` includes `gputime`, so the difference between the two is essentially the overhead of each kernel invocation. In my case, the overhead was ~4x greater than the `gputime` itself, whereas one would hope it would be just a small fraction of it. Therefore, in order for my strategy to be effective, I would have to buffer more work for each kernel invocation, thereby reducing the total number of kernel calls. However, I saw no way to do this without restructuring the function significantly, which didn’t seem feasible given my limited understanding of the implementation.

Eric, one of the `Infernal` developers, suggested that I take a look at `RefIInsideScan` - a reference implementation of the Inside algorithm, and one that would be simpler to work with. The suggestion was a good idea, although the implementation was not fundamentally different. However, it did allow me to see things clearly enough to develop an approach to parallelization at the level of what I call the v-loop (the loop over CM states). This loop was nested inside the j-loop, or the loop over positions in the database sequence. In my test case, the v-loop was 229 iterations, and the j-loop was over 17000 iterations. Therefore, my approach was to assign each v-loop iteration to a separate GPU thread in a kernel that would be invoked ~ 17000 times per function call (`RefIInsideScan` was called 3 times during the program execution, the first time being a short test to extrapolate a final runtime; in that case, the j-loop was only 400 iterations). The total number of kernel invocations, roughly 35,000, is far fewer than the almost 22 billion kernel invocations in the `FastIInsideScan` parallelization.

The work I had to do for this part was similar to the `FastIInsideScan` parallelization, but much more involved. The following is a list of arrays that had to be copied to the GPU before invoking the v-loop kernel. (Naturally, I wanted to do this *outside* the j-loop, so that the copies were made only once per function call; this required redefining how and where some of these arrays were initialized.)

Array	Height (2-D)	Width (1-D)	Depth (3-D)
dnAA	W+1	cm->M	
dxAA	W+1	cm->M	
jp_wA	L	W+1	
cm->sttype		cm->M	
esc_vAA	cm->M	cm->abc->Kp+1	
cm->itsc	cm->M	6	
cm->stid		cm->M	
cm->cnum		cm->M	
dsq		j0+1	
cm->cfirst		cm->M	
dmin		cm->M	
dmax		cm->M	
init_scAA	cm->M	W+1	
alpha	cm->M	W+1	2
alpha_begl	cm->M	W+1	W+1
vsc		cm->M	
ilogsum_lookup		LOGSUM_TBL	

Stack variables passed to the kernel included the length of the v-loop (`cm->M-1`), `j`, `W`, `cur`, `prv`, `do_banded`, `cm->abc->Kp`, `dnA_dxA_offset`, and `jp_wA_offset`. Additionally, GPU versions (`__device__`) of the following functions had to be made: `Emitmode`, `StateRightDelta`, `StateDelta`, `ILogsum`, and `Scorify`. The arrays that had to be copied *out* of the GPU were `alpha`, `alpha_beg1`, and `vsc`. Ideally, these copies would take place outside of the j-loop, which would essentially require running the entire contents of the j-loop on the GPU (not yet implemented). It took a very long time to get this implemented with any degree of confidence – then to remove compiler/linker errors – and then to work through the runtime errors (segmentation faults or the GPU equivalent thereof). During this I found at least one error in the program’s source-level documentation; I believe the array bounds described for `dnAA` and `dxAA` have the height and width backwards, relative to the implementation.

There are currently two problems with the v-loop parallelization. For one, it is not correct – the program output does not match the original implementation, which implies there are bugs left to be worked out. Two, the program is still ~7x slower than the original reference implementation. This time, I believe the problem is primarily large numbers of incoherent loads/stores, which is easily demonstrated with CUDA profiling turned on. Therefore, additional effort would be needed to optimize the implementation in the standard ways: coalesce reads/writes to GPU global memory, use GPU shared memory where appropriate, eliminate bank conflicts, and so on. I’m only mildly optimistic about the work/reward ratio with regard to undertaking this, given my experience thus far.

Conclusion

This project could really be divided into two phases. During the first phase, I was reading all the literature I could in this area, as well as relevant book chapters in an effort to understand the biological, statistical, and algorithmic underpinnings of the methods that were being employed. In my view, the area is interesting from all of these perspectives – an immense variety of biological richness to model, and a corresponding variety of ways to do so algorithmically and mathematically. Dynamic programming algorithms rely on optimal substructure, which can make them challenging to parallelize due to their recursive nature.

The second phase of the project was very nitty-gritty, and involved a great deal of uncertainty – it isn’t easy to take complex scientific code others have written and understand it well enough to restructure it, if that becomes necessary. That said, it did look like I was operating in the right place, and perhaps I was closer to success than the numbers indicate. Perhaps someone will continue this work, and then we’ll know. All told, it was quite an interesting way to be introduced to an exciting area of bioinformatics, and I’m thankful for having had the opportunity through this class.

References

1. Infernal, <http://infernal.janelia.org/>.
2. Griffiths-Jones S, Moxon S, Marshall M, Khanna A, Eddy SR, et al. (2005). Rfam: Annotating non-coding RNAs in complete genomes. *Nucleic Acids Res* 33: D121–D141.
3. Sonnhammer ELL, Eddy SR, Birney E, Bateman A, Durbin R (1998). Pfam: Multiple sequence alignments and HMM-profiles of protein domains. *Nucleic Acids Res* 26: 320–322.
4. Finn RD, Mistry J, Schuster-Bockler B, Griffiths-Jones S, Hollich V, et al. (2006). Pfam: Clans, web tools and services. *Nucleic Acids Res* 34: D247–D251.
5. Eddy SR, Durbin R (1994). RNA sequence analysis using covariance models. *Nucleic Acids Res* 22: 2079–2088.
6. COVE, <ftp://selab.janelia.org/pub/software/cove/>.
7. RNAalifold, <http://rna.tbi.univie.ac.at/cgi-bin/RNAalifold.cgi>.
8. CMfinder, <http://bio.cs.washington.edu/yzizhen/CMfinder/>.
9. Stemloc, <http://biowiki.org/StemLoc>.
10. CONSAN, <http://selab.janelia.org/software/consan/>.
11. EP Nawrocki and SR Eddy. Query-dependent banding (QDB) for faster RNA similarity searches. *PLoS Computational Biology*, 3:e56 (2007).