GPU Accelerated RNA Folding Algorithm

Guillaume Rizk[1] and Dominique Lavenier[2]

[1] Univ-Rennes 1/IRISA
[2] ENS-Cachan/IRISA
IRISA - Symbiose Campus universitaire de Beaulieu,
35042 Rennes Cedex, France
{guillaume.rizk,dominique.lavenier}@irisa.fr

Abstract. Many bioinformatics studies require the analysis of RNA or DNA structures. More specifically, extensive work is done to elaborate efficient algorithms able to predict the 2-D folding structures of RNA or DNA sequences. However, the high computational complexity of the algorithms, combined with the rapid increase of genomic data, triggers the need of faster methods. Current approaches focus on parallelizing these algorithms on multiprocessor systems or on clusters, yielding to good performance but at a relatively high cost. Here, we explore the use of computer graphics hardware to speed up these algorithms which, theoretically, provide both high performance and low cost. We use the CUDA programming language to harness the power of NVIDIA graphic cards for general computation with a C-like environment. Performances on recent graphic cards achieve a $\times 17$ speed-up.

Keywords: GPGPU, RNA, secondary structure, minimum free energy.

1 Introduction

The computation of secondary structural folding of RNA or single-stranded DNA is a key element in many bioinformatics studies and, as such, has been extensively studied for many years. The firsts to propose an algorithm to predict the folding structure of RNA or DNA sequences were Waterman, Smith and Nussinov et al. [1,2]. This algorithm was based on dynamic programming with a complexity of $\mathcal{O}(n^3)$, yet their approach had several issues.

Following this pioneer work, several improvements have been done leading to different kinds of dynamic programming algorithms. We can cite: (1) the computation of the most stable structure through energy minimization running in $\mathcal{O}(n^3)$, introduced by Zuker and Stiegler [3] which outputs a single optimal structure and its corresponding energy ; (2) the computation of a partition function over all possible structures for deriving additional properties of the thermodynamic ensemble such as the base pairing probabilities of any base pair, introduced by McCaskill [4] ; (3) the computation of suboptimal structures [5] which generates all structures within a given energy range of the optimal one. Implementations of those algorithms are found in two major packages, ViennaRNA and Unafold [6,7].

Despite many huge efforts to reduce the algorithmic computational complexity, execution times are steadily increasing due to the fast growing of genomic databases and, for the last years, the relative stagnation of the microprocessor frequencies. One solution is to use multi-core systems or clusters, which can yield good performance but at a high cost. Another approach is the use of computer graphics hardware, which possibly exhibits a higher performance/cost ratio than clusters.

Indeed, the raw power of graphics processing unit (GPU) has a faster increase rate than traditional microprocessors. Moreover, recent improvements in the programmability of GPUs have opened the way to new applications from which GPUs were not initially designed for. General purpose computation on GPU (GPGPU) is now a field of research investigated in many domains requiring high performances. Among many others, successful applications include bioinformatics with the Smith-Waterman sequence alignment [8,9].

In this paper, we investigate how GPUs can be used to accelerate the computation of the minimum free energy of RNA or DNA sequence folding. We use the implementation of the Unafold package given in the function *hybrid-ss-min* [7]. This function is intensively used in different programs of the Unafold package and represents the most time consuming part. We show that adding a graphical board can speed-up the whole program by a factor $\times 17$ compared to a sequential execution on a one-core microprocessor.

Although the RNA folding algorithm studied uses dynamic programming just like the Smith Waterman algorithm, they should not be confused. Both algorithms are very different, thus previous GPU implementations of the Smith Waterman algorithm [8,9] did not prefigure the feasibility of an efficient GPU implementation here. On the contrary, its complexity in terms of memory access patterns and parallelization issues makes it a real challenge.

The rest of paper is organized as follows: In Section 2, we introduce the folding algorithm. In section 3, the GPU implementation of the folding algorithm is explained. Finally, section 4 gives the performance results obtained on different platforms.

2 Folding Algorithm

This section briefly exposes the principles of the folding algorithm as implemented in the Unafold package in the function *hybrid-ss-min* [7].

2.1 RNA Structure

RNA or Ribonucleic acid is a chain of nucleotide units. There are four different nucleotides, also called *bases*: adenine (A), cytosine (C), guanine (G) and uracil (U). Two nucleotides can form a bond thus forming a *base pair*, according to the Watson-Crick complementarity: A with U, G with C; but also the less stable combination G with U, called wobble base-pair. All the base pairs of a sequence force the nucleotide chain to fold into a system of different recognizable domains like hairpin loops, bulges, interior loops or stacked regions. This is called the *secondary structure* of the sequence. The different loop types are introduced in Fig. 1. The secondary structure can also form complex patterns like *pseudoknots* which consist of two base pairs $i \cdot j$ and $k \cdot l$ that do not verify the nesting property $i < j < k < l$. The secondary structure is often determinant in the functional role of the RNA molecule.

2.2 Energy Model

The algorithm is designed to find the most stable structure of a RNA sequence. It is used in many bioinformatics pipelines such as the search of micro RNAs where the stability of the secondary structure is an important feature.

A secondary structure is described by a list of base pairs $i \cdot j$ where each base forms at most one pair. The algorithm is based on a decomposition of the secondary structure into its constituent loops. Each loop is associated with an experimentally measured energy according to its sequence, length and type. The stability (free energy) of a structure is the sum of the energies of all its loops.

In the dot bracket representation given in Fig. 1, an unpaired base is depicted by a dot, and a pair by a matching pair of parenthesis. In the model used, matching pairs of parenthesis have to be well nested, i.e there are no pseudoknots. This restriction is a requirement to allow a relatively fast dynamic programming approach as the one developed by Zuker and Stiegler. Indeed, it ensures that the secondary structure of each subsequence i, j can be computed independently from the rest of the sequence, a required feature for dynamic programming.

2.3 Algorithm

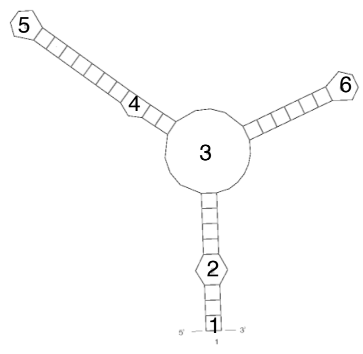
The dynamic programming algorithm uses three tables: $Q'_{i,j}$ is the minimum energy of folding of a subsequence i, j given that bases i and j form a base pair; $Q_{i,j}$ and $QM_{i,j}$ are the minimum energy of folding of the subsequence i, j assuming that this subsequence is inside a multiloop and that it contains respectively at least one and two base pairs. A simplified model of the recursion relations can be written as:

$$Q'_{i,j} = \begin{cases} \min \begin{Bmatrix} Eh(i,j) \\ Es(i,j) + Q'_{i+1,j-1} \\ \min\limits_{k,l \in]i;j[^2} Ei(i,j,k,l) + Q'_{k,l} \\ QM_{i+1,j-1} \end{Bmatrix} & \textit{if pair } i \cdot j \textit{ is allowed} \\ \infty & \textit{if pair } i \cdot j \textit{ is not allowed} \end{cases} \quad (1)$$

$$QM_{i,j} = \min_{i<k<j} (Q_{i,k} + Q_{k+1,j}) \quad (2)$$

$$Q_{i,j} = \min \left\{ QM_{i,j}, \min(Q_{i+1,j}, Q_{i,j-1}), Q'_{i,j} \right\} \quad (3)$$

$Eh(i,j)$ $Ei(i,j,k,l)$ and $Es(i,j)$ are respectively the energies of:



```
AAAAAAGGGAAAAGAACAAAGGAGACUCUUCUCCUUUUUCAAAGGAAGAGGAGACUCUUUCAAAAAUCCCUCUUUU
((((.(((((...(((.((((((((....)))))))))))...(((((((....))))))).....))))).)))) (-24.5)
```

Fig. 1. Secondary structure. The secondary structure begins in 1 with stacked base pairs (two closing base pairs with both sides of the loop of length zero). 2 is an interior loop (two closing base pairs with both sides non null). 3 shows a multiloop (several closing base pairs). 4 is a bulge loop (two closing base pairs with one loop side of length zero and the other greater than zero. 5 and 6 are hairpin loops (one closing base pair). The structure can also be written in a dot bracket representation where an unpaired base is a dot and a base pair is a matching pair of parenthesis. The free energy of the structure (−24.5) is the sum of the energies of its constituent loops.

– $Eh(i,j)$: a hairpin loop closed by the pair $i \cdot j$.
– $Ei(i,j,k,l)$: an interior loop formed by the two base pairs $i \cdot j$, $k \cdot l$.
– $Es(i,j)$: two stacked base pairs $i \cdot j$ and $(i+1) \cdot (j-1)$.

These functions compute energies through the use of lookup tables containing energy parameters according to the size and sequence of the loop.

E_j being the minimum free energy of subsequence $1 \dots j$, the minimum free energy E_n of the whole sequence is then obtained through the recursion:

$$E_j = \min \left\{ E_{j-1}, \min_{1<k<j} (E_{k-1} + Q'_{k,j}) \right\} \tag{4}$$

Dynamic programming using this recursion computes the minimum free energy of a sequence of length n in $\mathcal{O}(n^2 \cdot L^2 + n^3)$ by restricting the loop size of interior loops to L. The corresponding secondary structure is then obtained by a trace-back procedure.

3 GPU Implementation

3.1 Architecture and Programming

GPUs are massively parallel architectures providing cheap high performance computing. We choose in our work CUDA as it combines high performance with the ease of use of a C-like environment [10]. The latest NVIDIA GPU, the GT200, is divided into 30 multiprocessors each being a SIMD unit of 8 32-bit processors. A GPU procedure is a *kernel* called on a set of threads, divided in a *grid* of *blocks* each running on a single multiprocessor. Furthermore *Blocks* are divided in warps of 32 threads that must execute the same instruction simultaneously. Thus, *branching* (if-then-else control flow instructions) does not impact performance as long as each thread within a warp take the same code path. Moreover, only threads within a block can be synchronized and can share the fast on-chip shared memory. One key difference with a traditional CPU implementation is that the programmer has to explicitly handle several memory spaces of different performance, size, scope and lifetime: global, texture, constant and shared memory as well as registers.

3.2 Parallelization Scheme

Algorithm 1 shows the main loops of the computation along with the several ways to expose parallelism. We chose a mixed approach: we compute the minimum free energy of folding of several sequences in parallel, each one being itself parallelized. According to this parallel scheme, we can provide the GPU with many independent tasks together with a low memory consumption. The number of sequences being computed simultaneously is adapted according to their length: one large sequence can provide enough independent tasks to the GPU whereas small ones have to be computed by groups. We also implemented a multi-GPU algorithm by dividing work among GPUs at the coarse-grained level, each GPU computes a different group of sequences.

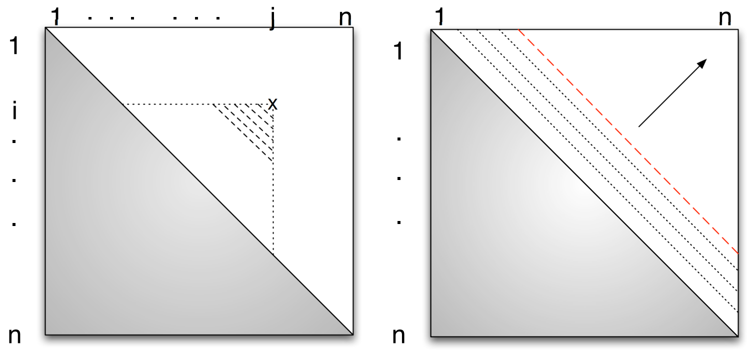


Fig. 2. Left: Data dependency relationship. Each cell of the matrix contains the three values Q', QM and Q. As subsequence i, j is the same as subsequence j, i only the upper half of the matrix is needed . The computation of cell i, j needs the lower left dashed triangle and the two vertical and horizontal dotted lines. **Right: Parallelization.** According to the data dependencies, all cells along a diagonal can be computed in parallel from all previous diagonals.

Figure 2 shows the data dependencies coming from the recursion (1) to (3). They imply that, given all previous diagonals, all cells of a diagonal can be processed independently. Three kernels are designed for the computation of $Q'_{i,j}$, $QM_{i,j}$ and $Q_{i,j}$, according to equations (1) to (3). Each one computes one diagonal of several sequences. The whole matrix is then processed sequentially through a loop over all diagonals. The next step corresponding to equation (4) is a combination of reductions (search of the minimum of an array) which is parallelized in another kernel. The final step, the traceback procedure for computing the secondary structure, is currently left on the CPU as its execution time is far lower.

3.3 Optimization Key Points

Memory accesses are the bottleneck of the implementation. Here, the data are divided into three groups: the base sequence, the three tables Q', QM,Q and the energy parameters needed for the computation of loop energies. Maximum performances are obtained when available memory resources are used to their maximum and when the best match between the different memory spaces and type of data are found. Here, the texture memory is used for the sequence and parts of the tables which both show some spatial locality in their access pattern, as for the computation of one cell $QM_{i,j}$ where equation (2) shows that accesses to all elements of a line and column of matrix Q have to be made. For energy parameters, the best choice is the constant memory. However, its small size compels us to also employ the global memory for the least used ones. Lastly, the shared memory is kept for storage of intermediate results in the computation.

Another important issue of the implementation comes from equation (1) which shows that the computation of table Q' is not the same for all cells: if the pair $i \cdot j$ is forbidden then cell $Q'_{i,j}$ is set to ∞. This hurts the SIMD model of GPU which, as stated section 3.1, says that in order to get full performance all threads of a warp must execute the same instruction path. To solve this issue our implementation computes on CPU an index of all the cell positions that have their base pairs allowed, which is then handed to the GPU. This increases the amount of data transferred between the CPU and the GPU but decreases *branching* in GPU kernels. Moreover CPU computation can be overlapped with GPU computation thus allowing us to better use all available resources.

We found that for maximum efficiency the parallelization has to be done up to the the finest grain achievable, to ensure the GPU reaches its maximum potential while using the less memory possible. Different levels of parallelization are exploited: parallelization across several sequences, across several cells of a diagonal, and across tasks required for the computation of a single cell itself: the search of a minimum is parallelized on several threads of a same block sharing intermediate results through shared memory.

Algorithm 1. Main function and parallelizable loops

```
1: Input: $N$ sequences of length $L$
2: Output: minimal energy of the $N$ sequences
3: Coarse-grained level: parallelization over multiple sequences
4: for sequence $s$ in $[1; N]$ do
5:   for diagonal $d$ in $[1; L]$ do
6:     Medium-grained level: parallelization over multiple cells of a diagonal
7:     for i in $[1; L - d]$ do
8:       $j \leftarrow i + d$
9:       Fine-grained level: parallelization over the minimization computation
10:      compute $Q'(i, j, s)$, $QM(i, j, s)$, $Q(i, j, s)$
11:    end for
12:  end for
13:  compute $E_L(s)$
14: end for
```

4 Results

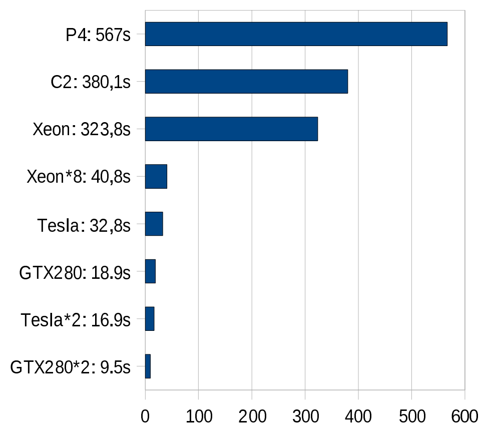
GPU and CPU implementations are both compared on different graphic cards and processors. The main testing platform is an octo-core Xeon E5430 2.66Ghz ($4 \times$ 6MB L2 cache) with 8GB RAM and two NVIDIA Tesla C870 cards, each having 16 multiprocessors. We also test older processors, a Pentium 4 3Ghz (1MB L2 cache), a Core2 6700 2.66GHz (4MB L2 cache), and the latest high-end graphic card the NVIDIA GTX280 with 30 multiprocessors.

4.1 Analysis on 120 Bases-Long Sequences

Problem specifications. A typical use of the algorithm is the computation of the secondary structures of many small RNA sequences. The search of micro RNAs in a whole genome requires, for example, to know the secondary structure of millions of sequences of length approximately 120 [11]. Therefore we first choose to test the algorithm on sequences of this length, here with 40000 randomly generated sequences.

Figure 3 reports running times in seconds and the corresponding speedup achieved by different combination of cards versus one or eight CPU cores. Our CPU multi-core implementation is done on a coarse-grained level by parallelizing the work over multiple sequences, corresponding to line 4 of algorithm 1.

Results. We achieve a speed-up of about $\times 10$ for one Tesla card versus one core of a Xeon. An interesting point is that although the algorithm was originally developed with the Tesla, it scales well with the latest graphic card. The GTX280 is 70% faster than the Tesla with a speed-up of $\times 17$ versus one core of a Xeon, which roughly corresponds to the increase of memory bandwidth between the two cards. With the two Tesla, the speed-up becomes $\times 19$, and two GTX280 get $\times 33.1$, which shows that the processing power of cards adds up well when used together.



	Xeon	Xeon x8
Tesla C870	9.9	1.2
GTX 280	17.1	2.1
Tesla C870 x2	19.2	2.4
GTX 280 x2	33.1	4.2

Fig. 3. Left: Execution Time. Time spent in seconds for the computation of the minimum free energy of 40000 randomly generated sequences of length 120, energy only (option -E of the *hybrid-ss-min* function). Processors are: P4 a pentium4 3.0Ghz (1MB cache), C2 is one core of a core2 2.66 Ghz (4MB cache), Xeon and Xeon*8 are respectively one and eight cores of Xeon 2.66Ghz (6MB cache). Graphic cards are NVIDIA Tesla C870, GTX280, bi-Tesla C870, and bi-GTX280. **Right: Corresponding speed-up.** Acceleration ratio of graphic cards versus Xeon processor, one core or octo-core configuration.

Accuracy. Our GPU implementation uses exactly the same algorithms and thermodynamic rules as Unafold, thus the results and accuracy obtained on GPU is exactly the same as the standard CPU Unafold function.

Performance / cost analysis. When using a parallelized version of the algorithm on the eight CPU cores, speed-ups are much less (Figure 3), yet the performance/cost ratio is clearly in favor of the GPU implementation. Indeed our results show that a system with two GTX280, easily doable for 2500 euros, would be roughly equivalent to four octo-core computers costing a total of more than 8000 euros.

As for standard computers at everyone disposal, the advantage of GPUs is also obvious: considering every systems are now dual-cores, adding a GTX280 would allow to get at least $\times 8$ performance even if both CPU cores are used, at a cost of about 400 euros.

4.2 Analysis Across Varying Sequence Lengths

The algorithm is then experimented upon with various sequence lengths. Speed-up of Tesla and GTX280 versus one Xeon processor core are showed in figure 4. It should first be noted that the GTX 280 is always at least 50% faster than the Tesla except for very long sequences, where it begins to lack memory (Tesla has 1.5 GB whereas GTX 280 has 1.0 GB). We see that performance is good for short

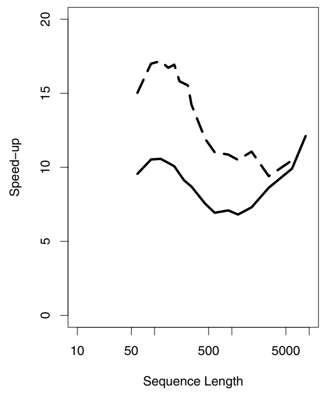


Fig. 4. Speed up comparison. Speed up of Tesla C870 and GTX 280 graphic card versus one core of a 2.66 Ghz Xeon for randomly generated sequences of different lengths. Solid line is Tesla C870, dashed line is GTX 280.

sequences (Tesla gets $\times10$ speed-up), then it comes to a minimum for 1000 bases long sequences (Tesla gets $\times7$) and it rises again for very long sequences ($\times12$ for Tesla with sequence of length 9000). This comes from the fact that different portions of the code do not have the same computational complexity and GPU efficiency. With n the length of a sequence, QM computation is in $\mathcal{O}(n^3)$ whereas Q' computation is in $\mathcal{O}(n^2)$. The efficiency of the $\mathcal{O}(n^2)$ part decreases when n increases due to different memory access patterns, which explains the decrease in performance. The $\mathcal{O}(n^3)$ part of the algorithm is always very efficient on GPU but only becomes a preponderant part of the algorithm for long sequences, which explains the overall speed up increase we observe for long sequences.

4.3 Comparison Against GTfold

A.Mathuriya *et al.* implemented a CPU multicore algorithm for RNA secondary structure prediction which uses what we call in algorithm 1 the medium-grained level [12]. They compute in their study the folding of the HIV-1 sequence and a set of 11 Picornaviral sequences on a 32-core IBM P5-570 server. Table 1 compares the running time they obtain against our GPU implementation on one Tesla C870 card. It shows that an expensive 32-core server only gets $\times1.6$ the performance of a single GPU.

Table 1. Running times on HIV-1 sequence (9781 nucleotides) and a set of 11 Picornaviral sequences (7124 to 8214 nucleotides), cf [12] for sequence accession numbers

	GTfold 32-core IBM P5-570	GPU Tesla C870	Unafold 1 core Xeon
HIV-1	84 s	133 s	1876 s
11 Picornavirus	480 s	765 s	7902 s

5 Future Work

This work is the first step in parallelizing RNA folding algorithm on GPU. It shows that GPUs can deliver significant speed-ups even on algorithms with complex memory access patterns.

However, although GPUs recently became easier to use, an efficient GPU implementation remains a lengthy process. For years programmers have developed purely sequential algorithms, yet it appears that future systems will become more and more highly parallel architectures. Thus, a future challenge will be to find a way to facilitate implementation of algorithms for a parallel execution; on multi-core chips using the MIMD paradigm, on GPUs using the SIMD paradigm, and the trickiest task, on a combination of both.

References

1. Waterman, M., Smith, T.: RNA secondary structure: a complete mathematical analysis. Math. Biosci. 42, 257–266 (1978)
2. Nussinov, R., Pieczenik, G., Griggs, J., Kleitman, D.: Algorithms for loop matchings. SIAM J. Appl. Math. 35(1), 68–82 (1978)
3. Zuker, M., Stiegler, P.: Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. Nucleic Acids Res. 9(1), 133–148 (1981)
4. McCaskill, J.: The equilibrium partition function and base pair binding probabilities for RNA secondary structure. Biopolymers 29(6-7), 1105–1119 (1990)
5. Wuchty, S., Fontana, W., Hofacker, I.L., Schuster, P.: Complete suboptimal folding of RNA and the stability of secondary structures. Biopolymers 49, 145–165 (1999)
6. Hofacker, I.L., Fontana, W., Stadler, P.F., Bonhoeffer, L.S., Tacker, M., Schuster, P.: Fast folding and comparison of RNA secondary structures. Monatsh. Chem. 125, 167–188 (1994)
7. Markham, N., Zuker, M.: DINAMelt web server for nucleic acid melting prediction. Nucleic Acids Research 33, W577–W581 (2005)
8. Liu, W., Schmidt, B., Voss, G., Schroeder, A., Muller-Wittig, W.: Bio-sequence database scanning on gpu. In: Proceeding of the 20th IEEE International Parallel & Distributed Processing Symposium: 2006(IPDSP 2006) (HICOMB Workshop) (2006)
9. Svetlin, M., Giorgio, V.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 9 (2008)
10. Nvidia cuda, `http://developer.NVidia.com/object/cuda.html`
11. Stark, A., Kheradpour, P., Parts, L., Brennecke, J., Hodges, E., Hannon, G.J., Kellis, M.: Systematic discovery and characterization of fly microRNAs using 12 Drosophila genomes. Genome Research 17(12), 1865 (2007)
12. Mathuriya, A., Bader, D., Heitsch, C., Harvey, S.: GTfold: A Scalable Multicore Code for RNA Secondary Structure Prediction. Technical report, Georgia Institute of Technology (2008)