Accelerating the Nussinov RNA Folding Algorithm with CUDA/GPU

Dar-Jen Chang[1] Christopher Kimmer[2] Ming Ouyang[1]
[1]Computer Engineering & Computer Science Department, University of Louisville, Louisville, KY 40292, USA
[2]Informatics Department, Indiana University Southeast, New Albany, IN 47150, USA

***Abstract*—Graphics processing units (GPU) on commodity video cards have evolved into powerful computational devices. The RNA secondary structure arises from the primary structure and a backbone of canonical, Watson-Crick base pairings (A-U, C-G), and to a lesser extent, the G-U pairing. Early computational work by Nussinov formulated the problem of RNA secondary structure prediction as a maximization of the number of paired bases, which led to a simplified problem amenable to a dynamic programming solution for $O(n^3)$ serial time. This article describes a GPU implementation of the Nussinov dynamic programming. Computation results show that the GPU implementation is up to 290 times faster than the CPU.**

I. Introduction

Graphics processing units (GPUs) on commodity video cards have evolved into powerful computational devices tailored towards the needs of the 3-D gaming industry for high-performance, real-time graphics engines. In the past, software development on GPUs has been geared exclusively towards graphics through the use of languages such as OpenGL shading language and Direct3D high-level shader language. In 2006, Nvidia Corporation released a new generation of GPUs designed for general purpose use. These G80 series GPUs provide up to 128 stream processors and support 12,288 active threads. This new architecture facilitates efficient general purpose computing on GPUs (GPGPUs). In 2007, Nvidia Corporation released an extended C language for GPU programming called CUDA [7], short for Compute Unified Device Architecture. Using CUDA, innovative data-parallel algorithms can be implemented in general computing terms to solve many important and non-graphics applications such as database searching and sorting, medical imaging, protein folding, and fluid dynamics simulation.

A less explored but potentially fruitful area suitable for GPU acceleration is the prediction of RNA's three-dimensional, secondary structure. This structure buries certain nucleotides, allowing it to regulate gene expression in some instances by shielding gene initiation sites which should not be decoded [6]. Moreover, RNA secondary structure is more strongly conserved than the nucleotide sequence so that analysis of secondary structures across species is the primary means of determining evolutionary distance between RNA sequences [5]. Consequently, prediction of secondary structure for a given RNA sequence is an important problem in computational biology [1]. The secondary structure arises from the primary structure (i.e. the nucleotide sequence) and a backbone of canonical, Watson-Crick base pairings (A-U, C-G), and to a lesser extent, the G-U pairing. More complex structural features arise when there are incompatible or unmatched bases leading to bulges, loops, and hairpins. The structures containing only these features are topologically the simplest. If additional contacts between nucleotides occur beyond simple pairing, pseudoknots result where a base is bonded to more than one other at a time.

Corresponding author: Ming Ouyang, ming.ouyang@louisville.edu

The secondary structure is determined by the free energy of the RNA molecule, but a first-principles treatment is not practical for even moderately-sized molecules. There is a separation of energy scales where the atoms within a base are tightly bonded while the inter-base contacts are much more loose [9]. Consequently, each base can be regarded as a fixed body where the important interaction is between base pairs. Early computational work by Nussinov formulated the problem as a maximization of the number of paired bases, which led to a simplified problem amenable to a dynamic programming solution. More advanced methods based on so-called stacking energies [10] are required for better structural predictions, but the lack of work on this problem using GPUs makes the Nussinov method a worthy starting point. Moreover, little work using GPUs has focused on dynamic programming problems with greater than quadratic complexity, although at least one work has focused on the problem without describing the CUDA code in great detail [8].

The work described herein implements the Nussinov algorithm for GPU computation. The results show that a GPU card is 290 times faster than the CPU in performing the Nussinov algorithm on an RNA sequence of 15,000 bases. Section II describes the algorithm and the implementation. Section III contains the results. Section IV presents some discussion.

II. Algorithm and Implementation

A. Algorithm

RNA is a polymer of four nucleotides, A, C, G, and U. An RNA molecule folds back onto itself, and G-C and A-U pairs may form hydrogen bonds. The unpaired nucleotides may form *bulges*, *hairpins*, *loops*, and *pseudoknots*. It is computationally intractable to consider all possible combinations of configurations in RNA folding. Thus heuristics are employed to simplify the computation in practice. The Nussinov algorithm [6] finds a folding with the maximum hydrogen-bonded pairs of nucleotides in $O(n^3)$ serial time.

978-1-4244-9991-5/11/$26.00 ©2011 IEEE

There may be multiple foldings with the same maximum number of bonded pairs.

Let $\texttt{rna}_1$, ..., $\texttt{rna}_n$ be the sequence of the RNA polymer where each $\texttt{rna}_i$ is a nucleotide. The Nussinov algorithm fills an $n \times n$ matrix C with non-negative integers. Durbin *et al.* described the following formulation in [1]. For initialization, $C_{i,i} = 0$ for $i = 1, \ldots, n$, and $C_{i,i-1} = 0$ for $i = 2, \ldots, n$; that is, the main diagonal and the diagonal immediately below it are filled with zeros (Figure 1.a). Let the function $\texttt{bond}(a, b)$ return 1 if a and b form hydrogen bonds, and 0 otherwise. Let us assume $i < j$ for the rest of this article. Then the upper triangle of C is recursively defined by:

$$C(i,j) = \max \begin{cases} C(i+1,j), \\ C(i,j-1), \\ C(i+1,j-1) + \texttt{bond}(\texttt{rna}_i, \texttt{rna}_j), \\ \max_{i<k<j-1}\{C(i,k) + C(k+1,j)\}. \end{cases} \quad (1)$$

After the matrix C is filled, $C(i,j)$ is the maximum number of bonded pairs in the subsequence $\texttt{rna}_i$, ..., $\texttt{rna}_j$; in particular, $C(1,n)$ is the maximum number of bonded pairs for the whole RNA.

The initialization and recursion in Equation (1) allow base-pairing between two adjacent bases, which is prohibited in biological molecules. For example, Figure 1.b shows the bonding of neighboring G and C as calculated by the algorithm. This can be easily fixed by using a slightly different initialization: $C_{i,i} = 0$ for $i = 1, \ldots, n$, and $C_{i,i+1} = 0$ for $i = 1, \ldots, n-1$; that is, the main diagonal and the diagonal immediately above it are filled with zeros (Figure 1.c).

For considerations in implementation, two revisions of the recursion in Equation (1) are needed. First, to reduce divergence in CUDA/GPU parallelization, the first two cases in the recursion of Equation (1) can be absorbed into the fourth case, leading to:

$$C(i,j) = \max \begin{cases} C(i+1,j-1) + \texttt{bond}(\texttt{rna}_i, \texttt{rna}_j), \\ \max_{i \le k < j}\{C(i,k) + C(k+1,j)\}. \end{cases} \quad (2)$$

The second revision, which results in an important improvement to performance, comes from the following observation. The second case of Equation (2) finds the maximum in the vector sum of the row vector $C(i, i \ldots j-1)$ and the column vector $C(i+1 \ldots j, j)$ (Figure 1.d). If the matrix C is stored in a row-major array, the access to the column vector is very inefficient with the modern computer memory hierarchy. Specifically, all elements of the column vector reside in different cache lines, because in practice the width of the matrix C is much wider than the width of a cache line. On top of this, if the column vector is long, its access may exhaust the cache memory, and there will be many cache misses. For CUDA/GPU memory hierarchy, the memory access to a column vector stored in a row-major array produces what is called *non-coalesced global memory access* in Nvidia publications [7]. This kind of access is executed in a serial way although the CUDA code is meant for parallel execution. Regardless of the platforms, this problem can be circumvented by copying $C(i,j)$ to $C(j,i)$ after $C(i,j)$ is calculated; that is, both the upper and the lower triangles are symmetrically filled. Then the row vector $C(j, i+1 \ldots j)$ is used in the place of the column vector $C(i+1 \ldots j, j)$. Thus the recursion of the Nussinov algorithm is further rewritten as:

$$C(i,j) = \max \begin{cases} C(i+1,j-1) + \texttt{bond}(\texttt{rna}_i, \texttt{rna}_j), \\ \max_{i \le k < j}\{C(i,k) + C(j,k+1)\}. \end{cases} \quad (3)$$

Figure 1.e illustrates the memory access patterns of Equation (3).

The matrix element $C(1,n)$ gives the number of base pairs in the optimal base-paired structures. In general there are many alternative base-paired structures with the same maximal number of base pairs. A traceback algorithm given in [1] can be used to find one optimal structure. To find all optimal structures, however, an exhaustive tree search algorithm has to be implemented and the time and space complexity of the search algorithm could be exponential depending on the looping pattern of the RNA sequence at hand. The sequence AAAGCUUU has a unique optimal base-paired structure:

```
1 2 3 4 5 6 7 8
A A A G C U U U
8 7 6 0 0 3 2 1
```

This means first base A is paired with eighth base U, the second base paired with the seventh, the third paired with the sixth, the fourth paired with none, and so on. However, the sequence GGGAAAUCC has six optimal base-paired structures. Wiese *et al.* [11] described *jViz.RNA*, a Java tool for RNA secondary structure visualization, which can be used to visualize the generated base-paired structures.

B. Implementation

1) Implementation on the CPU Platform: The implementation of the Nussinov algorithm on the CPU platform is straightforward. Let D_3 be the diagonal ($n-2$ elements) above the main diagonal of the matrix C, let D_4 be the one ($n-3$ elements) above D_2, and so on, and let D_n be the last diagonal (1 element). The C code on the CPU platform has three nested loops. The outermost one loops over the diagonals: $D_3, \ldots, D_n$, the middle one loops over the elements along the diagonal, and the innermost one loops over the vector sum of the two row vectors of Equation (3) and finds the maximum.

2) Overview of the CUDA/GPU Platform: Three Nvidia GPU device cards, Tesla C870, Tesla C1060, and Tesla C2050, will be used in the present study. They were released in 2008, 2009, and 2010, respectively. Detailed information about them can be found in Nvidia publications such as [7]. Table I lists some features of these cards. The peak GFLOPs is for single-precision floating point calculation, which is only a guide for the purpose of the present study, because the Nussinov algorithm involves integer calculation only.

Nvidia has a programming guide to CUDA [7], which is an extension of C [3]. Briefly, the Single Program Multiple Data (SPMD) code is written in a GPU *kernel* function, the data to be operated on are copied from CPU RAM to GPU

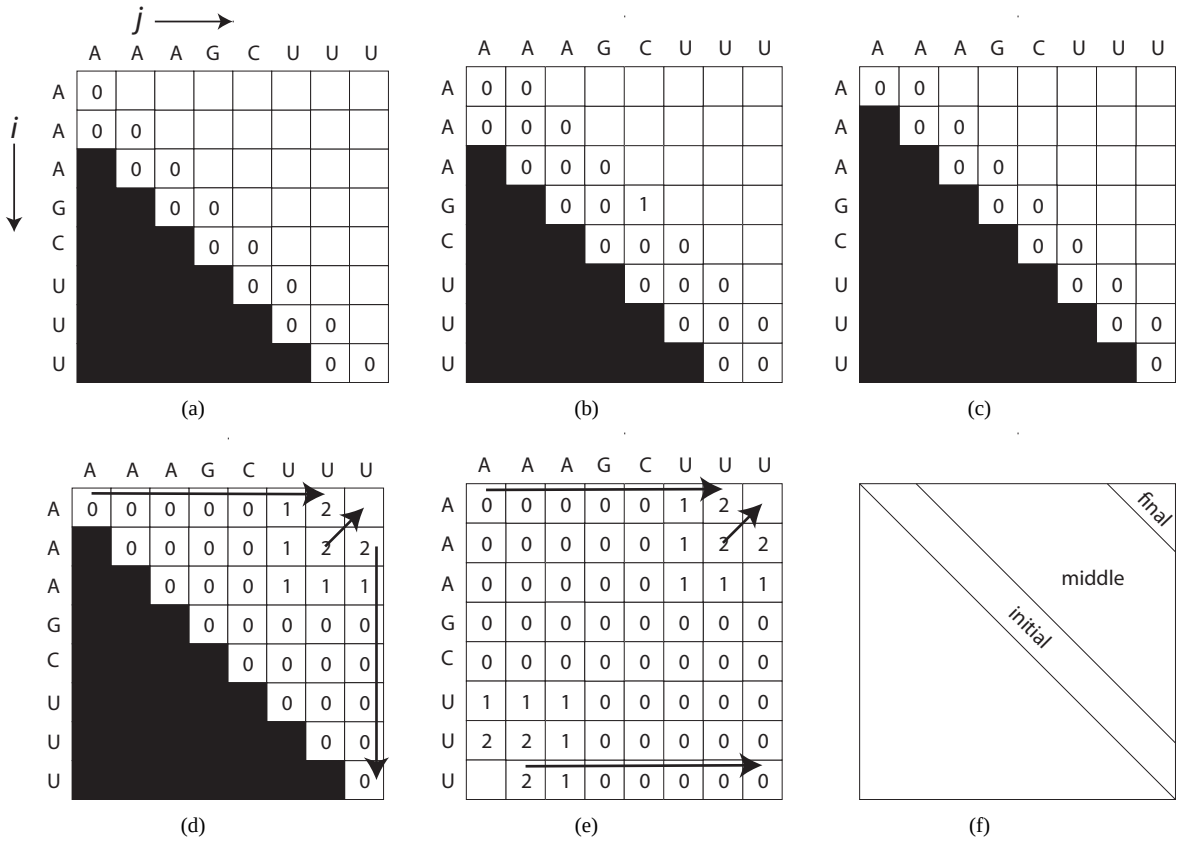


Fig. 1. Illustrations of implementing the Nussinov algorithm

	C870	C1060	C2050
# of multiprocessors	16	30	14
# of cores	128	240	448
Peak GFLOPs	518	933	1,288
RAM (MB)	1,536	4,096	3,072
Bus width (bits)	384	512	384

TABLE I
FEATURES OF THREE NVIDIA TESLA CARDS.

RAM, the C program running on the CPU initiates the data-parallel computation via a kernel function call, and the results are copied from GPU RAM back to CPU RAM. A GPU kernel function contains the code that will be executed simultaneously by the GPU processor cores. CUDA supports a large number of threads. The threads are organized into *blocks*, where there may be up to 512 (2^9) threads in each block for Tesla C870 and Tesla C1060, and up to 1,024 for Tesla C2050. The blocks are further organized into a *grid* of up to $(2^{16}-1)\times(2^{16}-1)$ blocks. CUDA offers one- and two-dimensional grids and one-, two-, and three-dimensional blocks of threads. Each block of threads is executed on a multiprocessor. In general, high performance is achieved by having many blocks of threads so that the utilization of all the multiprocessors is kept high.

The GPU device provides *registers* and *local memory* for each thread, a *shared memory* for each block, and a *global memory* for the entire grid of blocks of threads. Although all threads execute the same GPU kernel function, a thread is aware of its own identity through its block and thread indices, and thus a thread can be assigned a specific portion of the data on which it can perform computation. The shared memory for a block of threads is fast, yet it is limited in size. One strategy to attain high performance is for the threads in the same block to collaborate on loading data that they all need from the global memory to the shared memory. The shared memory is further partitioned into banks. The threads in the same block may access different banks simultaneously, yet a memory bank conflict will serialize the threads involved in the conflict. Thus another strategy for high performance is to avoid bank conflicts as much as possible. Because the data are stored in the global memory before a kernel function starts execution, the access to the global memory may become a bottleneck. If a *warp* of 32 threads (being simultaneously executed on a multiprocessor) accesses consecutive global memory locations with the proper alignment, and if the memory bus width accommodates them, the 32 load instructions are coalesced into one global memory access. This motivates the formulation of the Nussinov algorithm in Equation (3).

3) Implementation on the CUDA/GPU Platform: Three CUDA kernel functions are used to fill the upper triangle of C. Let us consider Figure 1.f, which divides the upper triangle of C into three segments: initial, middle, and final. In the

initial segment, there are many elements along the diagonals. However, the computation at each element is shallow in the sense that the length of the row vectors $C(i, i \ldots j-1)$ and $C(j, i+1 \ldots j)$ in Equation (3) is short. Thus in the kernel function for the initial segment, one thread is tasked for one element of the diagonal. The grid contains $\lceil n/128 \rceil$ blocks, and a block contains 128 threads. However, there is no interaction or collaboration among the threads. This initial kernel is invoked once for every diagonal in the initial segment.

The middle segment is characterized by many elements along the diagonals and adequate computation at each element. Thus in the kernel function for the middle segment, one block of threads is tasked for one element of the diagonal. The size of the grid is the length of the diagonal. When a block of threads works on an element, they calculate the vector sum in the second case of Equation (3), and they perform a reduction operation to find the maximum. Parallel reduction on the CUDA/GPU platform is extensively discussed in [2]. After the reduction, the first thread of the block compares the maximum to the first case of Equation (3) and writes the final maximum to the global memory. This middle kernel is invoked once for every diagonal in the middle segment.

The final segment is characterized by few elements along the diagonals and intensive computation at each element. Thus in the kernel function for the final segment, a whole grid is tasked for one element of the diagonal. The second case of Equation (3) is calculated by two reductions. The first reduction is performed by all blocks where threads within a block work together to reduce their portion of the vector sum into one maximum number. The second reduction is to further reduce these per-block maximum numbers into the overall maximum number. This final kernel is invoked once for every element in the final segment.

The transition from the initial kernel to the middle one is guided by the desire to switch to the parallel reduction in the middle kernel as soon as there is enough work for a block of threads. For Tesla C870 and Tesla C1060, this happens when the length of the vectors in the second case of Equation (3) is at least 32; for Tesla C2050, the length is at least 64.

The transition from the middle kernel to the final one is delayed as long as the diagonal length is no less than the multiprocessor number. If the middle kernel is still used when the diagonal length is less than the multiprocessor number, the extra multiprocessors will be idling and the GPU utilization is reduced. This transition happens when the diagonal length is less than 16, 32, and 14 for Tesla C870, C1060, and C2050, respectively.

III. Results

The performance of the CPU and GPU implementations are compared by folding RNA sequences of lengths $1,000, 2,000, \ldots, 16,000$. The Linux server has a quad-core CPU (AMD Phenom II X4 965 Processor) running at 3.4 GHz. There are 16 GB of CPU RAM. The three Nvidia Tesla cards are installed on PCI express $16\times$ slots. The computation time in Table II is the average of three separate runs. The variation in the running time is less than 1%. When folding the longest sequence (16,000 bases), the (single thread) CPU computation takes four hours 45 minutes, while the newest GPU (C2050) needs only a little over one minute. The largest speedup is achieved when a sequence of 15,000 bases is folded; the Tesla C2050 is 290 times faster than a single thread CPU computation.

As illustrated in Figure 1.f, the GPU computation is separated into three segments: initial, middle, and final. Table III lists the breakdown of the Tesla C2050 computation time in these three segments. If executed serially, the initial, middle, and final segments will take $O(n)$, $O(n^3)$, and $O(n)$ time, respectively. Indeed, Table III shows that the computation time is mostly spent in the middle segment.

IV. Discussion

The Nussinov dynamic programming algorithm is computed by three GPU kernels. The characteristics of the initial segment (Figure 1.f) are that there are $O(n)$ elements along the diagonals, and each element involves $O(1)$ serial computation. The CUDA kernel is invoked for one diagonal at a time, where the execution configuration uses a one-dimensional grid with one-dimensional blocks, and one thread is tasked for one element of the diagonal. After the value of an element is computed, it is written to the matrix C stored in the GPU global memory. This value will be used up to three times in the recurrence of Equation (3). Because the value is used only three times, it is not worth the trouble to load it to the shared memory to be used by collaborating threads.

The characteristics of the middle segment are that there are $O(n)$ elements along the diagonals, and each element involves $O(n)$ serial computation. The CUDA kernel is invoked for one diagonal at a time, where the execution configuration uses a one-dimensional grid with one-dimensional blocks, and one block of threads is tasked for one element of the diagonal. The threads in the same block collaborate to compute the second case of Equation (3), which is an example of parallel reduction, and it is performed in the shared memory as described in [2].

The characteristics of the final segment are that there are $O(1)$ elements along the diagonals, and each element involves $O(n)$ serial computation. The CUDA kernel is invoked for one element of the diagonal at a time, where the execution configuration uses a one-dimensional grid with one-dimensional blocks, and the whole grid is dedicated to the computation the element. One possible direction for improvement of performance is to do the following. The CUDA kernel is invoked for one diagonal at a time. The execution configuration uses a two-dimensional grid, where the first dimension of the grid matches the length of the diagonal, and the second dimension uses multiple blocks of threads to compute one element of the diagonal. This new design will reduce the number of kernel launches, and it will increase the number of blocks during each launch and thus it will increase the utilization of the GPU streaming multiprocessors. However, the improvement to the overall performance will be marginal, because the final segment takes a small fraction of the total time (Table III).

sequence length	CPU time	C2050 time	C1060 time	C870 time	C2050 speedup	C1060 speedup	C870 speedup
1000	0.6	0.1	0.1	0.2	7.5	7.8	3.1
2000	11.6	0.3	0.3	1.4	40.0	36.2	8.2
3000	50.8	0.7	0.9	4.7	71.3	55.9	10.9
4000	135.9	1.4	2.0	10.9	95.9	66.9	12.5
5000	281.1	2.5	3.8	22.4	112.1	73.1	12.6
6000	534.3	4.1	6.5	36.3	131.6	81.7	14.7
7000	1028.5	6.2	10.3	57.6	166.0	100.0	17.8
8000	1509.1	8.9	15.3	85.6	168.7	98.4	17.6
9000	2731.0	12.4	21.7	121.8	219.9	126.1	22.4
10000	3931.7	16.8	29.6	171.2	234.6	132.7	23.0
11000	5462.2	21.9	39.3	222.2	249.3	139.0	24.6
12000	6816.1	28.3	50.9	288.7	240.8	133.8	23.6
13000	9564.4	35.6	64.6	365.7	268.7	148.0	26.2
14000	12189.5	44.0	80.7	456.8	277.1	151.1	26.7
15000	15591.6	53.7	99.1	567.6	290.1	157.3	27.5
16000	17118.6	65.0	120.8	678.5	263.2	141.7	25.2

TABLE II
COMPUTATION TIMES (SECONDS), AND GPU SPEEDUPS OVER CPU.

	initial	middle	final
1000	0.01	0.06	0.01
2000	0.03	0.25	0.01
3000	0.05	0.64	0.02
4000	0.09	1.29	0.03
5000	0.12	2.35	0.05
6000	0.18	3.82	0.06
7000	0.24	5.88	0.09
8000	0.30	8.53	0.11
9000	0.37	11.91	0.14
10000	0.46	16.14	0.17
11000	0.54	21.17	0.20
12000	0.64	27.43	0.24
13000	0.74	34.58	0.28
14000	0.86	42.82	0.32
15000	0.98	52.42	0.37
16000	1.10	63.51	0.42

TABLE III
BREAKDOWN OF C2050 COMPUTATION TIME (SECONDS) AMONG THE INITIAL, MIDDLE, AND FINAL SEGMENTS.

In Table II, an intriguing phenomenon is that the speedup is not monotonically increasing with the length of the sequences. In particular, when the sequence lengths are 8,000, 12,000, and 16,000, the speedups are less than when the lengths are 7,000, 11,000, and 15,000, respectively. This anomaly can be attributed to the seemingly sudden boost in CPU computation. Figure 2.a compares the computation times of CPU and Tesla C870; clearly, the CPU time is not smoothly increasing. At lengths 8,000, 12,000, and 16,000, the CPU computation is noticeably faster than what one would predict from the curve. This may be explained by that, at these specific sequence lengths, the width of the matrix C may be some multiples of the cache line. Additionally, cache prefetching as generated by the compiler may happen to be more effective at these lengths than others. In contrast, Figure 2.b plots the computation times of the three Tesla cards; clearly, the GPU computation time is smoothly increasing as one would predict from the curves.

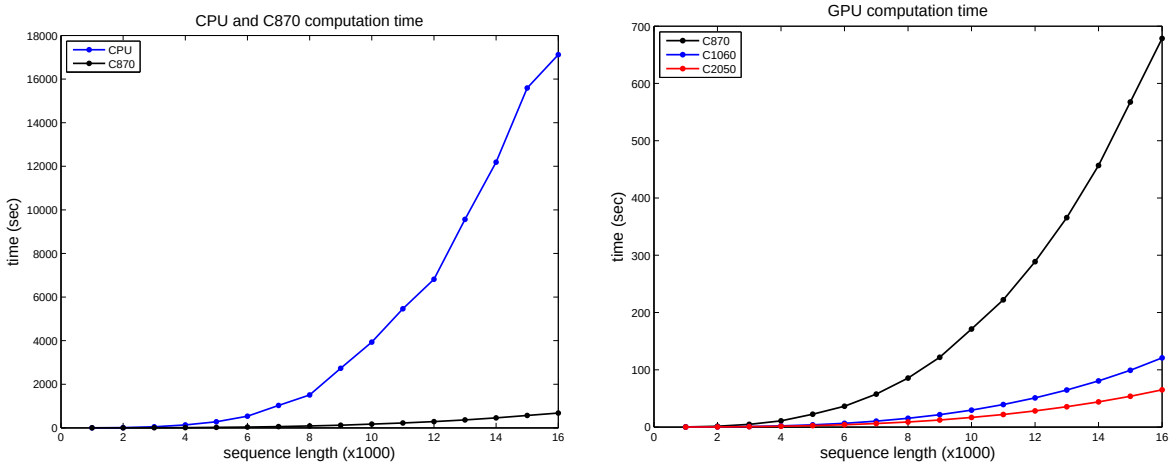
A key factor in attaining the performance described herein is the copying of the upper triangle of the matrix C to the lower triangle so that the elements of C are accessed in the row-major fashion (Figure 1.e). This allows coalesced global memory access. Otherwise, non-coalesced access will result in significant slow-down of the computation. The Nussinov algorithm considers the number of matched base pairs as the objective function. A different approach minimizes the free energy of the folded RNA structure, which requires $O(n^4)$ serial time [4]. It presents an interesting and challenging direction for further research.

ACKNOWLEDGMENT

Ming Ouyang is partially supported by DOE grant DE-EM0000197 to Ted Kalbfleisch and Eric Rouchka.

REFERENCES

[1] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis*. Cambridge University Press, 1998.
[2] Mark Harris. *Optimizing parallel reduction in CUDA*. Nvidia Corporation, 2007.
[3] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, 1988.



(a) (b)

Fig. 2. (a) CPU computation time versus C870 computation time. (b) GPU computation times.

[4] R B Lyngso, M Zuker, and C N Pedersen. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics*, 15(6):440–445, 1999.

[5] D. H. Mathews, J. Sabina, and M. Zuker. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. *JMB*, 288:911–940, 1999.

[6] Ruth Nussinov, George Pieczenik, Jerrold R. Griggs, and Daniel J. Kleitman. Algorithms for loop matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, 1978.

[7] Nvidia. *NVIDIA CUDA C Programming Guide, Version 3.1*. Nvidia Corporation, 2010.

[8] Guillaume Rizk and Dominique Lavenier. Gpu accelerated rna folding algorithm. In Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, Geert van Albada, Jack Dongarra, and Peter Sloot, editors, *Computational Science - ICCS 2009*, volume 5544 of *Lecture Notes in Computer Science*, pages 1004–1013. Springer Berlin / Heidelberg, 2009.

[9] I Tinoco Jr. and C. Bustamante. How RNA folds. *JMB*, 293:271–281, 1999.

[10] I Tinoco Jr., O. C. Uhlenbeck, and M. D. Levine. Estimation of secondary structure in ribonucleic acids. *Nature*, 230:362–367, 1971.

[11] Kay C Wiese, Edward Glen, and Anna Vasudevan. jviz.rna a java tool for rna secondary structure visualization. *IEEE Transactions on NanoBioscience*, 4(3):212218, 2005.