Parallelization of Dynamic Programming in Nussinov RNA Folding Algorithm on the CUDA GPU

Marina Zaharieva Stojanovski[1], Dejan Gjorgjevikj[2] and Gjorgji Madjarov[2]

[1]Formal Methods and Tools Group, Faculty EEMCS, University of Twente, The Netherland
[2]Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, Skopje, Macedonia
[1]m.zaharieva@utwente.nl, [2]{dejan, gjorgji.madjarov}@finki.ukim.mk

Abstract. When an RNA primary sequence is folded back on itself, forming complementary base-pairs, a form called RNA secondary structure is created. The first solution for the RNA secondary structure prediction problem was the Nussinov dynamic programming algorithm developed in 1978 which is still an irreplaceable base that all other approaches rely on. In this work, the Nussinov algorithm is analyzed but from the CUDA GPU programming perspective. The algorithm is radically redesigned in order to utilize the highly parallel NUMA architecture of the GPU. The implementation of the Nussinov algorithm on CUDA architecture for NVidia GeForce 8500 GT graphic card results with substantial acceleration compared with the sequential executed algorithm.

Keywords. Dynamic programming, parallel, RNA, GPU, CUDA, secondary structure, Nussinov

1 Introduction

The tertiary structure of the RNA molecule plays the most important role in understanding and analyzing the RNA molecule function. However, since prediction of this structure can be done only with some expensive and time-consuming methods, the easier solution is predicting the RNA secondary structure which can be later used for the tertiary structure prediction. As a result of this, although the form in which the RNA occurs in the real organisms is its tertiary structure, the research in this domain has been primarily focused on the RNA secondary structure prediction. This is a problem which has been researched for more than 30 years, but is still very popular in the field of bioinformatics and there is still no single solution considered as the most valuable one.

The Nussinov algorithm [1] is the base and carries the logic which is used in most of the other algorithms. This paper focuses on implementing the Nussinov algorithm on CUDA GPU architecture. The results obtained with this implementation would not be as accurate as those from the other later developed more advanced algorithms, but the key point with this work is accelerating the Nussinov algorithm as a base for the other algorithms. The positive result found with this implementation would lead to reusing the base idea from this work for efficiency improvement of others advanced

RNA folding algorithms. The choice for the Nussinov algorithm to be the target of this research was predominantly because of its simplicity.

1.1 RNA Secondary Structure

In mathematical term, the RNA primary structure can be considered as a sequence of symbols contained in the set S={A,C,G,U} where each of the symbols is representing one of the four possible bases: Adenine, Cytosine, Guanine, and Uracil, respectively. This long sequence can be folded [6] in a way that the symbols are paired with each other forming the well-known Watson-Crick pairs (A-U, C-G) or the Wooble pair (G-U). Thus, the RNA secondary structure can be defined as a set of ordered pairs

$$S = \{(i,j) \quad | \quad 1 \le i < j \le n, \quad \textit{where n is the length of the sequence}\}$$

The pair (i, j) represents that a pairing has occurred between symbols at position i and position j. The following rules must be satisfied:

1. The distance between two paired symbols must be greater than 3;
2. A symbol can be included in at most 1 pair.

The secondary structure prediction problem can be defined as:
Given the RNA primary structure, the secondary structure should be predicted.

1.2 Nussinov Algorithm for RNA Folding

The first solution for the RNA secondary structure prediction problem is the Nussinov algorithm [1] developed in 1978. The base idea for Nussinov algorithm approach is the assumption that the secondary structure is the form in which the number of paired bases is maximized. Since the number of possible secondary structures grows exponentially as the sequence length increases, the most naive solution (finding all possible structures and selecting the one with the largest number of pairs) would have no sense. The most obvious technique that can deal with this problem while achieving an efficient solution is the dynamic programming.

The core of each dynamic programming algorithm is a set of recurrent relations that define the additional algorithm steps as well as the performance and memory complexity. The relations for the Nussinov algorithm are displayed below.

$$N(i,j) = \max \begin{cases} N(i+1,j) \\ N(i,j-1) \\ N(i+1,j-1) + \delta(i,j) \\ \max_{i<k<j}[N(i,k) + N(k+1,j)] \end{cases} \qquad (1)$$

where

$$\delta = \begin{cases} 1, & \textit{if } x_i \textit{ and } x_j \textit{ may pair} \\ 0, & \textit{else} \end{cases} \qquad (2)$$

The equations defined show that the performance complexity of the Nussinov algorithm is cubic O(n^3) with linear complexity of computing the value of one element. The memory complexity is O(n^2), since the computed values are stored in a two-dimensional *n x n* array.

The cubic performance complexity results with very inefficient and time-consuming execution of the Nussinov algorithm considering the fact that the experiments are usually made with a sequence of several thousand bases. This is where the main motivation of this paper came from. The acceleration of the Nussinov algorithm is done by implementing the algorithm on CUDA GPU architecture [2][3] in order to achieve massive parallelization and to substantially decrease the execution time.

2 CUDA Implementation

2.1 General Overview

The starting point of designing a parallel algorithm is making a good analysis of the sequential algorithm and getting the basic idea of how this algorithm can be divided into smaller tasks that can run in parallel. Analyzing the equation (1), enough information can be extracted for the Nussinov algorithm procedure. A two-dimensional array (*n x n*) is used for storing the computed values in this dynamic programming algorithm (Fig. 1 left). The values in the main diagonal are initialized with zeros and the lower triangular part of the matrix remains unused. Additionally the values for the next three matrix diagonals above the main diagonal are set to zero which comes from the fact that the distance between two paired bases must be greater than 3. The diagonals above the main diagonal are iterated and the values of the elements in the diagonal are computed in each iteration. The last diagonal contains only one element (the upper right corner) with the result value indicating the number of pairs contained in the secondary structure for the current RNA sequence.

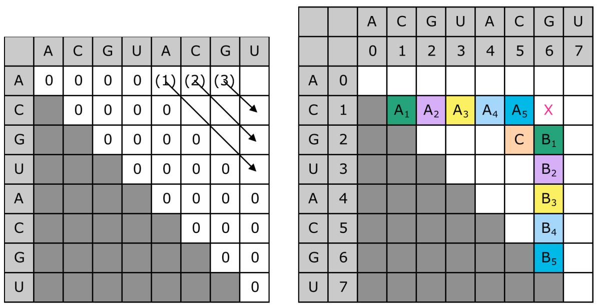

Fig.1. Nussinov Matrix (left) and Computing the value of an element (right)

Considering the fact that one matrix element can be found as a computation of the already computed values in the same row and the same column and the adjacent element (down-left) (Fig. 1 right), the following conclusion can be defined:

All elements in a same diagonal are independent of each other and their computing can run in parallel.

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.

A logical consequence of these facts is the general overview of the algorithm design defined through the following steps: 1) The 2-dimensional array is stored into CPU initialized with all values equal to zero; 2) The array is copied to the GPU global memory; 3) All diagonals above the main one are iterated and a kernel function is called for each of them. This design provides parallelism of computing the values for elements from one diagonal, while their independency ensures that the computed values would always be correct.

2.2 Kernel Design

Designing the kernel function means defining subtasks and mapping them to number of threads in a way that they would all execute the preferred function – computing the values of the elements for a specific diagonal. The first question is: “How many threads would be responsible for computing the value of one element?”. The proposed solutions are: 1) a thread; 2) a block of threads; 3) a grid of threads.

If we consider the solution (1), one thread per element would mean that the number of threads included in the kernel would not be large enough in order to fully utilize the GPU device capacity. The maximum number of threads in the grid would be equal to the primary sequence length decreased by 4 (that is the length of the first iterated diagonal). Another disadvantage for this solution is the fact that computing the value for one element would remain sequential with linear complexity.

These negative sides would be avoided in case the solution (2) is used. However, in this case additional effort is needed for defining the parallel execution of the block of threads. The threads will have to be synchronized so that the computation of one element value can run in parallel. Shared memory would be used so that the threads would communicate with each other.

We avoid the solution (3) since if the whole grid is responsible for computing the value of single element, the latent global memory should be used and achieving efficiency would be more complicated. A possible solution is to use this kernel function for diagonals containing small number of elements, but this is not the case in the implementation we propose.

Following these analyses, in the current implementation we have chosen the solution (2). However, since the diagonals that are closer to the main diagonal contain elements whose computations are simple and depend of a small number of elements, we create two different kernel functions. The first one would be defined according to solution (1) and will be called only for diagonals which are close to the main diagonal, and the second one would follow the solution (2) and would be called for the remaining diagonals.

Kernel 1 function The base idea for this kernel function is that the task of computing the values for one diagonal is divided into smaller subtasks – computing the value for one element and each of these subtasks is mapped to a specific thread. The diagonal is divided into a number of segments each of them assigned to a specific block of threads (Fig. 2). The kernel configuration parameters would be *(blocksize, [n/blocksize])* where *blocksize* is the number of threads per block which we define to be equal to 128 and *n* is the number of elements in the specified diagonal.

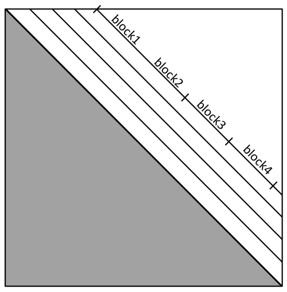

Fig.2. Computing values for a matrix diagonal with kernel 1 function

The logic of this kernel function is simple: First the index of the thread is computed (using the build-in variables) and then it is used to identify the element the thread should work on. There is no need to use the shared memory since during the execution of the kernel for diagonal d, a value of the element (x, y) from the global memory can be accessed at most 3 times (while computing: the value of the element at row x contained in d; the value of the element at column y contained in d; the value of the element (x+1, y-1) if it is contained in d.

Kernel 2 function This kernel function is designed in a way that the subtask of computing the value of one element is executed by a block of threads. The kernel is called with the following configuration parameters *(blocksize, n)* where *blocksize* is set to be 128, and *n* is the number of elements in the appropriate diagonal. Fig. 1 shows the process of computing the value in one cell. The value X is computed as:

$$X = \max\{A_1 + B_1, A_2 + B_2, A_3 + B_3, A_4 + B_4, A_5 + B_5, C + \delta(G,C)\}$$

The most suitable solution would be the following: The sum $A_i + B_i$ *(i=1..5)* is computed by the thread *i*. The number of sums (in this case 5) will usually be greater than the number of threads contained in the block, thus the thread *i* would be responsible for computing more than one sum, $A_i + B_i$, $A_{i+blocksize} + B_{i+blocksize}$, $A_{i+2*blocksize} + B_{i+2*blocksize}$… and finding the max value from all sums. Each computed maximum value would be stored in the shared memory.

This procedure will result with an array of elements with length equal to *blocksize* (the number of threads per block) stored in the shared memory. The next step is finding the maximum of this array which can be seen as a typical reduction pattern problem. Thus, the reduction pattern is used in a way that a thread is computing the maximum value of two array elements, then after the threads are synchronized the same step is repeated. With every iteration the array size would be divided by two and after $\log_2 N$ operations the final result would be found, the maximum value of all elements in the array.

Global memory access pattern impact on performance This solution seems valuable, but it still does not give sensible results. The problem that must be handled is the global memory access that gives a great negative influence to the algorithm execution performance. When loading the values from the global memory, the accesses must be coalesced in order to avoid the degradation that may happen because of the highly latent global memory. In order to achieve the coalesced global memory access the following solution is proposed:

Since the lower matrix triangular half part does not store any values, the values from the upper part are copied symmetrically to the main diagonal and shifted one position left. This way, when a thread accesses the two matrix elements which sum should be computed, both of the elements would be stored in the same matrix column.

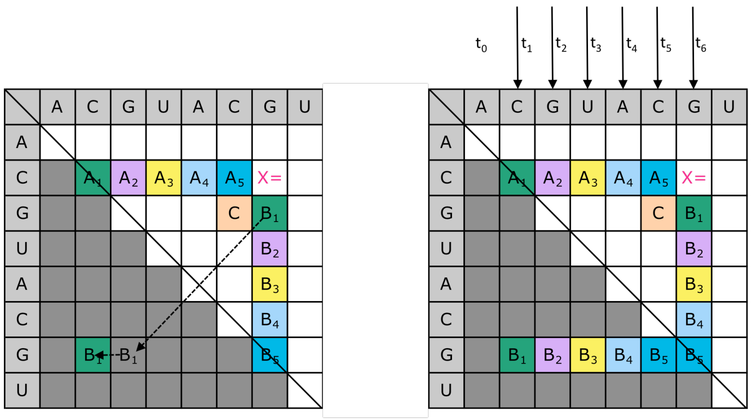

Fig.3. Copying matrix element and distributing the threads accessing global memory

In order to maximize the memory bandwidth while accessing the elements they should be $16k$ elements apart, requesting the matrix dimension to be $n = 16k$ ($k \in N$). This insures that the OFFSET value while accessing the global memory will be factor of 16, that provides most effective bandwidth. All matrix values are initially set to 0. When new value for a matrix element is computed, it is stored both in the original matrix cell (above the main diagonal) and in its mirrored copy (below the main diagonal).

The Traceback Stage In the second part of the Nussinov algorithm, the traceback stage is executed after all kernel functions have finished. The matrix with all updated values is copied back to the host, where the traceback stage is executed sequentially. The solution is recursive with O(n^2) performance complexity.

3 Results

The efficiency of the proposed Nussinov algorithm is tested with a set of experiments using RNA primary sequences with different lengths. The CUDA implementation is compared with the sequential Nussinov algorithm implemented in C++ and Java and substantial results were found.

The experiments were made on a Windows PC with Intel Core2Duo E4500 running at 2.2GHz and 1GB of RAM. The parallel CUDA algorithm was executed on NVidia GeForce 8500 GT architecture (2 multiprocessors and 16 cores). All the tables below present the execution time in seconds. It should be noticed that these results do not include the traceback stage but this would not influence the comparison since this part of the algorithm is executed sequentially on the CPU host both in the original sequential and in the parallel CUDA algorithm.

The test sequences are randomly generated using the Markovian model having order 0 (Bernoulli model) where the frequency of occurrence of the symbols A, C, G, U is equal to 0.25. The GenRGenS software [4] is used.

3.1 Sequential and Parallel CUDA Algorithm Comparison

The experiments include execution of three different implementations of the Nussinov algorithm on different sequence lengths. The three implementations include: sequential implementation in JAVA (JRE 1.6), sequential implementation in C++ (Visual C++ 2008 Express Edition, Maximize Speed Optimization) and parallel implementation in CUDA C (CUDA kernel functions called from JAVA using JCUDA [5]).
The running times of the three different implementations on RNA sequences with different lengths are given in Table 1.

Results in Table 1 shows that as the sequence length grows, the acceleration is greater, thus for sequences with small length, there is even no acceleration, but for sequence with length equal 7860, the acceleration is 18 times compared to the sequential execution of the C++ implementation, and 31 times compared to the sequential execution of the JAVA implementation. Fig. 4 shows these results graphically.

Table 1. Sequential and parallel CUDA Nussinov algorithm execution results

Sequence length	GPU (Cuda)	CPU (C++)	CPU (JAVA)
640	**0.6**	0.5	0.6
1280	**1.8**	3	4.3
1920	**2.23**	10	17
2560	**3.98**	31	42.1
3200	**6.72**	68	86.9
3840	**10.47**	119	160.2
4480	**15.55**	198	270.8
5120	**21.9**	312	438
5760	**30.53**	452	690.9
6400	**41.3**	660	1081.8
7040	**54.6**	895	1511.7
7860	**68.5**	1242	2119.2

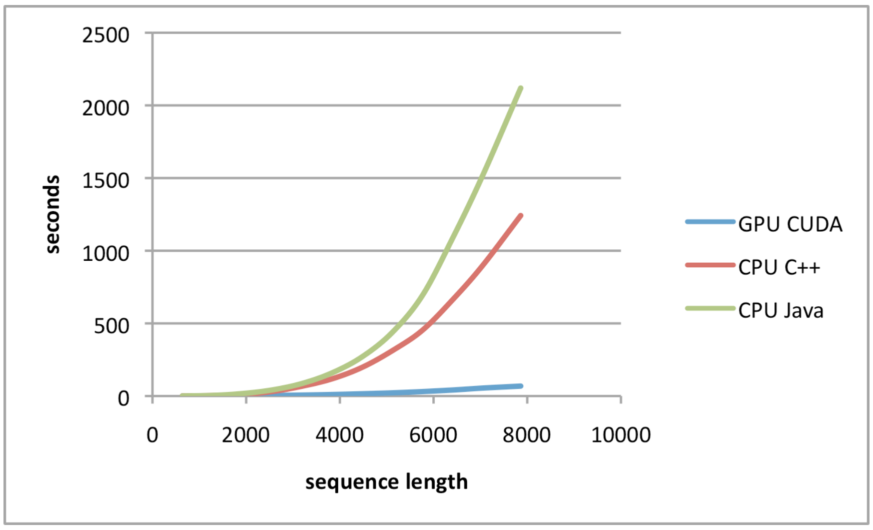

Fig.4. Nussinov algorithm in C++, Java and CUDA

3.2 Conditions for Achieving Acceleration

The following experiments are done in order to show the key points in the algorithm that contribute the most to acceleration. If we avoid the second kernel function and use only the kernel 1 function for all diagonals which means sequentially computing the value of element matrix, one even gets increase in the execution time compared to the sequential execution. Other experiments are done when no symmetrical copying of the elements is done. The result shows that the uncoalesced global memory accesses degrade the algorithm efficiency and the execution time is even longer compared to the algorithm with only one kernel. Fig. 5 displays these results graphically.

Table 2. Comparison between: CUDA using only kernel 1, CUDA with uncoalesced global memory accesses and CUDA proposed solution

Sequence length	CUDA with kernel 1 only	CUDA within coalesced accesses	CUDA proposed solution
640	1	1.3	**0.6**
1280	3.4	4.2	**1.8**
1920	13.5	15.6	**2.23**
2560	38.5	41.9	**3.98**
3200	93.5	102.5	**6.72**
3840	192.3	201.2	**10.47**

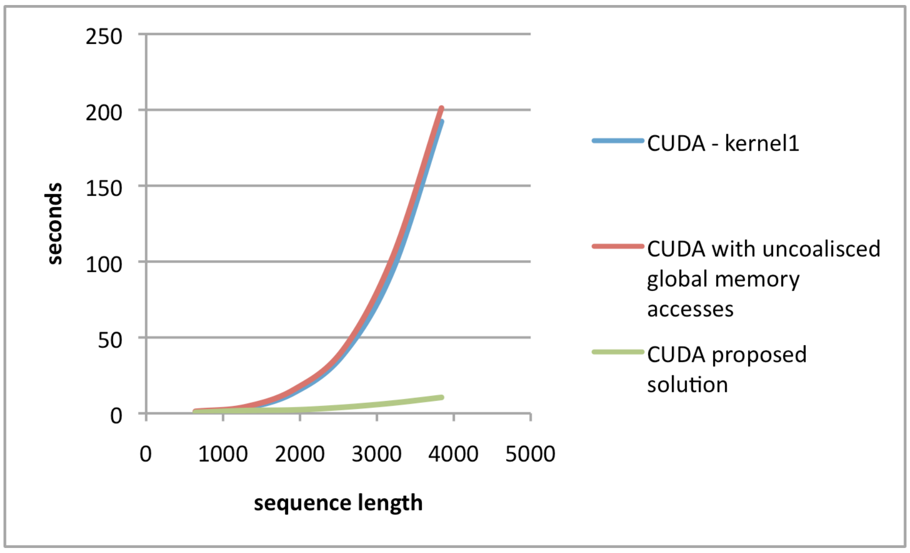

Fig.5. CUDA using only kernel1, CUDA with uncoalesced global memory accesses and CUDA proposed solution

As a conclusion, the key point of the algorithm acceleration is a proper design of the kernel 2 function and coalesced global memory access which in the proposed solution is achieved by symmetrically copying the matrix elements.

3.3 Border between Kernel 1 and Kernel 2

The moment when the kernel 2 is called instead of kernel 1 is after 32 iterations, which means that the sums of pairs that should be computed would be 32 (Table 3). The basic idea for this is that if we call kernel 2 function earlier, not all threads in a warp will be active when computing a value for an element.

Table 3. Choosing the border between kernel 1 and kernel 2 function.

Sequence length	0iterations (kernel 2 only)	32 Iterations	48 iterations	64 iterations	96 Iterations
1920	2.27	**2.23**	2.27	2.26	2.32
2560	4	**3.98**	4.02	4.05	4.06
3200	6.72	**6.72**	6.73	6.74	6.82
3840	10.48	**10.47**	10.55	10.53	10.59
4480	15.59	**15.55**	15.6	15.68	15.93
5120	22.1	**21.9**	22.03	22.02	22.17
5760	30.52	**30.50**	30.51	30.63	30.8
6400	41.5	**41.3**	41.4	41.9	42.1
7040	55	**54.6**	54.9	54.8	55.9
7860	69.1	**68.5**	68.7	69.5	69.8

Results show that the value 32 is the most appropriate to be the border between kernel 1 and kernel 2. The differences are small, and even if kernel 1 is totally avoided the results are similar with those of the proposed solution.

4 Conclusions

In this work, the Nussinov algorithm is redesigned in order to utilize the highly parallel NUMA architecture of the GPU. The implementation of the Nussinov algorithm on CUDA architecture for NVidia GeForce 8500 GT graphic card results with substantial acceleration compared with the sequential executed algorithm. The positive result in this implementation would lead to reusing the base idea from this work for efficiency improvement of others advanced RNA folding algorithms.

References

1. R. Nussinov, G. Pieczenik, J. R. Griggs, D. J. Kleitman. :Algorithm for Loop Matching. SIAM Journal on Applied Mathematics 35 (1), 68-82 (1978)
2. NVIDIA.: CUDA C Best Practices Guide, Version 3.2. Aug. 2010
3. NVIDIA.: CUDA C Programming Guide, Version 3.2, Oct.2010
4. Y. Ponty, M. Termier and A. Denise, GenRGenS.: Software for generating random genomic sequences and structures, Bioinformatics, 22(12, 1534-1535 (2006)
5. Yonghong Yan, Max Grossman and Vivek Sarkar “JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA”, Proceedings of Euro-Par 2009, August 2009.
6. W. Fontana, P. F. Stadler, E. G. Bornberg-Bauer, T. Griesmacher, I. L. Hofacker, M. Tacker, P. Tarazona, E. D. Weinberger, P. Schuster. RNA folding and combinatory landscapes. Phys Rev E 47(3), 2083–2099 (1993)