Accelerating Nussinov RNA secondary structure prediction with systolic arrays on FPGAs

Arpith Jacob*, Jeremy Buhler*, and Roger D. Chamberlain*†

*Dept. of Computer Science and Engineering, Washington University in St. Louis
†BECS Technology, Inc., St. Louis, Missouri
{*jarpith,jbuhler,roger*}@*cse.wustl.edu*

Abstract

RNA structure prediction, or folding, is a compute-intensive task that lies at the core of several search applications in bioinformatics. We begin to address the need for high-throughput RNA folding by accelerating the Nussinov folding algorithm using a 2D systolic array architecture. We adapt classic results on parallel string parenthesization to produce efficient systolic arrays for the Nussinov algorithm, elaborating these array designs to produce fully realized FPGA implementations. Our designs achieve estimated speedups up to $39\times$ *on a Xilinx Virtex-II 6000 FPGA over a modern x86 CPU.*

1. Introduction

Small RNA molecules carry out diverse functions in living cells, including catalysis of reactions [2], binding of small molecules [1], and targeted suppression of transcription and translation [5]. Although an RNA may be viewed as a linear sequence of characters, or *bases*, from the alphabet $\{A, C, G, U\}$, its function is actually determined by its *secondary structure* – the folded shape that results from pairing of complementary bases (mainly A-U and C-G) within one sequence. Determining this structure is key to analyses that identify and assign functions to RNAs.

Algorithms to predict the structure of single RNA molecules use empirical models to estimate the free energies of folded structures. These algorithms find a structure of minimum free energy for a given RNA using dynamic programming. The simplest folding algorithm, due to Nussinov [11], uses the number of base pairs as a proxy for free energy, preferring the structure with the most base pairs. This method was refined by Zuker [16] to use more detailed, more accurate energy models. More recent folding methods have used empirically learned models, in particular stochastic context free grammars (SCFGs) [14], that directly assign probabilities to potential RNA structures. All these methods, however use dynamic programming recurrences of the same basic shape as Nussinov's algorithm.

RNA folding by any of the above methods takes time cubic in the length of the RNA. Although folding computations are generally restricted to RNAs of fewer than than 200 bases, important biological applications, including RNA motif finding [6, 12] and search for functional RNAs in genomic DNA [10], may require folding thousands to millions of such short RNAs, which is computationally demanding. One way to overcome this computational bottleneck is to parallelize folding algorithms. In this work, we investigate the feasibility of fine-grain parallelization of Nussinov's algorithm, with the goal of creating an application-specific accelerator for large-scale search.

We base our designs on two classic 2D systolic arrays for optimal string parenthesization [3, 4], a problem whose dynamic programming solution has a shape similar to that of the Nussinov algorithm. We realize these designs in VHDL and benchmark our designs against software on a modern general-purpose x86 CPU, obtaining estimated speedups of up to $39\times$ using a Virtex-II 6000 FPGA. Greater speedups should be possible with newer generation FPGAs. Our work is an important step toward practical acceleration of streaming applications, like those mentioned above, that use RNA folding as their core computation.

In related work Tan et al. [15] describes a 2D array to implement the Zuker algorithm on a fine-grained architecture. Here dependent cells must be stored locally, requiring memory proportional to the size of the input in each processing element. In contrast, we use space-time methods to derive an array that has constant local memory requirement. Another recent thread of work in this area is Moscola's design of a systolic array architecture for SCFG alignment to RNA sequences [9].

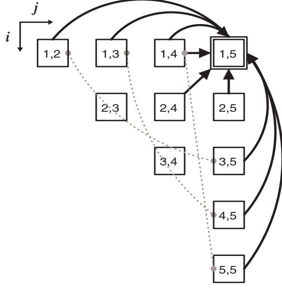

Figure 1: Nussinov data dependencies. $X(1,5)$ depends on three adjacent cells, plus five non-local cells.

2. The Nussinov Algorithm

Given RNA sequence S of length N, the Nussinov algorithm computes the largest number of base pairs $X(i,j)$ in any folded structure of subsequence $S[i]..S[j]$ as follows:

$$X(i,j) = \max \begin{cases} X(i+1,j) \\ X(i,j-1) \\ X(i+1,j-1) + \delta(S_i,S_j) \\ \max_{i<q<j} [X(i,q) + X(q+1,j)]. \end{cases} \quad (1)$$

The score δ evaluates to 1 if bases S_i and S_j are complementary or 0 otherwise. The data variable X is defined over the domain $1 \leq i < j \leq N$; the score of the best structure of the entire sequence is at $X(1,N)$. In this work, we are interested in accelerating only the computation of the value $X(i,j)$; the actual folded structures for sufficiently high-scoring RNAs may be computed later in software.

The Nussinov recurrence is challenging to accelerate because of its non-local dependency structure, shown in Figure 1. Although the first three dependencies of $X(i,j)$ reference only adjacent cells of the dynamic programming matrix, the fourth term references widely separated cells. For example, $X(1,5)$ depends on $X(1,4)$, $X(2,4)$, and $X(2,5)$ but also on $X(1,2)$, $X(1,3)$, $X(3,5)$, $X(4,5)$, and $X(5,5)$. On an FPGA, non-local cell dependencies result in routing delays that dominate the critical path of the computation. Worse yet, these non-local dependencies are *affine*; that is, $X(i,j)$ depends on other cells $X(r,s)$ such that the differences $i-r$ or $j-s$ are not constant but rather depend on i and j. Affine dependencies result in a nonuniform structure that cannot easily be mapped onto an array of identically connected processing elements.

To find a workable approach to parallelizing the Nussinov recurrence, we first recognize that it is closely related to the well-known *optimal string parenthesization problem*. Given a string S of length N and a cost $W(i,j)$ for adding parentheses around substring $S[i..j]$, this problem seeks the minimum cost $X(i,j)$ of any properly nested set of parentheses for $S[i..j]$. Its recurrence mirrors the fourth case of the Nussinov algorithm: $X(i,j) = W(i,j) + \min_{i<q<j}[X(i,q) + X(q,j)]$. Techniques for parallelizing this recurrence may therefore prove useful for Nussinov.

A classic result of Guibas et al. [4] shows how to parallelize string parenthesization using a 2D systolic array. Gachet et al. [3] later proposed an alternative array. While these designs overcome the key problem of non-local, affine dependencies, to our knowledge they have neither been implemented on FPGAs nor been adapted to perform RNA folding. The following sections review these arrays and describe how we adapt them to the Nussinov algorithm.

3. Computing the Schedule

We first modify Recurrence 1 to convert the $O(N)$-ary $\max$ operator to a binary operation. We then use middle serialization [3] to derive the following recurrence.

$$X(i,j,k) = \max \begin{cases} \delta(S_i,S_j) & \text{if } j-i=1 \\ \\ X(i+1,j,k) & \\ X(i,j-1,k) & \\ X(i+1,j-1,k) + \delta(S_i,S_j) & \text{if } k=1 \\ X(i,j,k+1) & \\ X(i,i+k,1) + X(i+k+1,j,1) & \\ X(i,j-k,1) + X(j-k+1,j,1) & \\ \\ X(i,j,k+1) & \\ X(i,i+k,1) + X(i+k+1,j,1) & \text{otherwise} \\ X(i,j-k,1) + X(j-k+1,j,1). & \end{cases}$$

Here, we have introduced a third dimension k, where $1 \leq k \leq \lfloor (j-i)/2 \rfloor$, that aggregates the fourth term in Recurrence 1. The score of the best structure is now at $X(1,N,1)$.

A data dependency of a cell $z = [i,j,k]$ in a recurrence can be described by a function $f(z) = Az + b$, where A is a 3×3 matrix and b a 3-vector. The dependency is uniform if A is the identity matrix; otherwise, it is affine. The above recurrence has four affine dependencies f_{1-4} in the terms $X(i,i+k,1)$, $X(i+k+1,j,1)$, $X(i,j-k,1)$ and $X(j-k+1,j,1)$. We must replace these affine dependencies by simpler uniform dependencies to produce a systolic array.

In what follows, we use concepts from the space-time mapping [7] of a recurrence onto a systolic array. For all points z computed by the recurrence, this mapping specifies

1. A *schedule* $\tau(z) = \lambda z + \alpha$ giving a positive execution time for every point such that if z depends on y, $\tau(z) > \tau(y)$ (causality constraint). Here λ is a vector and α a scalar value.

2. An *allocation function* $\pi(z) = [\pi_1(z), \pi_2(z)]$, giving the processing element (PE) in a 2D systolic array on which z is to be executed. The schedule and allocation functions satisfy the constraint $\tau(z) = \tau(y) \Rightarrow \pi(z) \neq \pi(y)$ for any two points z, y.

3.1. Pipelining Affine Dependencies

We use nullspace pipelining [13] to replace affine with uniform dependencies. Suppose a cell q is a dependency

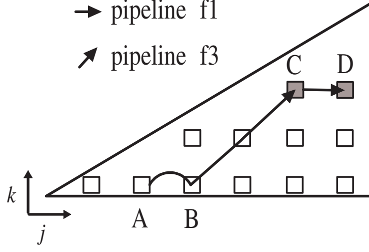

Figure 2: Simple pipeline for dependency f_3 and multistage pipeline for dependency f_1.

for cells $p_1 \dots p_m$. Rather than broadcast q's value to all cells, we may first deliver it to p_1, then transfer it from p_1 to p_2, from p_2 to p_3, and so forth up to p_m. The resulting dependency between adjacent cells p_t and p_{t+1} is uniform.

We first identify a basis vector for the nullspace of the matrix A in the dependency function that satisfies the causality constraint of the schedule. Adding this basis vector repeatedly to an initial cell p_1 describes a pipeline of cells that eventually includes every cell dependent on q. We initialize this pipeline by making available the cell value at q to the earliest scheduled dependent cell p_1. This is possible with a uniform dependency if p_1 is a constant distance from q; if not, we may be able to use multistage pipelining [13].

For lack of space, we do not show the steps to derive the pipelines for dependencies f_{1-4}. These steps are described for the string parenthesization problem in [13]. We use the following basis vectors for the four pipelines: $v_1 = [0,-1,0]$, $v_2 = [1,0,-1]$, $v_3 = [0,-1,-1]$, and $v_4 = [1,0,0]$. Pipelines for dependencies f_2 and f_3 can be initialized at the boundary $k = 1$ by vectors $v_2' = [2,0,0]$ and $v_3' = [0,-1,0]$ respectively. Recurrence 2 shows the two uniform variables X_2 and X_3 that replace the affine terms $X(i+k+1,j,1)$ and $X(i,j-k,1)$. Pipelines of f_1 and f_4 intersect the domain boundary $k = (j-i)/2$ and can only be initialized by multistage pipelining. We initialize the pipeline for f_1 by X_3 and the pipeline for f_4 by X_2. Two new uniform variables X_1 and X_4 replace affine terms $X(i,i+k,1)$ and $X(j-k+1,j,1)$.

Figure 2 shows the pipeline for dependencies f_1 and f_3 in the $j-k$ plane for a fixed value of i. The dependent cell **A** is pipelined through the set of cells that depend on it in a linear chain (shown by the straight line). Pipeline f_3 runs from **B** to **C** and is initialized with the value at cell **A** by a uniform dependency (shown by the arc). Pipeline f_1 running horizontally from **C** to **D** is a multistage pipeline and uses the last stage of f_3 for initialization.

A shortest schedule for the Nussinov recurrence is given by $\tau(i,j,k) = -2i+2j-k-1$ where $\lambda = [-2,2,-1]$ and $\alpha = -1$. Figure 3 shows points of a time hyperplane specified by the schedule that are computed in parallel. The total computation time is given by the execution time of the last cell, i.e. $\tau(1,N,1) = 2N-4$ clock cycles.

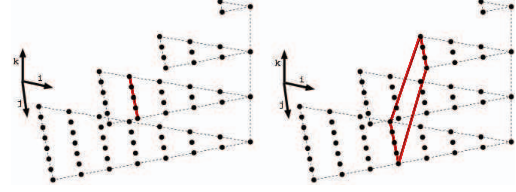

Figure 3: Points in the domain computed at times 7 and 8 respectively. The latter shows the time hyperplane.

3.1.1 Pipelining the Input Sequence

We now deal with the input sequence S used in the score computation of the recurrence. The input variables S_i and S_j are defined over one-dimensional domains but are used in computing the three-dimensional variable X. We therefore align S_i along the $j = 0 \wedge k = 1$ line and S_j along the $i = 0 \wedge k = 1$ line. The affine dependencies introduced are made uniform using nullspace pipelining. Their pipelines P and Q propagate horizontally and vertically across the array and are initialized by the input sequence at the domain boundary $j - i = 1$. The final, transformed uniform recurrence equations are shown below:

$$X(i,j,k) = \max \begin{cases} \delta(P(i,j,k), Q(i,j,k)) & \text{if } j-i=1 \\ \\ X(i+1,j,k) & \\ X(i,j-1,k) & \\ X(i+1,j-1,k) + & \\ \quad \delta(P(i,j,k), Q(i,j,k)) & \text{if } k=1 \\ X(i,j,k+1) & \\ X_1(i,j,k) + X_2(i,j,k) & \\ X_3(i,j,k) + X_4(i,j,k) & \\ \\ X(i,j,k+1), & \\ X_1(i,j,k) + X_2(i,j,k) & \text{otherwise} \\ X_3(i,j,k) + X_4(i,j,k) & \end{cases} \tag{2}$$

$$X_1(i,j,k) = \begin{cases} X_3(i,j,k) & \text{if } j-i=2k \\ X_1(i,j-1,k) & \text{otherwise} \end{cases}$$

$$X_2(i,j,k) = \begin{cases} X(i+2,j,k) & \text{if } k=1 \\ X_2(i+1,j,k-1) & \text{otherwise} \end{cases}$$

$$X_3(i,j,k) = \begin{cases} X(i,j-1,k) & \text{if } k=1 \\ X_3(i,j-1,k-1) & \text{otherwise} \end{cases}$$

$$X_4(i,j,k) = \begin{cases} X_2(i,j,k) & \text{if } j-i=2k \\ X_4(i+1,j,k) & \text{otherwise} \end{cases}$$

$$P(i,j,k) = \begin{cases} S_i & \text{if } j-i=1 \\ P(i,j-1,k) & \text{if } k=1 \end{cases}$$

$$Q(i,j,k) = \begin{cases} S_j & \text{if } j-i=1 \\ Q(i+1,j,k) & \text{if } k=1. \end{cases}$$

3.2. Pipelining Control

Recurrence equations that contain conditional operations (e.g. "if $k = 1$") naively require logic in each PE to check the condition. We can eliminate this extra logic by generating a pipeline from a basis vector for the lattice generated by points satisfying the condition. The pipeline transmits a single bit that is true whenever the condition is satisfied in a cell and is initialized when the boundary cell is evaluated.

The two conditions in the Nussinov recurrence that must be pipelined are $j - i = 2k$ and $k = 1$. The condition $j - i = 2k$ may use either $[0, -2, -1]$ or $[1, -1, -1]$ as valid basis vectors for its pipeline. We use the latter, which guarantees nearest-neighbor communication. Note that the condition is valid only for even diagonals. At the even diagonal border $j - i = 2$, the control pipeline is initialized to 1 and then pipelined to subsequent even diagonals. The control pipeline is absent at odd diagonals in the array.

When aggregating along the k-axis, we need to substitute the term $X(i, j, k + 1)$ by zero at $k = \lfloor (j - i)/2 \rfloor$. We use a pipeline similar to the one above except that both odd and even diagonal pipelines are initialized to 1. For the condition $k = 1$, we use the basis vector $[1, 0, 0]$.

4. Processor Allocation and PE Design

Once a schedule has been found, the allocation function is computed. We project the domain of computation for the Nussinov recurrence in two directions to derive two systolic arrays, called GKT and GJQ (after the authors of the corresponding arrays for string parenthesization). The GKT array, similar to the array of Guibas et al. [4], projects along the direction $[0, 0, -1]$ with $\pi(i, j, k) = [i, j]$. The GJQ array, similar to that of Gachet et al. [3], projects along the direction $[-1, 0, 0]$ with $\pi(i, j, k) = [j, k]$. Both arrays have the same fixed latency of $2N - 4$ clock cycles.

4.1. GKT array

We can compute the GTK array's structure by combining the dependencies of Recurrence 2 with the processor allocation function $\pi(i, j, k) = [i, j]$. The latency of each wire is computed as λb_i, where λ is the linear part of the timing function τ and b_i is the constant part of the wire's corresponding data dependency. The number of cells required for a sequence of size N with the GKT projection is $N(N + 1)/2$.

The control pipeline for condition $j - i = 2k$ maps to the interconnection $[1, -1]$, which moves data diagonally across the array with latency 3. The pipeline for condition $k = 1$ maps to $[1, 0]$, which moves data vertically with latency 2. The interconnections and their latencies are shown entering a single PE of the array in Figure 4. Because the PE registers its output, there is one fewer register on each wire than the wire's latency.

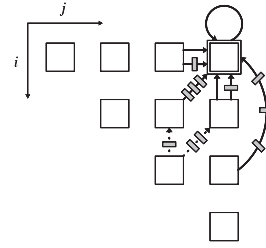

Figure 4: Projection along direction [0, 0, -1] yields the GKT array. Data pipelines are represented as interconnections from neighboring cells with registers realizing the required delays. Control pipelines are shown by dashed lines.

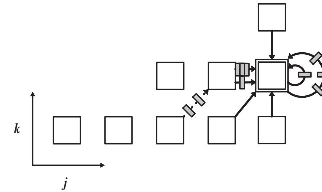

Figure 5: Projection along direction [-1, 0, 0] yields the GJQ array.

To detect the final result $X(1, N, 1)$, we may either use a counter to count up to the latency (execution time) of the target cell or use control pipelines to detect the conditions $i = 1 \wedge j = N \wedge k = 1$. The first two conditions are always satisfied at PE $[1, N]$; for the condition $k = 1$, we use its existing control pipeline.

4.2. GJQ array

The structure of the GJQ array, shown in Figure 5, derives from the allocation function $\pi(i, j, k) = [j, k]$. The control pipeline for condition $j - i = 2k$ maps to the diagonal interconnection $[-1, -1]$ and has a latency of 3. The condition $k = 1$ is always true for all PEs $[j, 1]$, so its pipeline can be discarded.

Cells on the diagonals $j - i = 2$ and $j - i = 3$ have dependencies that lie on diagonal $j - i = 1$, which is initialized from the input. In the GKT projection, the allocation function maps each diagonal's cells to a distinct PE. This naturally maps the dependencies to spatially distributed links that can be individually wired to the input. In contrast, GJQ maps cells on both diagonals to the same set of PEs. We use a multiplexer to select initial values for different dependencies transferred over the same link at different times.

In the GJQ array, the target value is computed by PE $[N, 1]$. We introduce a new control signal for the condition $i = 1$, which is initialized at PE $[2, 1]$ and propagates left to right with a latency of 2 through the bottom row of PEs.

This signal is used to detect the result at PE $[N, 1]$.

The number of PEs required for a sequence of size N with GJQ is $N/2(N/2+1)$, half the number required by the GKT projection. Another advantage of GJQ over GKT is the former's lack of long range interconnections ($[i+2, j]$), which causes fewer routing constraints.

4.3. PE Precision and Smaller Inputs

The width of data paths in the array depends on the precision of the scores required in the Nussinov computation. Increased precision uses larger bit widths, which use more hardware resources and increase the critical path delay. The Nussinov recurrence uses a simple scoring scheme of $+1$ for each base pair. For a sequence of length N, the maximum number of base pairs possible is $N/2$. We therefore limit the bit width of the PEs to $\log_2(N/2+1)$.

The Nussinov arrays are designed for input sequences of length exactly N, determined at compile time. Given this length, the target score is always computed by a fixed PE. Input sequences of length $< N$ are handled as a special case. Suppose we have a sequence of length $m < N$ whose final score is computed at PE $\pi(1, m, 1)$. When calculating the score of a sequence of length $m+1$, by the principle of optimality there exists at least one case in the dynamic programming recurrence that has a direct or indirect dependency on $X(1, m, 1)$. We can ensure that this value is always the optimal case by evaluating δ to zero whenever $i, j > m$. To achieve this, we pad the input sequence by $N - m$ special "unpairable" characters that yield a zero score when paired with anything. We can then show inductively that cell $X(1, N, 1)$'s value equals that of $X(1, m, 1)$.

5. Results

We have implemented our designs in VHDL, verified them in simulation, and synthesized them post place-and-route. The architecture of the Nussinov engine contains a sequence module that buffers input sequences and transfers each sequence to the systolic array every $2N - 4$ clock cycles (the fixed latency of both arrays).

We first built the two arrays on the Virtex-II 6000-6 FPGA to process as large an input sequence as possible. Both arrays were built with a PE data width of 5 bits and clocked at 70 MHz. We report the total number of slices used (including area for a PCI-X interface module) in Table 1. The GKT array can process a sequence of length 34 using 78% of slices on the FPGA. The GJQ array, however, can process sequences of length 62, $1.8\times$ longer.

We next built the GJQ array with varying cell precisions as reported in Table 2. Increasing the data width noticeably increases area and so reduces the number of PEs that fit on the FPGA. Our design's critical path is also sensitive to the logic delays caused by the longer combinational path of the larger bit widths. More complex RNA secondary structure algorithms will likely need a precision of at least 16 bits, requiring a larger FPGA.

Table 1: Comparison of GKT and GJQ arrays.

Projection	N	SLICES
GKT	34	26,482 (78%)
GJQ	62	27,810 (82%)

Table 2: Variation of resource usage with PE precision.

Precision	N	SLICES	Frequency (MHz)
5	62	27,810 (82%)	70
8	50	28,879 (85%)	70
16	36	31,115 (92%)	60

To provide a baseline for estimating the speedup of our hardware, we wrote an optimized version of the Nussinov algorithm (Recurrence 1) in C. The software was compiled with gcc 4.1.2 with flags -O3 -march=nocona -fomit-frame-pointer. The baseline system was an Intel Core 2 Duo CPU running at 3 GHz. We measured time to compute the Nussinov score on a single core (I/O time not included). Running time was averaged over 1000 randomly generated sequences for each case. The hardware time ($2N - 4$ clock cycles) was estimated at a clock rate of 70 MHz.

Table 3 shows the expected speedups of the Nussinov arrays for various sequence lengths. The GKT and GJQ arrays have the same execution time of $2N-4$ clock cycles and hence the same speedups. Our GKT implementation can process a sequence of at most 34 residues on the Virtex-II 6000 FPGA and achieves an estimated speedup of $14\times$. Our larger GJQ implementation achieves a more than two-fold estimated speedup over the GKT array. Newer FPGAs will support even longer sequences; for example, we built a GJQ array handling sequences of length 100 on a Virtex-4 LX160 FPGA. Speedups increase significantly for arrays handling longer sequence lengths because the array runs in time $O(N)$ rather than the software's $O(N^3)$.

6. Partitioning Overview

Our GKT array can process sequences of length 34, but RNA folding problems of practical interest typically have lengths of 100-200. We briefly describe the LPGS [8] (Locally Parallel, Globally Sequential) approach to partition-

Table 3: Speedup of Nussinov array vs. modern x86 CPU.

N	S/W time (ns)	H/W time (ns)	Speedup
34	$13,672$	915	$14\times$
62	$68,021$	$1,715$	$39\times$
100	$261,254$	$2,800$	$93\times$

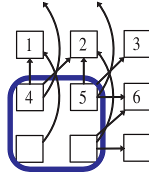

Figure 6: Square LPGS partitioning of the GKT array with border interconnections.

ing that we plan to take in the future. Here the virtual array is partitioned into serially computed blocks, with points within a block computed in parallel. We outline some challenges in partitioning the GKT array and suggest solutions.

Consider the square partition of the GKT array with edges parallel to the $i = 1$ and $j = 1$ axes shown in Figure 6. Here global execution is from left to right, bottom to top of the array. Cell values from every link of a PE at the top and right border are written into a FIFO and read at the bottom and left borders. This step can be implemented using block RAMs on FPGAs that are otherwise largely unused in our design.

Links parallel to block edges have sink PEs located in the same block. If this is not the case, it can increase the complexity of control. For example, the diagonal links from PE 4 to 2 and from PE 5 to 3 lead to two distinct blocks. Our solution is to convert non-parallel edges to a sequence of parallel edges. Here, the link from PE 5 to 3 is split into two links: 5 to 6 and 6 to 3.

The number of cell values written by a border PE depends on the number of points computed on the k-axis, which varies according to the position of the block in the computation domain. We can, however, use the control signals for the conditions $k = \lfloor (j - i)/2 \rfloor$ to start writing, and $k = 1$ to stop writing cell values from a border PE.

Partitioning an array introduces inefficiencies due to lost parallelism; however, we still expect to see significant speedups for large input sequences.

7. Conclusion

We have produced two systolic array designs for the Nussinov RNA folding algorithm. While the core designs draw on classic results for string parenthesization, we have elaborated these designs into fully realized FPGA implementations and analyzed their performance. Our designs achieve a 39× speedup over a recent x86-family CPU when implemented on a Virtex-II 6000 FPGA. The current designs are limited to RNAs of 30-60 bases—enough to find simple stem-loop structures—but we plan to extend to longer RNAs by partitioning. Our realization of the Nussinov algorithm is an important starting point for future designs that more closely approximate the detailed energy models used by the Zuker algorithm.

Acknowledgements

This work was supported by NIH/NHGRI award 1 R42 HG003225-01. R. D. Chamberlain is a principal in BECS Technology, Inc.

References

[1] R. T. Batey. Structures of regulatory elements in mRNAs. *Current Opinion in Structural Biology*, 16:299–306, 2006.

[2] E. A. Dougherty and J. A. Doudna. Ribozyme structures and mechanisms. *Annual Review of Biophysics and Biomolecular Structure*, 30:457–75, 2001.

[3] P. Gachet, B. Joinnault, and P. Quinton. Synthesizing systolic arrays using DIASTOL. In *International Workshop on Systolic Arrays*, pages 25–36, 1986.

[4] L. T. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Proc. Cal. Tech. Conf. VLSI*, pages 509–525, 1979.

[5] G. J. Hannon. RNA interference. *Nature*, 418:244–51, 2002.

[6] M. Höchsmann, T. Töller, R. Giegerich, and S. Kurtz. Local similarity in RNA secondary structures. In *Proc. of the IEEE Bioinformatics Conference*, pages 159–68, 2003.

[7] D. Lavenier, P. Quinton, and S. Rajopadhye. Advanced Systolic Design, in Digital Signal Processing for Multimedia Systems, Chapter 23, Parhi and Nishitani Eds, March 1999.

[8] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. Comput.*, 35(1):1–12, 1986.

[9] J. M. Moscola. *Techniques for hardware-accelerated parsing for network and bioinformatic applications*. PhD thesis, Washington University in St. Louis, St. Louis, MO. USA. 63130, May 2008.

[10] T. Mourier et al. Genome-wide discovery and verification of novel structured RNAs in Plasmodium falciparum. *Genome Research*, 18:281–92, 2008.

[11] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman. Algorithms for loop matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, July, 1978.

[12] G. Pavesi, G. Mauri, and G. Pesole. An algorithm for finding conserved secondary structure motifs in unaligned RNA sequences. *Journal of Computer Science and Technology*, 19:2–12, 2004.

[13] S. V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3(2):88–105, 1989.

[14] Y. Sakakibara, M. Brown, R. Hughey, I. S. Mian, K. Sjolander, R. C. Underwood, and D. Haussler. Stochastic context-free grammars for tRNA modeling. *Nucleic Acids Research*, 22:5112–20, 1994.

[15] G. Tan, L. Xu, S. Feng, and N. Sun. An experimental study of optimizing bioinformatics applications. In *Proc. Int'l Parallel and Distributed Processing Symposium*, 2006.

[16] M. Zuker. Computer prediction of RNA structure. *Methods in Enzymology*, 180:262–88, 1989.