Rapid RNA Folding: Analysis and Acceleration of the Zuker Recurrence

Arpith C. Jacob
Jeremy D. Buhler
Roger D. Chamberlain

Arpith C. Jacob, Jeremy D. Buhler, and Roger D. Chamberlain, “Rapid RNA Folding: Analysis and Acceleration of the Zuker Recurrence,” in *Proc. of 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2010, pp. 87-94.

Dept. of Computer Science and Engineering
Washington University in St. Louis

Rapid RNA Folding: Analysis and Acceleration of the Zuker Recurrence

Arpith C. Jacob*, Jeremy D. Buhler*, and Roger D. Chamberlain*†
*Department of Computer Science and Engineering, Washington University in St. Louis
†BECS Technology, Inc., St. Louis, Missouri
Email: {jarpith,jbuhler,roger}@wustl.edu

***Abstract*—RNA folding is a compute-intensive task that lies at the core of search applications in bioinformatics such as RNAfold and UNAFold. In this work, we analyze the Zuker RNA folding algorithm, which is challenging to accelerate because it is resource intensive and has a large number of variable-length dependencies. We use a technique of Lyngsø to rewrite the recurrence in a form that makes polyhedral analysis more effective and use data pipelining and tiling to generate a hardware-friendly implementation. Compared to earlier work, processors in our array are more efficient and use fewer logic and memory resources.**

We implemented our array on a Xilinx Virtex 4 LX100-12 FPGA and experimentally verified a $103\times$ speedup over a single core of a 3 GHz Intel Core 2 Duo CPU. The accelerator is also $17\times$ faster than a recent Zuker implementation on a Virtex 4 LX200-11 FPGA and $12\times$ and $6\times$ faster respectively than an Nvidia Tesla C870 and GTX280 GPU. We conclude with a number of lessons in using FPGAs to implement arrays after polyhedral analysis. We advocate using polyhedral analysis to accelerate other dynamic programming recurrences in computational biology.

***Keywords*-RNA secondary structure; Zuker; polyhedral model; FPGA**

An RNA is a single-stranded molecule that may be represented by a linear sequence of nucleic acids, or *bases*, from the alphabet $\{A, C, G, U\}$. RNAs implement diverse functions in organisms, including initiation of translation, catalysis of reactions, regulation of genes, and targeted suppression of transcription and translation. An RNA's function is determined by the shape it folds into as complementary bases on the sequence pair up (mostly A-U and C-G) to form its so-called *secondary structure*.

Since the experimental determination of an RNA's secondary structure is time-consuming, biologists use computer programs to predict the structure of single RNA molecules. These algorithms find a folded structure with minimum free energy using empirical models. The first such algorithm was due to Nussinov [1]; it finds a structure with the largest number of base pairs. The Zuker dynamic programming algorithm [2] predicts the structure using more detailed and accurate energy models. Figure 1 shows a folded RNA molecule, highlighting the types of structural features that are explicitly modeled by the Zuker algorithm.

Biologists generally fold RNAs of only a few hundred bases because the structure of large molecules cannot be predicted accurately using pure folding approaches. However, many important applications, including RNA motif finding [3] and the search for functional RNAs in genomic DNA [4], may fold millions of short RNAs. Folding algorithms require time at least cubic in the length of the sequence, so high-throughput folding is a major computational challenge. Consequently, researchers have parallelized folding algorithms using both multi-core general-purpose processors [5] and specialized architectures on Field-Programmable Gate Arrays (FPGAs) [6] and Graphics Processing Units (GPUs) [7].

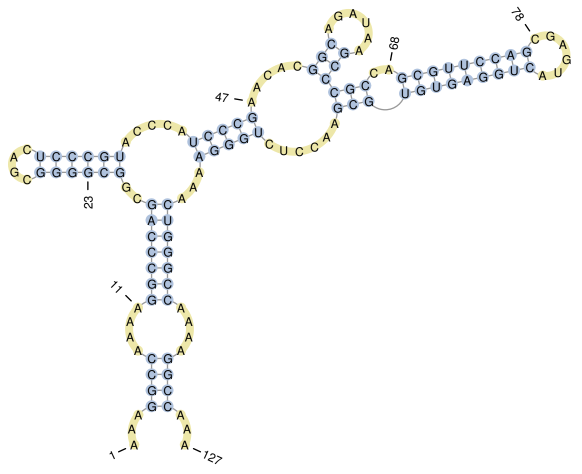

Figure 1. An example of an RNA folded into its secondary structure with free energy -53.90 kcal/mol. Types of structural features modeled by the Zuker folding algorithm include: dangling ends (1), internal loop (11), stack (23), multi loop (47), bulge (68) and hairpin loop (78).

This work makes two main contributions. First, we analyze the Zuker algorithm to build an application-specific FPGA accelerator. Our experiments show that we are able to fold RNAs of length 273 bases on a Xilinx Virtex 4 LX100-12 FPGA $103\times$ faster than a single core of a 3 GHz Intel Core 2 Duo processor. Our array is also an order of magnitude faster than existing FPGA and GPU accelerators built with the same generation of process technology.

Second, our Zuker implementation illustrates use of the *polyhedral model* [8] to analyze dynamic programming algorithms in computational biology. This model is a powerful theoretical framework that can represent and analyze regular loop programs with static dependencies. Loop transformations such as interchange, skewing, unrolling, and tiling can be investigated systematically to expose far more parallelism than *ad hoc* methods.

Many important algorithms in computational biology implement compute-intensive dynamic programming recurrences that can be analyzed with the polyhedral model. Examples include Smith-Waterman pairwise sequence alignment and motif search with profile hidden Markov models. Derrien and Quinton [9] used polyhedral methods to powerful effect in building a hardware array for the latter task that outperformed arrays built using extant approaches. However, both these algorithms have only quadratic runtime, use simpler dependence structures, and exhibit more regular computations in the loop body than does Zuker.

Based on our experience with the Zuker algorithm, we outline general techniques useful for synthesis of efficient FPGA-based accelerators after polyhedral analysis. We believe our approach holds promise in speeding up the many data- and compute-intensive dynamic programming algorithms in computational biology.

I. The Zuker Recurrence

We now describe a simplified version of the RNA folding algorithm implemented in the Vienna RNA [10] and UNAFold [11] packages. We have omitted some details for brevity, including modeling special-case loops and dangling ends. Our accelerator, however, implements these computations and yields results identical to the `fill_arrays` function of Vienna RNA version 1.8.4.

Given a sequence S of length N, the Zuker algorithm recursively computes three data variables W, V, and VBI defined over the domain $\mathcal{D} = \{\ 1 \le i \le N; i \le j \le N\ \}$. Supporting energy functions computed via table lookup are shown in lower case.

$$W(i,j) = \min \begin{cases} W(i+1,j)+b \\ W(i,j-1)+b \\ V(i,j)+\delta(S_i,S_j) \\ \min\limits_{i<k<j} \{W(i,k)+W(k+1,j)\} \end{cases} \tag{1}$$

$$V(i,j) = \min \begin{cases} \infty & \textit{if } (S_i,S_j) \textit{ is not a base pair} \\ eh(i,j) & \textit{otherwise} \\ V(i+1,j-1)+es(i,j) \\ VBI(i,j) \\ \min\limits_{i<k<j-1} \{W(i+1,k)+W(k+1,j-1)\}+c \end{cases} \tag{2}$$

$$VBI(i,j) = \min_{i<i'<j'<j} \{\ V(i',j')+ebi(i,j,i',j')\ \}. \tag{3}$$

$V(i,j)$ is the energy of the optimal structure formed by subsequence $S_{i...j}$, closed by the base pair (S_i, S_j). The two bases at i and j may close a hairpin loop (eh), form part of a stack (es), close an internal loop or bulge (VBI), or be part of a multi loop (W). See Figure 1 for examples of these features. The most likely scenario is selected using the minimization operation in Equation 2. If these two bases do not pair, however, the energy score is set to infinity.

Equation 3 calculates the scores of internal loops and bulges and is subject to the constraint $i' - i + j - j' > 2$.

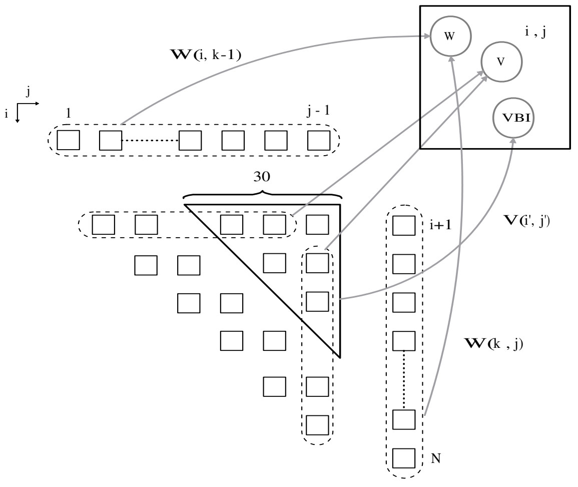

Figure 2. Long-range dependencies for the cell (i, j).

This equation also determines the time complexity of the algorithm, which is quartic in the length of the sequence. However, large internal loops are uncommon in nature, so implementations of Zuker sacrifice accuracy for speed by limiting the size of internal loops to at most 30, reducing the overall time complexity to $\Theta(N^3)$, albeit with a large constant factor.

The computation domain for the recurrences is a triangle, as shown in Figure 2. Long-range dependencies of the enlarged cell (i, j) are depicted in dashed boxes. Dependencies for VBI are in the bold triangle, whose base and height are each at most 30 cells. Note that all cells (i, j) on a given anti-diagonal line $j - i = \textit{constant}$ can be computed in parallel.

Finally, the minimum free energy $F(j)$ of the folded RNA subsequence $1 \dots j$ is given by

$$F(j) = \min \begin{cases} F(j-1) \\ \min\limits_{1 \le i < j} \{V(i,j)+F(i-1)\}. \end{cases} \tag{4}$$

$F(N)$ is then the free-energy score of the entire molecule. We can perform a traceback procedure on this recurrence to find the optimal structure; this traceback requires negligible compute resources, so we do not accelerate it here.

II. Related Work

Recently, significant effort has been spent in accelerating the Zuker algorithm on multi-core processors, graphics accelerators, and FPGAs.

GTfold [5] is an OpenMP shared memory implementation of Zuker on an IBM P5-570 server with 16 dual-core 1.9 GHz CPUs. GTfold is able to fold an RNA of length 9781 bases 19× faster than a sequential implementation. The time to fold 11 RNAs, each over 7000 bases, was reduced from about 3 months to under 9 minutes. The weakness of this approach is that the sequences must be large enough that the communication time between cores is minimal compared to the computation time. For short sequences of a few hundred bases, which encompass the majority of biologists' workload, there is too little work to distribute among the 32 cores. Modern RNA folding algorithms have poor accuracy when the sequence is larger than 100-200 bases [12] and are considered unreliable for sequences several thousand bases in length. For example, the referenced study shows a median accuracy of 80% for sequences of average length 120, but accuracy falls to 41% for sequences of length 1000-3000. In contrast to the GTfold approach, one may accelerate the folding of a database of short sequences by distributing the workload among independent executions on 32 cores to achieve a near $32\times$ speedup.

Rizk and Lavenier [7] used an NVIDIA GTX280 GPU to search for microRNAs of length 120 bases, accelerating Zuker 17-fold versus one core of a Xeon 2.66 GHz workstation. This GPU has 30 multiprocessors, each a SIMD unit of eight 32-bit processors. The authors use multiple levels of parallelism: across several sequences, across cells on a diagonal, and across independent threads for the computation for each variable in the algorithm. Memory access is the bottleneck in their implementation — there is too little low-latency memory for fast access to the data variables and energy parameters.

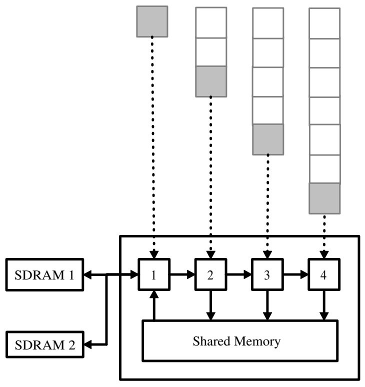

Figure 3. An FPGA array by Dou et al. [6] to accelerate Zuker.

The special-purpose FPGA accelerator sketched in Figure 3 was suggested by Dou et al. [6]. It uses a linear array of processing elements (PEs) connected to SDRAM memories through an on-chip cache. Each PE requires ten block RAM memories to store a copy of the energy parameters, despite extensive optimizations done by the authors. Due to this heavy memory usage, V and W elements that are computed in the array must be stored in two SDRAM memories with a cache to support efficient reuse. The first PE requires direct access to SDRAM, but all other processors in the array need only left-to-right communication. Each PE is assigned one column of the computation domain, and the entire array computes cells along the same anti-diagonal simultaneously. Computation proceeds from bottom to top of a column, after which there is a synchronization phase when the results are written back to memory; then the next block of columns is processed.

The authors' implementation on a Xilinx XC4VLX200-11 is memory-limited to 16 PEs clocking at 135 MHz. They achieve a $9.8\times$ speedup for 145-base sequences compared to a Pentium 4 2.6 GHz CPU. If a modern processor core were used as the baseline, we expect the speedup to halve.

Dou et al. mention a number of challenges faced in accelerating Zuker, including the number of dependence terms in computing the internal loop energy and the large number of energy parameters required. In particular, storage requirements for empirically computed energy tables of all possible hairpin and internal loops of certain sizes present a challenge to any meaningful acceleration. Variable dependence distances make it difficult to find task assignments that are balanced among PEs, and long-range dependencies make it difficult to efficiently schedule the movement of V and W variables to the required PEs. One solution to the latter problem is to store these variables in a central cache until such time as they are required. However, this strategy may affect scalability if there are a larger number of PEs in the array. The large number of small-granularity access operations also make it difficult to optimize scheduling for efficient external memory access. Furthermore, the long latency for off-chip memory access degrades performance.

In our previous work, we have accelerated the far simpler Nussinov RNA folding algorithm [13], which uses the dependence structure of Equation 1 alone. We were able to analyze the recurrence using the polyhedral framework to build space-efficient arrays for 62-base RNAs that are $39\times$ faster than a software implementation running on a single core of a 3 GHz Intel Core 2 Duo CPU. We also used nullspace pipelining [14] to convert the long-range dependencies in the fourth term of Equation 1 to nearest-neighbor communication, thus avoiding a shared memory cache. We build on these ideas to accelerate the Zuker algorithm, which remains a significant challenge.

III. Recurrence-to-1D-Array Mapping

Given a system of recurrences, we would like to derive a linear processor and time mapping that describes a 1D array. We will use tools from polyhedral theory [8], which we summarize in this section.

Let an n-dimensional *iteration vector* $z = [i_1, \ldots, i_n]$ identify the indices of a recurrence. We first extract dependencies from a recurrence's iteration vectors. As an example, in Equation 1, because $W(i,j)$ requires $W(i+1,j)$ for its computation, we say that $[1,0]$ is a dependency. An array for a recurrence can be described by an *allocation function*, which maps iteration vectors z onto an (n-1)-dimensional grid of "virtual" PEs, and a *schedule* that gives the execution time of z.

We use a linear function $\pi(z) = [\pi_1(z), \ldots, \pi_{n-1}(z)]$ for the allocation, which induces the "virtual" PE space $\mathcal{V} =$

$\{ 1 \le i_1 \le N_1; \; \dots \; ; 1 \le i_{n-1} \le N_{n-1}\}$. We are only interested in unpartitioned 1D arrays, so we select one of the elements of the allocation, say i_1, as the physically realized PE array. The iteration vectors of the other virtual PEs must be executed by this 1D array.

A linear schedule $\tau(z)$ is constructed to respect dependencies, i.e., if z depends on some other iteration vector y, it must be that $\tau(z) > \tau(y)$. A schedule must also assign the iteration vectors of the virtual PEs on the 1D array without creating a conflict: there must be no two vectors z and y with $\tau(z) = \tau(y)$ and $\pi_1(z) = \pi_1(y)$.

Darte et al. [8] give a constructive procedure to create such schedules for the 1D array, which are of the form (up to a permutation of the iteration vector indices)

$$\tau(i_1, \dots, i_n) = a_1 i_1 + a_2 i_2 + a_3 N_2 i_3 + \dots + a_n N_2 \cdots N_{n-1} i_n \qquad (5)$$

where $a_n = \pm 1$, and the greatest common divisor of a_k and N_k is 1.

Notice that this description assumes that the virtual processor space is a rectangular parallelepiped. In contrast, the Zuker recurrence instantiates a triangular domain. We simply create a rectangular bounding box for this domain and assume the PEs assign ∞ to the data variables when computation is outside the triangle. This does, however, result in a suboptimal schedule.

In the next section, we apply high-level transformations to simplify the complexity of the Zuker recurrences and remove long-range dependencies in order to support the generation of efficient 1D arrays.

IV. Transformations for Increased Parallelism

First, we simplify the reduction operations in the Zuker recurrences. Notice that the fourth terms in Equations 1 and 2 are similar. Let the final term in Equation 1 be defined as

$$T(i,j) = \min_{i<k<j} \{W(i,k) + W(k+1,j)\} \; . \qquad (6)$$

We can rewrite the final term in Equation 2 in terms of T:

$$\min \left\{ \begin{array}{l} W(i+1,i+1) + W(i+2,j-1) + c \\ T(i+1,j-1) + c \; . \end{array} \right. \qquad (7)$$

$W(i{+}1,i{+}1)$ is always ∞, so we can ignore it, allowing us to replace the reduction in Equation 2 with $T(i{+}1,j{-}1){+}c$.

A. *Simplifying the Internal Loop Computation*

The internal loop and bulge structure computation in variable *VBI* presents two challenges. First, every cell (i,j) depends on a very large triangular section of cells. Second, because the four indices (i,i',j,j') do not define a rectangular domain, an array generated using the linear polyhedral framework will have a suboptimal schedule.

Fortunately there exists an algorithmic technique to reduce the complexity of *VBI without* affecting its result. Lyngsø [15] observed that the energy function *ebi* is not arbitrary but depends on the size of a loop, enabling him

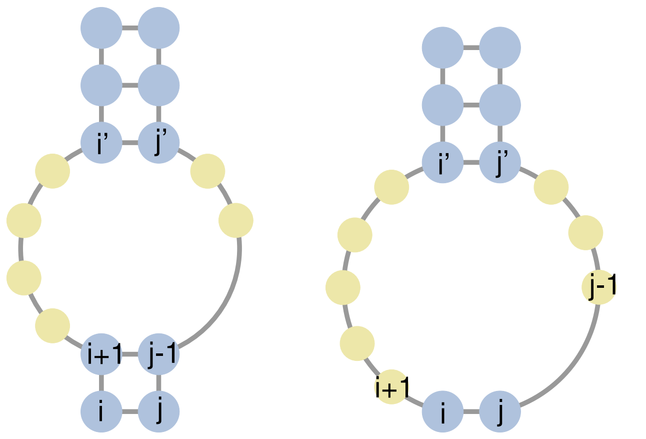

Figure 4. Difference in internal loop energy as the exterior base pair changes from $(i{+}1,j{-}1)$ to (i,j). Using Equation 8 we see that the energy for the second loop is $ebi(i{+}1,j{-}1,i',j') - ebistacking(S_{i+1},S_{j-1}) + ebistacking(S_i,S_j) - ebisize(l-2) + ebisize(l)$. Here $l = i' - i + j - j' - 4$ is the loop size.

to simplify the internal loop computation to cubic time complexity. Lyngsø's optimization did not yield a speedup in software because standard implementations already limit the size of internal loops to 30; in fact, there was a slowdown due to additional variables introduced during the transformation. Here we use the Lyngsø transformation to reduce the complexity of the recurrences to make them more amenable to hardware acceleration.

We now give a brief overview of the internal loop energy.[1] We define two distinct energy functions: *ebb* for bulges and *ebi* for internal loops. The energy function *ebi* can be split into contributions from three parameters: stacking energies of interior and exterior base pairs, the size of the loop, and the asymmetry of the loop. The energy function for bulges is similarly defined, though without an asymmetry component.

Let $k = i' - i + j - j' - 2$ be the size of the internal loop. The energy function *ebi* is expressed as

$$\begin{array}{l} ebi(i,j,i',j') = ebistacking(S_i,S_j) + ebistacking(S_{i'},S_{j'}) + \\ \quad ebisize(k) + ebiasymmetry(|(i'-i-1)-(j-j'-1)|) \; . \end{array} \qquad (8)$$

Consider the internal loop closed by base pairs $(S_{i'},S_{j'})$ and (S_{i+1},S_{j-1}) in Figure 4. We say the loop is *lopsided* because the unpaired region on the left has two more bases than the region on the right. The asymmetry energy component is a penalty dependent on the loop's lopsidedness.

Because *ebisize* is constant for a fixed loop size, we can reformulate the internal loop calculation to aggregate all dependencies $V(i',j')$ that have $i'{-}i{+}j{-}j' = constant$. We build from smaller to larger subproblems that have identical lopsidedness, i.e., whenever i is increased by one base, j is also decreased by one. This idea is illustrated in Figure 4.

Let $VBI'(i,j,k)$ and $VBI(i,j,k)$ be the score of the

[1]A detailed description of all energy functions is available online at http://www.bioinfo.rpi.edu/zukerm/seqanal/.

best internal loop of size k and $\geq k$ respectively that have exterior base pair (S_i, S_j). We use the following recurrence to compute the internal loop energies of all loops of size ≥ 5. For smaller loops, we do a lookup into an experimentally computed score table provided by Vienna RNA (not shown).

$$VBI(i,j,k) = \min \begin{cases} VBI(i,j,k+1) & \text{if } k \geq 5 \\ VBI'(i+1,j-1,k-2)+ & \\ \quad ebistacking(S_i,S_j)- & \\ \quad ebistacking(S_{i+1},S_{j-1})+ & \\ \quad ebisize(k) - ebisize(k-2) & \end{cases}$$

$$VBI'(i,j,k) = \min \begin{cases} V(i+1,j-k-1)+ & \text{if } k = 1 \\ \quad ebi(i,j,i+1,j-k-1) & \\ V(i+k+1,j-1)+ & \\ \quad ebi(i,j,i+k+1,j-1) & \\ & \\ V(i+1,j-k-1)+ & \text{if } k = 2 \\ \quad ebi(i,j,i+1,j-k-1) & \\ V(i+k+1,j-1)+ & \\ \quad ebi(i,j,i+k+1,j-1) & \\ V(i+2,j-2)+ & \\ \quad ebi(i,j,i+2,j-2) & \\ & \\ VBI'(i+1,j-1,k-2)+ & \text{if } k \geq 3 \\ \quad ebistacking(S_i,S_j)- & \\ \quad ebistacking(S_{i+1},S_{j-1})+ & \\ \quad ebisize(k) - ebisize(k-2) & \\ V(i+1,j-k-1)+ & \\ \quad ebi(i,j,i+1,j-k-1) & \\ V(i+k+1,j-1)+ & \\ \quad ebi(i,j,i+k+1,j-1). & \end{cases}$$

Note how, when aggregating at $VBI'(i,j,k)$ from $VBI'(i+1,j-1,k-2)$, we subtract the energy contribution of $ebisize(k-2)$ and add the contribution of $ebisize(k)$. This is possible because the energy components are the same for all interior pairs $V(i',j')$ $s.t.$ $k = i'-i+j-j'-2$, forming loops of the same size. Lopsidedness also remains unchanged since an extra base is added to both sides of the loop.

We can further simplify this recurrence by observing that the $ebistacking$ term is independent of k; we can add this contribution at $k = 1$. The terms $V(i+1,j-k-1)$ and $V(i+k+1,j-1)$ represent long-range dependencies that must be pipelined for an efficient array synthesis.

Similarly, $VBB(i,j,k)$ is the score of the best bulge loop of size $\geq k$; we have omitted details due to space limits.

B. Pipelining the Zuker algorithm

As our final transformation, we pipeline long-range dependencies using nullspace pipelining [14]. This includes W terms to compute T in Equation 6 and V terms to compute VBI' and VBB from the previous section. An application of nullspace pipelining is described in previous work [13].

The transformed Zuker recurrence equations are defined over the domain $\mathcal{D} = \{\ 1 \leq i \leq N; i \leq j \leq N; 1 \leq k \leq \min\{30, j-i-2, \lfloor \frac{j-i}{2} \rfloor\}\ \}$. In the interest of clarity, we have replaced dependencies on the sequence variables PA and PB in the energy functions by "..." The final system of recurrences for Zuker is shown below.

$$W(i,j,k) = \min \begin{cases} W(i+1,j,k)+b & \text{if } k=1 \\ W(i,j-1,k)+b & \\ V(i,j,k)+\delta(\ldots) & \\ T(i,j,k) & \end{cases} \quad (9)$$

$$V(i,j,k) = \min \begin{cases} eh(\ldots) & \text{if } k=1 \\ V(i+1,j-1,k)+es(\ldots) & \\ VBB(i,j,k) & \\ VBI(i,j,k) & \\ T(i+1,j-1,k)+c & \end{cases} \quad (10)$$

$$PA(i,j,k) = \begin{cases} S_i & \text{if } j-i=1 \\ PA(i,j-1,k) & \text{if } k=1 \end{cases} \quad (11)$$

$$PB(i,j,k) = \begin{cases} S_j & \text{if } j-i=1 \\ PB(i+1,j,k) & \text{if } k=1 \end{cases} \quad (12)$$

$$T(i,j,k) = \min \begin{cases} T(i,j,k+1) & \text{if } 2k \leq j-i \\ PW_1(i,j,k)+PW_2(i,j,k) & \\ PW_3(i,j,k)+PW_4(i,j,k) & \end{cases} \quad (13)$$

$$PW_1(i,j,k) = \begin{cases} PW_3(i,j,k) & \text{if } 2k = j-i \\ PW_1(i,j-1,k) & \text{if } 2k < j-i \end{cases} \quad (14)$$

$$PW_2(i,j,k) = \begin{cases} W(i+2,j,k) & \text{if } k=1 \\ PW_2(i+1,j,k-1) & \text{if } 2k \leq j-i \end{cases} \quad (15)$$

$$PW_3(i,j,k) = \begin{cases} W(i,j-1,k) & \text{if } k=1 \\ PW_3(i,j-1,k-1) & \text{if } 2k \leq j-i \end{cases} \quad (16)$$

$$PW_4(i,j,k) = \begin{cases} PW_2(i,j,k) & \text{if } 2k = j-i \\ PW_4(i+1,j,k) & \text{if } 2k < j-i \end{cases} \quad (17)$$

$$VBB(i,j,k) = \min \begin{cases} VBB(i,j,k+1)+ & \text{if } k=1 \\ \quad terminalAUGU(\ldots) & \\ V(i+1,j-2,k)+ & \\ \quad ebbsize(k)+es(\ldots) & \\ V(i+2,j-1,k)+ & \\ \quad ebbsize(k)+es(\ldots) & \\ & \\ VBB(i,j,k+1) & \text{if } k \leq \min\{30, j-i-2\} \\ PVB_1(i,j,k)+ebbsize(k) & \\ PVB_2(i,j,k)+ebbsize(k) & \end{cases} \quad (18)$$

$$PVB_1(i,j,k) = \begin{cases} V(i+1,j-2,k)+ & \text{if } k=1 \\ \quad terminalAUGU(\ldots) & \\ PVB_1(i,j-1,k-1) & \text{if } k \leq \min\{30, j-i-2\} \end{cases} \quad (19)$$

$$PVB_2(i,j,k) = \begin{cases} V(i+2,j-1,k)+ & \text{if } k=1 \\ \quad terminalAUGU(\ldots) & \\ PVB_2(i+1,j,k-1) & \text{if } k \leq \min\{30, j-i-2\} \end{cases} \quad (20)$$

$$VBI(i,j,k) = \min \begin{cases} VBI(i,j,k+1)+ & \text{if } k=1 \\ \quad ebistacking(\ldots) & \\ & \\ VBI(i,j,k+1) & \text{if } 2 \leq k \leq 4 \\ & \\ VBI(i,j,k+1) & \text{if } k \geq 5 \text{ and} \\ VBI'(i+1,j-1,k-2)+ & k \leq \min\{30, \\ \quad ebisize(k) & \quad j-i-2\} \end{cases} \quad (21)$$

$$VBI'(i,j,k) = \min \begin{cases} PVI_1(i,j,k)+ & \text{if } k=1 \\ \quad ebiasymmetry(k) & \\ PVI_2(i,j,k)+ & \\ \quad ebiasymmetry(k) & \\ & \\ PVI_1(i,j,k)+ & \text{if } k=2 \\ \quad ebiasymmetry(k) & \\ PVI_2(i,j,k)+ & \\ \quad ebiasymmetry(k) & \\ V(i+2,j-2,k-1)+ & \\ \quad ebistacking(\ldots) & \\ & \\ VBI'(i+1,j-1,k-2) & \text{if } k \geq 3 \text{ and} \\ PVI_1(i,j,k)+ & k \leq \min\{30, \\ \quad ebiasymmetry(k) & \quad j-i-2\} \\ PVI_2(i,j,k)+ & \\ \quad ebiasymmetry(k) & \end{cases} \quad (22)$$

$$PVI_1(i,j,k) = \begin{cases} V(i+1,j-2,k)+ & \text{if } k=1 \\ \quad ebistacking(\ldots) & \\ PVI_1(i,j-1,k-1) & \text{if } k \leq \min\{30, j-i-2\} \end{cases} \quad (23)$$

$$PVI_2(i,j,k) = \begin{cases} V(i+2,j-1,k)+ & \text{if } k=1 \\ \quad ebistacking(\ldots) & \\ PVI_2(i+1,j,k-1) & \text{if } k \leq \min\{30, j-i-2\}\ . \end{cases} \quad (24)$$

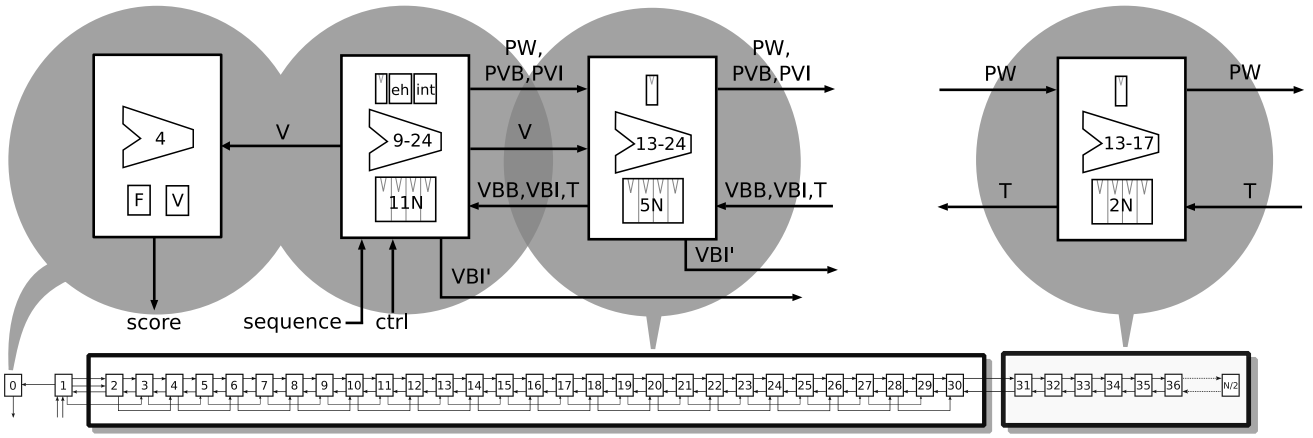

Figure 5. Overview of the 1D Zuker array, which uses four main PE types. Equation numbers (from Section IV-B) computed by a PE are shown in the ALU block; energy functions stored in block RAMs and registers are shown at the top of each PE; and the growth of delay registers as a function of RNA length is shown in the bottom block.

V. 1D Zuker Array

To map our array to physical processing elements, we allocate a 1D set of processors along the recurrence's k dimension — one at each integral point on the k axis. The schedule and allocation of the array are given by

$$\tau(i,j,k) = -2i + Nj - k$$
$$\pi(i,j,k) = k \ .$$

The schedule satisfies all the dependence constraints of the recurrence; to be conflict-free according to Equation 5, the greatest common divisor of N and 2 must be one, i.e., we can only build arrays for sequences of odd length. Even-length sequences are padded up using a special character.

Each processor in the array executes a rectangular domain of points $\{ (i,j) \mid 1 \leq i \leq N; 1 \leq j \leq N \}$. Our array assigns ∞ to the variables when computing outside the domain of the Zuker recurrence. An RNA of length N can be folded in $\tau(1,N,1) - \tau(3,3,1) = N^2 - 3N + 4$ clocks, though useful work is done on only 50% of the clock cycles.

A high level overview of our array is shown in Figure 5. It is instructive to study the characteristics of the array using the system of recurrences in Section IV-B. First, all communication between PEs is limited to the three adjacent neighbors. The PE placed at $k = 1$ is the most resource-intensive in the array, since it is the only one to instantiate the compute-intensive variables V and W defined at $k = 1$. Note that this PE alone holds a copy of the memory-intensive hairpin (eh) and internal loop energy tables used to compute V and W. Similarly, the stacking energy functions are required only at PE 1; since a PE k always processes internal loops of size k, we can distribute one copy of the size and asymmetry energy functions across the entire array. In contrast, the array described in Section II requires a copy of all energy functions at each PE and must implement all computation in Equations 9-24 in every PE in the array, limiting implementation to 16 PEs. This is because the array was placed along the j dimension of the recurrence.

Since internal loops are limited in size to 30, data variables $VBB, PVB_1, PVB_2, VBI, VBI', PVI_1$, and PVI_2 need be calculated only on PEs 1-30. All subsequent PEs in the array implement only the T and PW_1-PW_4 computations, the latter four being simple data pipelines. As a result, there are fewer dependencies in these PEs, so local memory to store intermediate values is greatly reduced.

We use a single processor (PE 0 in Figure 5) to execute Equation 4 sequentially after every column j of $V(i,j)$ has been buffered. This adds a latency of N clocks to the folding computation.

Input sequence data is sent to PE 1 at regular intervals determined by the schedule. The V table values, which are required for the minimum free-energy computation, are always available at PE 1 (arrays instantiated along any other axis would have multiple sources). The minimum free-energy score is available at PE 0.

Finally, as N increases beyond 60, the size of the array equals $\frac{N}{2}$. We can fold an RNA of length N with just half the processors needed by arrays placed along the i or the j axes; moreover, only the cheapest PE needs to be replicated.

VI. FPGA implementation

We have coded the array of the previous section in VHDL, parametrized by the RNA length, targeting a Xilinx Virtex 4 LX100-12 FPGA device. The array can either output V values for traceback in software or send just the minimum free-energy score. We use three bits to represent RNA characters and use 16-bit data paths with saturation arithmetic to avoid overflow. We have also implemented variables to calculate dangling energies and use ten block RAM memories for the empirical loop energy scores in PE 1; all other energies are implemented in distributed memory.

The latency of our implementation is

$$9 + (N^2 - 3N + 4) + N = N^2 - 2N + 13 \qquad (25)$$

clock cycles. Consequently, the input data rate is very low, approximately $3N$ bits per N^2 clocks.

A. Techniques for Synthesis after Polyhedral Analysis

We now give a few techniques that can be generally applied to synthesize optimized arrays on FPGAs after polyhedral analysis. Any dependence in a recurrence has to be sent from the source to sink PE after a fixed delay that is computed using the schedule. For example, dependency $d = [0, -1, 0]$ requires a delay of $|\tau(d)| = N$ clocks. These delays are usually programmed as shift registers with a global reset. Using a reset, however, can adversely impact the resource usage and speed of the implementation on a Xilinx FPGA. If these registers are coded with a reset, synthesis tools use an entire FPGA LUT and its associated flip flop to realize a single bit shift; without reset, a 16-bit shift register can be realized with these same resources. We can always remove reset on dependency delay registers so long as necessary initialization conditions are programmed at the boundaries of the computation domain. This optimization is extremely useful, allowing us to fold a sequence 45 bases longer than is otherwise possible.

Since our design uses very little on-chip memory for tables to support computation, a large fraction of the block RAM memories are unused. We can use these memories as FIFOs to implement the dependency delays. Our PE implementation is parametrized first to use all the on-chip block RAMs and only then to use LUTs to implement delays. With this optimization, we are able to increase by 75 the size of the largest RNA folded.

To achieve an acceptable clock frequency for our design, the computation in a PE must be pipelined. Unfortunately, the technique we have used assumes that an entire iteration is executed in a single clock cycle. The schedule we have selected, however, does instantiate a large number of delays on the majority of dependencies. It therefore becomes possible to pipeline certain paths of the computation in a PE, which synthesis tools can do automatically using the retiming optimization. For dependencies realized using block RAM memories, we always ensure that a small fraction of the delay is still realized using registers. Without retiming, our design synthesized to just 60 MHz; with retiming, the tools achieve a clock frequency of 130 MHz. This clock rate compares well with the manually optimized array mentioned in Section II that runs at 135 MHz on the same FPGA family.

Although the transformations we have applied to the original Zuker algorithm are provably correct, verifying the accuracy of an implementation is a challenging task. We first wrote a C loop implementation of the transformed system of recurrences and confirmed identical output as the `fill_arrays` function in the Vienna RNA package. The VHDL implementation of every energy function was first testbenched on all possible input bases to ease debugging. The entire array was simulated in ModelSim on randomly generated RNAs, and both the V table scores and the minimum free-energy scores were compared to Vienna RNA's output for an exact match. Finally, we validated correctness in hardware on our FPGA system.

VII. Results

We built our array for the Xilinx Virtex 4 LX100-12 FPGA using SmartXplorer to explore different build strategies. The results in this section are all from experiments run in real hardware on our FPGA system. To confirm accuracy of our implementation, we built arrays to fold sequences of length 121, 251, 261, and 273. We folded 10,000 randomly generated RNAs of each size and compared their minimum free-energy scores to those of RNAfold. The two matched exactly. We also folded 1090 sequences of length 99 bases from the Rfam database on our array and confirmed matching scores to those of RNAfold.

For our software baseline, we ran RNAfold on a single core of a 3 GHz Intel Core 2 Duo processor with 4 MB cache. We used gcc 4.4.0 with compilation flags *-O3 -march=nocona -fomit-frame-pointer* and measured only the time spent in computing the Zuker recurrence; traceback and I/O time were excluded.

We built an array with 136 PEs clocked at 130 MHz to fold RNAs of length 273 bases. This is the largest RNA that can be folded by our array on the given FPGA device; it subsumes most of the range of RNA sizes typically folded by biologists. Table VII lists the resource usage of each PE type as reported by the synthesis tool. PE 1, which is similar to the processors in the array by Dou et al., is the most expensive processor in the array. We implemented all of its delays using logic (SRL16); block RAMs are used to implement the energy functions. The empirical loop energy calculation in this PE is the critical path of the design; without it, the array clocks at 170 MHz.

PEs 2 to 30 are less resource-intensive but still use 5 block RAMs to implement the delay registers. We use two implementations for PEs 31 to $\frac{N}{2}$. Version A implements all delay registers using block RAMs. Once we have run out of these memories, version B, the processor that is used the most in our design, implements delay registers using logic. This PE uses just 16% of the logic resources consumed by PE 1 and requires no block RAMs. This strong contrast demonstrates why we are able to fit far more processors on an FPGA device than Dou et al.; it is the main reason for our array's superior performance. After place and route, 99% of the block RAMs and 92% of the slices are used.

To compare array performance to the software baseline, we generated 100,000 random RNAs of length 273 and folded them both in hardware and in software. Our array,

Table I
AREA REPORT OF PROCESSORS IN OUR ARRAY. THE THREE ROWS REPRESENT NUMBER OF LUTS USED AS SHIFT REGISTERS FOR DELAYS, NUMBER OF LUTS FOR ARITHMETIC, AND NUMBER OF BLOCK RAMS.

	Controller	**PE0**	**PE1**	**PE2-30**	**PE31-N/2 Ver. A**	**PE31-N/2 Ver. B**
SRL16s	20	17	1842	0	0	544
LUTs	1127	273	2426	762	276	136
BRAMs	0	3	10	5	2	0

running in hardware, took 57.14 seconds, which almost exactly equals the runtime predicted by Equation 25. The baseline system performed the same computation in $5,894.44$ seconds; our array is $103.2\times$ faster than the single core.

A. Comparison to Related Work

We can compare the performance of our array to related work described in Section II. Dou et al. implemented their array on the same FPGA family used in our experiment, a Xilinx Virtex 4 LX200-11, which can fold an RNA of length N using p processors in $\frac{N^3}{6p}+\frac{125p+12}{8p}N^2+\frac{p+183}{12}N-\frac{3}{8}N^2$ clock cycles. They were able to fit 16 PEs on the device, clocking the array at 135 MHz. The estimated runtime on the 100,000 RNAs of length 273 is $1,007.4$ seconds. Our array performs the same computation $17.6\times$ faster.

Rizk and Lavenier used their GPU implementation to fold 40,000 randomly generated RNAs of length 120 bases on an NVIDIA Tesla C870 and GTX280. The execution times of the two GPUs were 32.8 and 18.9 seconds respectively. We built a new bitfile to fold 121-base RNAs. We were able to fit two arrays on our FPGA and clock the design at 110 MHz. To amortize I/O, we folded 80,000 randomly generated sequences in hardware and halved the runtime to derive the execution time for 40,000 sequences — 2.72 seconds. Our design can fold 120-base RNAs $12.1\times$ and $6.9\times$ faster than the Tesla C870 and GTX280 respectively. Note that both GPUs we have compared against are from a newer generation compared to the Virtex 4 family. Our hardware is also $119\times$ faster than their baseline, a single core of a Xeon 2.66 GHz with 6 MB cache.

VIII. CONCLUSIONS

In this work, we have demonstrated the use of polyhedral analysis to build an array for the Zuker RNA folding algorithm. Our use of the polyhedral framework allowed systematic application of transformations and exploration of the design space, which is not easily achieved with *ad hoc* methods. We plan to investigate techniques that remove the limitation of a linear schedule, which in our case results in degraded performance. If we used FIFOs instead of fixed-delay registers for each dependency, we could double the speedup of our array by "skipping" iteration points outside the triangular computation domain.

Our array may be modified to implement closely related algorithms from the Vienna RNA package, including RNAalifold to fold a set of aligned RNAs and RNALfold to compute locally stable RNA secondary structures in entire genomes. Overall, we believe the polyhedral framework is well suited to accelerating the large number of data- and compute-intensive dynamic programming algorithms in computational biology.

ACKNOWLEDGMENT

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

REFERENCES

[1] R. Nussinov *et al.*, "Algorithms for loop matchings," *SIAM J. Appl. Math.*, vol. 35, no. 1, pp. 68–82, July, 1978.

[2] M. Zuker, "Computer prediction of RNA structure," *Methods in Enzymology*, vol. 180, pp. 262–88, 1989.

[3] M. Höchsmann *et al.*, "Local similarity in RNA secondary structures," in *Proc. IEEE Bioinformatics*, 2003, pp. 159–68.

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

[5] A. Mathuriya *et al.*, "GTfold: a scalable multicore code for RNA secondary structure prediction," in *Proc. ACM Symposium on Applied Computing*, 2009, pp. 981–988.

[6] Y. Dou *et al.*, "Fine-grained parallel application specific computing for RNA secondary structure prediction on FPGA," in *Intl. Conf. on Computer Design*, October 2008, pp. 240–247.

[7] G. Rizk and D. Lavenier, "GPU accelerated RNA folding algorithm," in *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, 2009, pp. 1004–1013.

[8] A. Darte *et al.*, "Constructing and exploiting linear schedules with prescribed parallelism," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 7, no. 1, pp. 159–172, 2002.

[9] S. Derrien and P. Quinton, "Parallelizing HMMER for hardware acceleration on FPGAs," *Application-specific Systems, Architectures and Processors*, pp. 10–17, July 2007.

[10] I. L. Hofacker *et al.*, "Fast folding and comparison of RNA secondary structures," *Chemical Monthly*, vol. 125, pp. 167–188, February 1994.

[11] N. R. Markham and M. Zuker, "DINAMelt web server for nucleic acid melting prediction," *Nucleic Acids Research*, vol. 33, pp. 577–581, 2005.

[12] K. Doshi, J. Cannone, C. Cobaugh, and R. Gutell, "Evaluation of the suitability of free-energy minimization using nearest-neighbor energy parameters for RNA secondary structure prediction," *BMC Bioinformatics*, vol. 5, no. 1, p. 105, 2004.

[13] A. Jacob, J. Buhler, and R. Chamberlain, "Accelerating Nussinov RNA secondary structure prediction with systolic arrays on FPGAs," in *Application-specific Systems, Architectures and Processors*, 2008, pp. 191–196.

[14] S. V. Rajopadhye, "Synthesizing systolic arrays with control signals from recurrence equations," *Distributed Computing*, vol. 3, no. 2, pp. 88–105, 1989.

[15] R. B. Lyngsø, M. Zuker, and C. N. S. Pedersen, "Fast evaluation of internal loops in RNA secondary structure prediction," *Bioinformatics*, vol. 15, no. 6, pp. 440–445, June 1999.