Accelerating ncRNA Homology Search with FPGAs

ABSTRACT
Over the last decade, the number of known biologically important non-coding RNA (ncRNA) has increased by orders of magnitude. The function performed by a specific ncRNA is partially determined by its structure, defined by which nucleotides of the molecule form pairs. These correlations may span large and variable distances in the linear RNA molecule. Because of these characteristics, algorithms that search for ncRNAs belonging to known families are computationally expensive, often taking many CPU weeks to run. To improve the speed of this search, multiple search algorithms arranged into a series of progressively more stringent filters can be used. In this paper, we present an FPGA based implementation of some of these algorithms. This is the first FPGA based approach to attempt to accelerate multiple filters used in ncRNA search. The FPGA is reconfigured for each filter, resulting in a total speedup of over 25x when compared with a single CPU.

Categories and Subject Descriptors
B.7.1 [**Integrated Circuits**]: Types and Design Styles–*Algorithms implemented in hardware;* J.3 [**Life and Medical Sciences**]: Biology and genetics

General Terms
Performance

Keywords
FPGA, ncRNA, reconfigurable computing, Viterbi, CYK

1. INTRODUCTION
Proteins are well known as the workhorse molecules for living organisms. The traditional view of ribonucleic acid (RNA) held that its main role was to encode proteins. It was a temporary product transcribed from deoxyribonucleic acid (DNA) and then translated into protein. In contrast with this role are functional non-coding RNA molecules. These ncRNAs fill a variety of biological roles, performing protein-like functions as diverse as catalyzing reactions and regulating gene expression or metabolism [3][4][13]. Recent discoveries include specific links between ncRNA and human diseases, including cancer and Alzheimer's [9].

One key problem in ncRNA bioinformatics is homology search: finding additional instances of a known ncRNA family across multiple genomes. The current algorithms to perform this task in software can have very long runtimes, on the order of days, weeks or even longer depending on the problem size. Faster implementations would also allow for searches to be run routinely that are currently impossible due to their complexity, opening up entirely new avenues of research. For these reasons, we have developed a field programmable gate array (FPGA) based implementation to reduce the runtime of the ncRNA search problem.

2. BACKGROUND
Like DNA, ncRNA is made up of a chain of nucleotides. From an abstract viewpoint, each of these nucleotides can be represented by a single base selected from an alphabet of A, C, G or U. The string of bases that make up the ncRNA molecule are called the sequence or primary structure. Much like DNA, the bases in ncRNAs can bond, or pair, with one another. The strongest bonds form between adenine (A) and uracil (U), and between guanine (G) and cytosine (C). In addition to these Watson-Crick pairs, ncRNAs also form weaker G-U wobble pairs, as well as other interactions. After being transcribed, some of the bases in the ncRNA molecule will bond with their complements, creating various two-dimensional features. This shape, known as the secondary structure, depends on both the sequence and on which bases ultimately end up paired. The actual location of each nucleotide in three-dimensional space inside the cell is referred to as the tertiary structure.

The Rfam database is an attempt to classify all known ncRNAs into families based on their functions [11]. Much like proteins, these functions strongly depend on the three-dimensional structure of the ncRNA molecule, but, also like in the case of proteins, this information is intensely difficult to compute. For RNA, the secondary structure provides a partial proxy and allows for the use of much more efficient algorithms for determining if a given ncRNA is a member of a functional family [8]. The use of the secondary structure, instead of the primary structure, is necessary because ncRNA of the same family (in the same or different species) may have very different sequences while sharing the same secondary structure.

Rfam seeks to maximize the accuracy of its ncRNA homology searches by modeling each family [6]. This is a particularly challenging problem for Rfam because traditional sequence alignment techniques are not appropriate. Tools like BLAST, Smith-Waterman and profile HMMs are designed for matching sequences. They make no use of secondary structure information, so it would be very hard for a single model to match two ncRNAs with the same secondary structure but radically different primary sequences without a huge sacrifice of specificity. To get around this limitation, Rfam uses covariance models (CMs) to represent families [8]. Covariance refers to the fact that for the secondary structure to be preserved, paired bases must remain complementary across different members of the family even if the bases themselves change.

Figure 1 shows the sequence for an Iron response element (IRE) ncRNA in four different species. The IRE plays an important role in iron metabolism and has over 3,000 alignments in Rfam. The blue and red sections of the sequence represent two helixes, as illustrated by the secondary structure shown in Figure 2. Note that each species has a different sequence for positions 3, 4 and 5 from the left, but the bases in the 5th, 4th and 3rd positions from the right change accordingly (covary) to preserve Watson-Crick (or G-U wobble) pairs. The gaps in the sequences, represented by dots in Figure 1, mean that these pairs are not at a fixed spacing. The color coding in Figure 2 represents another important concept captured by the CM. Warmer colors represent highly conserved bases, the ones that rarely change between species. Cooler colors represent bases that are more likely to vary. Highly conserved regions of ncRNAs reflect regions that are critical to the molecule's function [16]. For example, the highly conserved loop on the end of the IRE binds with important iron response proteins to help regulate iron metabolism [2]. Notice that the base-paired regions are also highly conserved, but in the more subtle sense that pairing is preserved, rather than specific nucleotides. CMs are especially useful for ncRNA homology search because they are capable of capturing these features, specifically variably spaced conserved sequences and pairs embedded in less well-conserved regions.

```
rainbow smelt  AUUCUUGCCUCAACAGUGAUUGAACGGAAC
red junglefowl AUUAUC..GGGGACAGUGUUUCCC.AUAAU
human          UUUCCUGCUUCAGCAGUGCUUGGACGGAAC
gray wolf      UCGUUC..GUCCUCAGUGCAGGGC.AACAG
```

Figure 1. The sequence for an IRE ncRNA in four species

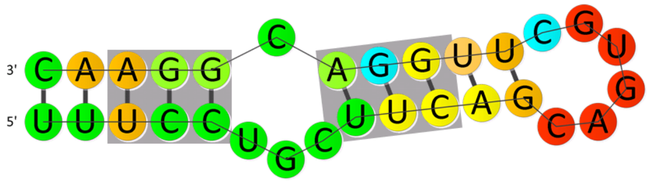


Figure 2. The secondary structure of the IRE, based on IRE_I summary [11]

Using CMs, it is possible to perform ncRNA homology searches relatively efficiently, but the computation is still very time-consuming. A typical search for a single family against a large sequence database can take days of CPU time [19]. The work presented in this paper is focused on accelerating the CM search pipeline in the Infernal 1.0 software package [15]. In addition to many other features, Infernal provides a very fast CM-based ncRNA homology search component. The speed of the search is due to three main features: 1) The use of efficient CM algorithms; 2) Highly optimized code; 3) Prefiltering the sequences using hidden Markov models (HMMs). This use of HMM filters is especially important for speedup, since the algorithmic complexity of an efficient HMM algorithm is $O(N^2)$ compared with $O(N^3)$ for a CM.

Infernal 1.0's search pipeline is made up of two filters followed by a final CM search. The first filter is HMM-based, and so it is asymptotically more efficient than a CM. This filter typically eliminates 98% or more of the dataset from consideration. The second filter is CM-based, and uses a slower but more sensitive search algorithm. This filter eliminates a large portion of the remaining input sequences, and the final most sensitive CM search algorithm, known as Inside, is run against what remains to find potential ncRNA family members.

3. PREVIOUS WORK

Infernal uses the Viterbi dynamic programming algorithm for scoring sequences against HMMs and CYK for CM scoring [15]. Both of these algorithms have a number of FPGA implementations [17][1][18][5]. However, to our knowledge there have been no previous efforts to accelerate Infernal's entire ncRNA homology search pipeline. Given the massive performance gains from prefiltering, any approach that only attacks a single stage of the Infernal pipeline is severely limited in terms of the speedup it can realize. Infernal without any filters is orders of magnitude slower, and with all pipeline stages enabled each of the three main stages is roughly equal in runtime.

Significant work has been done on using hardware to improve the runtime of the CYK portion of Infernal. The approach used by [14] is one more geared to an application-specific integrated circuit (ASIC) than an FPGA, due to the shared memory and very large switches it requires to handle CMs with many states. On a large Virtex-5 FPGA from Xilinx, this work is limited by the available logic and maxes out at around 20 processing elements (PEs). By way of contrast, the approach is designed to scale well to over 300 PEs on an ASIC. Although this is reasonable given that ASICs are the stated focus of the design, the FPGA performance is not optimized.

The Infernal CYK accelerator presented in [21] is much more FPGA centric. This design makes use of an array of processors that require minimal access to data that is not stored in the PE or available from one of its neighbors. This approach requires much less memory bandwidth and yields 32 PEs on a Virtex-5. Ultimately, it ends up being block RAM (BRAM) limited and it is most similar to the CYK acceleration approach presented in this paper. The significant differences are discussed below. Note that [21] does not address HMM filtering, so its speedup is limited to only the CYK stage of the pipeline.

Of the FPGA Viterbi implementations, the most relevant are those that implement HMMER HMMs. HMMER is a software package that makes use of Plan 7 HMMs to perform protein homology search [7]. This protein search, using families found in Pfam, is very similar to ncRNA homology search. Significantly however, given their size and structural restrictions, the primary structure alone provides sufficient sensitivity for finding protein family members. Of the efforts to accelerate HMMER on FPGAs, [18] is the most relevant. It presents a very fast FPGA implementation of Plan 7, with the ability to fit over 50 PEs on a Virtex-4. Plan 7 HMMs form the basis for the Plan 9 HMMs used in Infernal 1.0 [16], but Plan 9 is significantly different from Plan 7, and so is the FPGA implementation. The next section covers these differences in detail.

4. VITERBI

This section provides an overview of the Viterbi algorithm, as well as details and results for the Viterbi FPGA implementation.

4.1 Viterbi Algorithm and Plan 9 HMMs

The Viterbi algorithm is a two-dimensional dynamic programming algorithm [10]. Given an observed sequence and an HMM featuring a set of states with emission and transition probabilities for those states, the algorithm produces the highest scoring path through the states. This is the most likely path to have emitted the particular input sequence. Although the Viterbi algorithm itself is general and can be used with any HMM, the implementations used in HMMER and Infernal are limited to supporting only Plan 7 and Plan 9 HMMs respectively. Although they are very similar, there are some significant differences that affect the CM Plan 9 (CP9) FPGA design. The states of CP9 HMMs are shown in Figure 3.

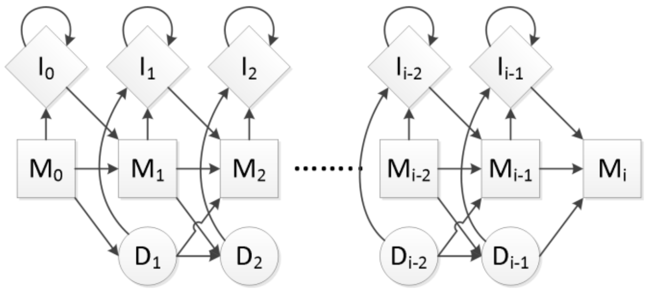


Figure 3. CM Plan 9 HMM States

In CP9, each node consists of three states: Insert, Match and Delete. The Match state emits a base that matches the one expected by the model. The Insert state emits an unexpected base without advancing to the next node. Finally, the Delete state advances to the next node without emitting. Note that every node is the same except for the first and last. For the first node, there is no Delete state and M_0 is equivalent to a Begin state, which represents the start of the model. For the final node there is no Insert or Delete and M_i represents an End state. The CP9 HMMs are built by Infernal from each family's CM. This process is based on the techniques described in [20] and creates a filter with a much faster runtime than the CM due to the better asymptotic efficiency of the Viterbi algorithm.

The first key difference between Plan 7 and CP9, and therefore between the FPGA Viterbi accelerator presented here and the previous work, is which state transitions are available. In Plan 7, there are no transitions from Insert to Delete or vice versa. Supporting these edges requires slightly more complex PEs in CP9. There are two more significant differences, but these require a discussion of the CP9 Viterbi algorithm first.

The Viterbi algorithm makes use of a two-dimensional dynamic programming table to store the probability scores computed for each state. The work for each node involves computing the probability of each transition into that node given the current sequence element and the scores of previous nodes. For example, to compute the Insert score for some node requires taking the sum of the Insert emission probability for that state, given the current base in the sequence, and the maximum of the scores of the three states that could transition into an Insert. Each of these scores must have the correct transition probability added to it before the max is computed. The final equation for an Insert score is as follows:

$$\text{Insert}(j,i) = \text{emit\_prob}(\text{Insert}_j, \text{Seq}_i) + \max\begin{cases} \text{Insert}(j,i-1) + \text{trans\_prob}(\text{Insert}\rightarrow\text{Insert}_j) \\ \text{Match}(j,i-1) + \text{trans\_prob}(\text{Match}\rightarrow\text{Insert}_j) \\ \text{Delete}(j,i-1) + \text{trans\_prob}(\text{Delete}\rightarrow\text{Insert}_j) \end{cases}$$

During the Viterbi run, the table cell *j*,*i* contains a score representing the probability of the sequence up to some element *i* matching up with the model up to some state *j*, and by the end of the run the entire table will be filled in this manner. The equations for the remaining transitions are very similar to those used in Plan 7 and can be found in the previous work [17]. Notice that the Insert equation above has the additional Delete to Insert transition. The others must be similarly modified.

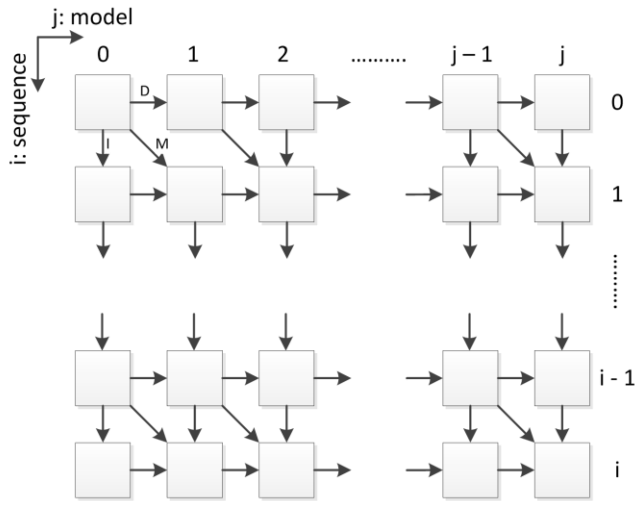


Figure 4. CP9 DP Table Data Flow

One dimension of the dynamic programing table is the model, so moving one cell to the right in Figure 4 is equivalent to moving to the next state in the model. The other dimension represents the sequence being scored, so moving down the table represents emitting one base. Each cell in Figure 4 represent an entire node, meaning that they each contain a Match, Insert and Delete state. The Delete state does not emit a base, instead it represents a state in the model that is not found in the sequence. For this reason, Delete is a transition from one node to its right hand neighbor in the table. Similarly, Insert emits a base that does not match any model state, so it moves down the table. The Match state is used when a base fits the model's expectations. In this case it both emits the base and moves on to the next model state, moving right and down one node in the table.

These data dependencies are such that the CP9 HMM has no backwards paths. This means that a computation can be performed along a wavefront starting with node (0, 0) (which depends on no other nodes) followed by computing (1, 0) and (0, 1) in parallel, then (2, 0), (1, 1) and (0, 2) also in parallel and so on. In other words, the lack of backwards data flow means that there is a huge amount of parallelism available in the computation. Note that communication is not all local from one node to the next. Not shown here are the Begin and End state transitions. These provide some (typically very low) probabilities to jump from the Begin state to any position in the model, and similarly from any state to End. Although these transitions are not local like the transitions in Figure 4, they are still entirely feedforward, so no parallelism is lost.

The fact that there are no feedback paths in CP9 HMMs is the second critical difference between CP9 and Plan 7 HMMs. The feedback path in Plan 7 allows for the model to match multiple copies of itself in succession [18]. This feedback requires special consideration for an FPGA implementation, and so the lack of it in CP9 gives us greater design freedom.

Among the changes from Plan 7, the third and most significant is the introduction of another state type. This End Local (EL) state only exists for some nodes along the model, and allows for large portions of the model to be bypassed. Because EL states also have a self-loop, like Inserts, they allow for a number of bases to be emitted while skipping over a potentially large portion of the model. More specifically, when the CP9 HMM is built from the CM, some nodes will contain an EL state and a probability to transition from Match to this state. The only other way to reach the EL state is through a self-loop which emits a base and stays in the same state. The only transition from the EL state is to some subsequent Match state. This transition is unique in CP9 in that it is the only transition other than from Match to End that can jump forward many states. It is also different from other states in that not every Match has an associated EL state. In addition, a single Match can have multiple incoming EL transitions. See Figure 5 for an example of some of these properties. Because of these properties a more flexible design is required to handle EL states.

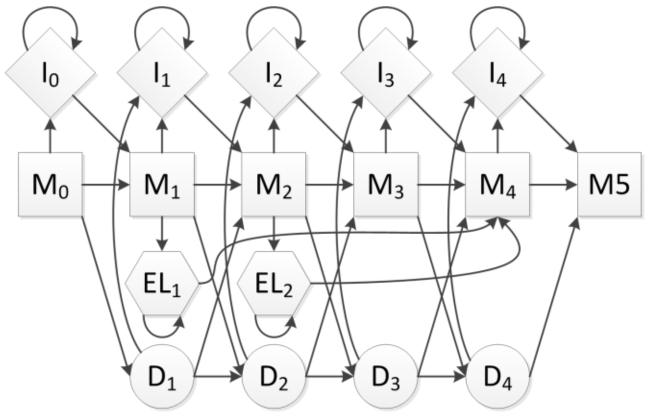


Figure 5. Example of a CP9 HMM Featuring EL States

4.2 FPGA Implementation

Figure 6 shows a block diagram of the CP9 Viterbi implementation developed in this work. For all of the results presented in this and subsequent sections, our FPGA designs targeted a Pico Computing system featuring an EX-500 backplane equipment with an M-503 module. This module features a Xilinx Virtex-6 LX240T as well as 8GB of DRAM and 27MB of SRAM. The design operates on a streaming paradigm. This is appropriate because a single, relatively small HMM may be run against a very long sequence or set of sequences. The sequence is streamed in from offchip and scores are streamed out on a similar channel. No other external memory or offchip communication is used.

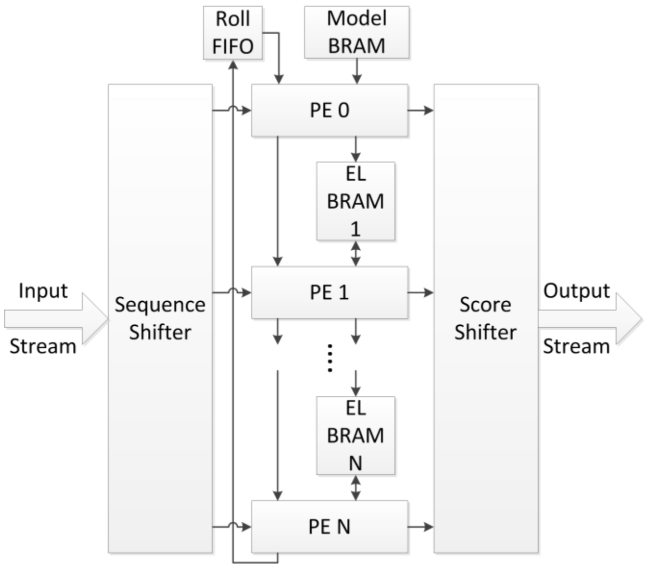


Figure 6. CP9 Viterbi FPGA Block Diagram

The CP9 Viterbi FPGA design is based on a linear array of PEs. Once the processor pipeline has filled, this allows all PEs to operate in parallel along the wavefront described in the previous section. Given the dynamic program table in Figure 4, it makes the most sense for a single PE to either move down a column, computing scores for many bases from the sequence for a single state, or across a row, handling all states for a single base. Either of these arrangements allows for neighbors to communicate model and score values. Because CP9 does not have a feedback state, but it does have EL states, it is most feasible for a single PE to handle all states for a given base, for the next PE to handle the next base, and so on down the sequence as shown in Figure 7. Every PE will be computing at all times except for the beginning and end of the sequence. There are some cases that break this parallelism, for example, a very short model with fewer states than there are PEs. However, this situation can be handled by replicating the model multiple times and shifting in the next base after reaching the end of the first copy of the model.

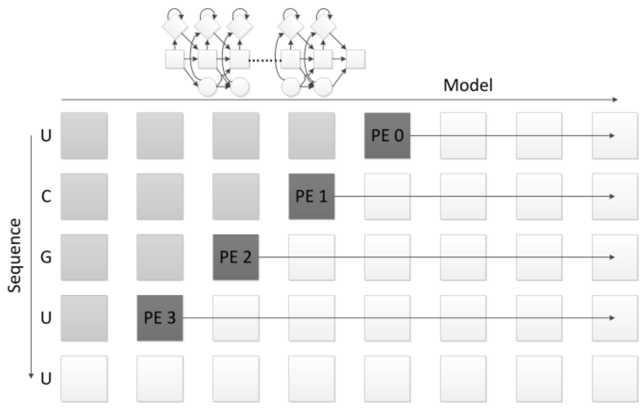


Figure 7. CP9 Viterbi PE Allocation and Wavefront

The Sequence Shifter itself is simply a set of shift registers. The first shifts bases in from the input stream and loads them into a second shift register when required. After a PE has completed work on a base, a new base is loaded from the second shift register for that PE. The Score Shifter works on similar principles and also filters output scores to avoid saturating output bandwidth.

The Model BRAM stores the CP9 HMM paramaters, which include emission scores for Insert and Match states and transition scores for all possible state transmissions. These scores are stored in log-odds form, which converts multiplications to addition of precomputed logs. Viterbi also requires maximum operations, which can use the same hardware in log and regular integer format. In addition, the precision of the calculations can be adjusted by changing the number of bits used to store the scores throughout the system. The amount of BRAM required to store the models depends on this width. Every state requires an emission probability for each possible base for Match and Insert, or 8 emission probabilities per state. There are transmission probabilities between every possible combination of the three states in the nodes for a total of 9 different values. In addition, there are transition probabilities for Begin, End and EL for each state, bringing the total up to 20. Letting w_s represent the number of bits per score, this requires $20 \times w_s$ bits per state. Setting the maximum model length to 2k, which can handle all of the current Rfam models, and given that the Virtex-6 series features 36 Kb BRAMs, this means that the total model BRAM requirement is $2k \times 20 \times w_s / 36k = 1.25 \times w_s$. For our current system, a w_s value of around 18 bits is sufficient, requiring about 24 of our FPGA's 416 BRAMs for model tables, due to padding.

The Processing Elements themselves are fairly simple. The first PE receives model information from the BRAM. Since the computation wavefront, shown in Figure 7, is such that each PE is always one node in the HMM behind the previous PE, this model information can be passed from one PE to the next as well. Dark grey cells represent current computation while light grey cells have already been scored. The alternative approach, where each PE handles many sequence bases for the same state, requires that each PE have access to its own portions of the table, potentially reducing efficiency of BRAM use or introducing additional multiplexers.

In addition to the transition and emission probability and sequence data discussed above, the PEs need some way to access and update the DP score table. Because all communication is either between adjacent bases or across many states but within the same base, there is no need to actually store the entire table in memory. Instead, the values that are required at any given time are only those immediately surrounding the computation wavefront. Furthermore, values that do require longer communications, such as Begin and End scores (with an End score for each base being the ultimate output of the FPGA accelerator) can be stored in the local PE. The only exception to this local communication is the EL state, which we will discuss in detail. Since the input sequence proceeds down the PEs in order, the next base after PE N has to be handled by PE 0. At this point, all score information must be transferred back to PE 0. For models much longer than the number of PEs, PE 0 will still be busy computing its current base, so this score information must be stored in the Roll FIFO shown in Figure 6. The total amount of score information required for each base is one score each from the Match, Insert, Delete and EL states, or $4 \times w_s$. There is also an EL BRAM (not shown) for PE 0 located after the FIFO.

The PEs themselves are simple pipelined arithmetic units. They perform the addition and max operations required to calculate the scores for the DP table cells they are processing before moving on to the next cell in the next cycle. It is worth noting that although the current version with a two stage pipeline can run at slightly over 100 MHz, it is challenging to add more pipeline stages. This is due to the various local dependencies for individual HMM states like EL and End. In addition to computing table values for the current cell, the PE stores the running best scores for the current base for long transitions. These include scores for EL, Begin and End.

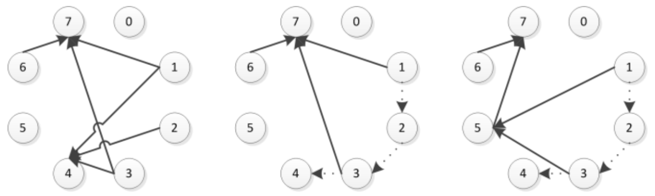
EL states are a special case for a number of reasons introduced earlier. To summarize, EL states introduce a data dependence between cells more than one cell apart. EL states also have a self-loop with a score penalty that transitions to the next base like the self-loop for an Insert state. In order to handle these irregularities, we introduce the concept of the EL memory. This memory is necessary because unlike the End state, storage for many EL scores could be required simultaneously. See Figure 5 for an example of how this could happen. In this case, a PE computing M_4 requires data from both EL_1 and EL_2. More complex models require storage for many more ELs. Furthermore, there are some situations with much greater EL fan-in than in this example. Although these situations are handled with a simple loop over the EL input states when computing a Match score in the Infernal software, that luxury is not available for an FPGA implementation. Instead, our design handles this problem by combining ELs in a tree-like structure while running along the model. The shape of this combination tree is computed in software and stored with the model information prior to a Viterbi run. This is possible because EL states are typically generated from states that pair in the CM, and physical limitations on the molecule shape prevent overly complex EL patterns. In fact, for many CP9 HMMs a stack would be sufficient for EL storage. The algorithm for EL combination is presented in Figure 8.

```
combineELs(ELMap)
Input: ELMap, containing list of all EL transitions
Output: ELCmd, containing EL memory and register
        commands for each CP9 HMM state

Convert ELMap to ELCmd equivalent (EL memory ops)
foreach link l in ELMap
    if l has only one destination state and
    if next l has only the same destination state
        remove l's EL memory ops from ELCmd
        add equivalent EL register op to ELCmd
foreach command c in ELCmd
    while c has reads from > 2 EL memory locations
        let wl be the origin state of data for reads in c
        sort wl by state
        foreach HMM state s starting at second state in wl
            if s has no EL memory ops (register ops are OK)
                add two reads (based on order in wl) to s
                remove those reads from ELCmd for c
                add write of combined data to ELCmd for s
                add a read of combined data to ELCmd for c
                terminate foreach
            if s = state of c
                insert dummy state after second state in wl
                terminate foreach
```

Figure 8. EL Combination Algorithm for CP9 HMMs

The EL combination algorithm above is capable of giving all states a maximum of one EL write and two EL reads, in addition to the use of an EL register inside the PE. In the FPGA, this translates to a pair of mirrored BRAMs for a total of one write port and two read ports. For an example of the algorithm's operation, see Figure 9. A represents the initial state after software EL states are converted into EL memory commands. States 4 and 7 both require three reads, so this is not a legal set of commands for our hardware. In B, after the EL register conversion has taken place, the extra reads on state 4 have been resolved. The register operations automatically combine EL scores and are represented with a dotted line. Finally, in part C, the first two EL reads in state 7 are reduced to a single read by doing an EL combination in state 5. Now no state has more than two reads and one write, resulting in a legal set of EL commands.



A. Initial Model

State	Cmd
1	W0
2	W1
3	W2
4	R 0, 1, 2
6	W3
7	R 0, 2, 3

B. After Register

State	Cmd
1	W0, Reg
2	Reg
3	W1, Reg
4	Read Reg
6	W2
7	R 0, 1, 2

C. After Combining

State	Cmd
1	W0, Reg
2	Reg
3	W1, Reg
4	Read Reg
5	R 0,1 W0
6	W1
7	R 0, 1

Figure 9. EL Combination Algorithm Example

Currently, two BRAMs per PE is not a limiting factor, as can be seen in the next section, but if it becomes an issue in the future it would be possible to double-pump the BRAM ports and use a single BRAM for each PE. Finally, although it is possible to conceive of a scenario where the dummy states inserted by the combination algorithm could have a substantial impact on runtime, or even make an HMM too long to fit in the Model BRAM, we have not yet encountered any Rfam families that require even a single dummy state.

4.3 Viterbi Results

Running at 100 MHz, the CP9 Viterbi implementation is limited by the available logic resources on our Virtex-6, as seen in Table 1. This is the expected result, given that each PE has many adders and max units. The current PEs make use of parallel maxes, a poison bit for overflow and other frequency optimizations that result in larger PEs. Determining the optimal tradeoff between frequency and quantity of PEs on an FPGA remains in the domain of future work.

Table 1. Post-map Logic Utilization of Viterbi Design

PEs	Slices	LUTs	BRAMs
4	14%	9%	13%
16	27%	15%	19%
64	67%	43%	42%
128	97%	77%	73%

The performance of the FPGA-based solution is very good when compared with Infernal. Figure 10 shows the speedup vs. Infernal 1.0 software running on a single Intel Xeon E5520 core. Adding PEs to the system results in nearly linear speedup, and this is to be expected given that there is no significant overhead. Speedup over Infernal 1.0 decreases as model size (shown in parentheses) increases, leveling out at around 230x. This is most likely due to inefficiencies in the software Viterbi implementation for very small models. Note that the median HMM model length in Rfam is under 100 states. This chart compares only the runtime of the Viterbi algorithm itself, not the entire FPGA and software systems.

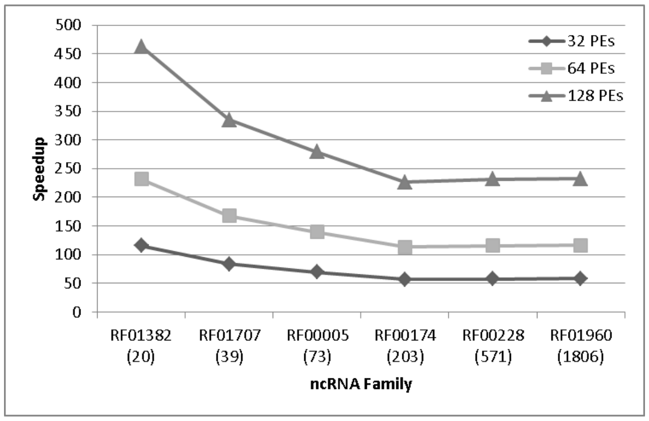


Figure 10. Speedup of FPGA Viterbi vs. Single CPU Infernal

5. CYK

This section provides an overview of the CYK algorithm, as well as details and results for the CYK FPGA implementation.

5.1 CYK Algorithm

To achieve high performance, Infernal's HMM filter stage removes the non-local dependencies in the ncRNA structure and simply looks at features detectable in a linear search. While this ignores important elements of the structure, the filter runs quickly and can eliminate most of the input data. However, the HMM will accept many regions that do not actually include the target ncRNA. Further filtering requires looking at the secondary structure of the ncRNA model for which we are searching, and is performed by the Cocke-Younger-Kasami (CYK) algorithm [22].

CYK is a dynamic programing algorithm, in some ways analogous to a three-dimensional version of Viterbi. Also like Viterbi, CYK is a general algorithm that can be used to parse any probabilistic context-free grammar in Chomsky normal form. CMs fit this description, as they are constructed using only a few rules of the appropriate form [8]. The CM directly represents the pairings and branchings found in the secondary structure of an ncRNA. This is done through Match Pair and Bifurcate nodes respectively. As can be seen in Figure 11, Match Pair nodes represent pairings of potentially distant bases in the ncRNA sequence, something the HMM model cannot support. In Hammerhead_3, the second and second-to-last bases are emitted by the same Match Pair node. Bifurcation nodes represent branches in the ncRNA secondary structure. Handling Bifurcations will require breaking the problem up into two smaller subproblems (one for the left branch and one for the right branch). Unfortunately, since each subproblem can use varying lengths of the target RNA, given that biological evolution can change the length of individual branches, we cannot know ahead of time where the split point occurs in the candidate sequence. This is solved by simply trying every possible split-point and picking the highest scoring match, a fairly time-consuming process. In Figure 11, the Bifurcate node takes place before the sets of Match Pairs that make up Hammerhead_3's two arms. This requires a Bifurcation state, because without one there would be no way to express two separate sets of paired bases.

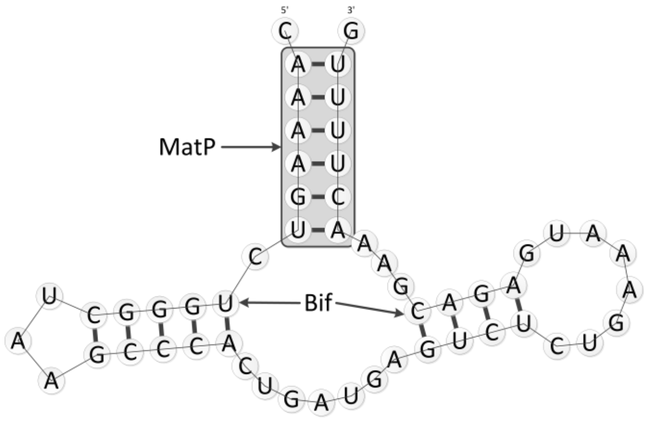


Figure 11. Secondary Structure of a Hammerhead_3 ncRNA

The CYK algorithm starts with a sequence of length N and a CM, and scores that sequence against the model with a better match resulting in a higher score. It converts this into a three-dimensional dynamic programming problem by creating layers of triangular DP matrixes, one for each state in the CM as in Figure 12. These matrixes are triangular because position *j*, *i* in state S represents the best matching of states $S \ldots S_{End}$ to region *i*…*j* of the target sequence. It does not make sense to match to a region whose start is after its end, so entries *j*, *i* where $j > i$ are useless. Emissions of single bases are achieved through Match Left and Right states, where Left states move up one cell in the DP matrix by subtracting one from *i* as shown in Figure 12. Similarly, Right states move right one cell by adding one to *j*. The final dimension, *k*, refers to the state.

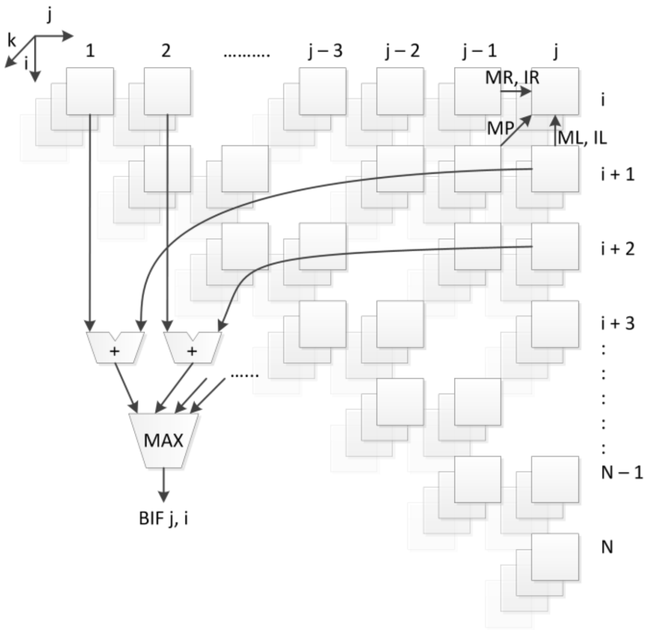


Figure 12. CM CYK Three-dimensional DP Table

As the CYK algorithm proceeds, it works from the last CM state S_{End} back to the starting state S_{Root}, and processes from small sequences of length 1 (i.e. squares *i*, *i*+1) towards longer sequences. The probability score for each subsequence is stored in the DP table to be reused without being recomputed. This process handles most parts of the model (including Match Pair) very efficiently, with only local communication. The one exception is the Bifurcation state. In this state, processing a sequence from *j* to *i* requires finding the best split point, mid, where $j < mid < i$. Since we don't know where the bifurcation point is in the target sequence, CYK must compute the highest score for all possible mids by maximizing the score(*j*, *mid*, $branch_1$) + score(*mid*, *i*, $branch_2$) where $branch_1$ and $branch_2$ are the states containing the total scores for the subtrees. The first two cells of this calculation are shown in Figure 12 and the equation can be found in [8]. This computation is very similar to matrix multiplication and requires $O(n^3)$ arithmetic operations for each Bifurcation resulting in a total CYK runtime on the order of $O(kn^2 + bn^3)$. Because of the expense of this operation, it can dominate the computation time for very large models as seen in Figure 13. In practice, the impact of Bifurcation states is not always as extreme as this chart makes it appear because over half of all Rfam CMs are under 100 nodes long.

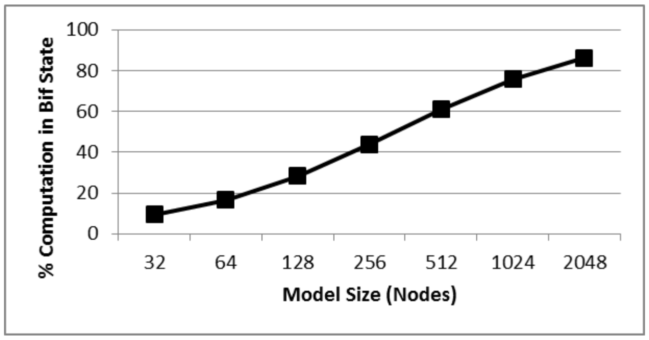


Figure 13. Computation in Bif State Assuming 0.9% Bifs

5.2 FPGA Implementation

Our implementation of CYK for CMs, diagramed in Figure 14, is based around a linear array of PEs, similar to the one used for Viterbi. The most significant difference in the design is the use of off-chip memories. Because it has a three-dimensional DP table, CYK requires substantial more storage. By dividing the entire matrix up into stripes and letting each PE handle a single column of the DP table as in Figure 16, most of this memory can be made local to the PE in the form of BRAM [21]. This method of striping the model is possible because, much like Viterbi, there are no non-local or backwards data dependencies for non-Bifurcation states. The calculations for a given cell in a non-Bifurcation state only require data from the same cell and its immediate neighbors from layers in the previous node. This means the computation wavefront is similar to Viterbi, and a similar arraignment of PEs inside the stripes is most effective. After completing one state for a given stripe, the PEs can begin work on the next layer. After completing all layers, the PEs begin work on the next stripe until computation is complete. Each PE contains many simple arithmetic units to update all states in the node simultaneously. This requires striping the state data across many local BRAMs.

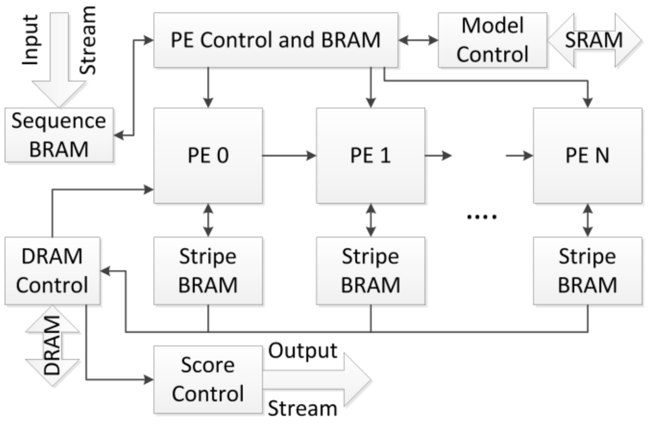


Figure 14. CM CYK FPGA Implementation Block Diagram

All of the DP cells for the current stripe can be stored in the local BRAM associated with each PE. Values from cells to the left can be passed from the previous PE. The values computed at the end of a stripe are written out to DRAM, to be reloaded when the same layer is reached during the next stripe. The only other time that DRAM storage is required is for one of the Begin layers prior to a Bifurcation state. Because the Bifurcation computation uses non-local data, it requires substantially more cells from DRAM. Figure 16 shows that PE operation is substantially different for Bifurcations and this requirement explains why. The row data used in the Bifurcation computation discussed in the previous section is streamed in from DRAM and passed from one PE to the next. Column data is loaded from BRAM as with other states. Every PE is equipped with multiple Bifurcation arithmetic units allowing for the use of multiple DRAM rows simultaneously, improving use of memory bandwidth for Bifurcation states. This can be seen in Figure 15.

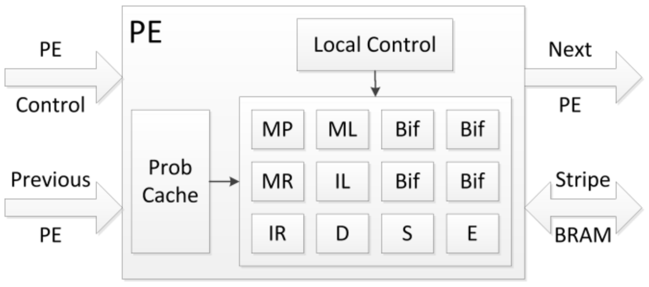


Figure 15.CM CYK PE Block Diagram

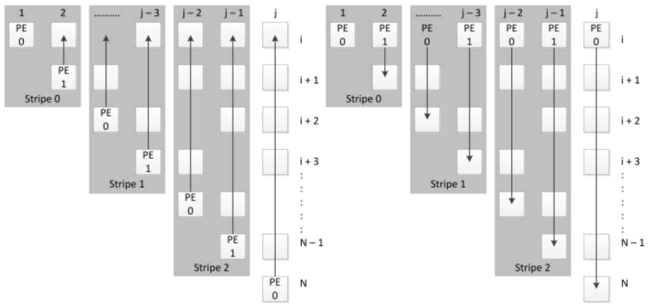


Figure 16. Striping for Normal (left) and Bif (right) states

In addition, the CM model itself can be much larger than the Viterbi model, and BRAM is the limiting resource for this design, so the model is stored off-chip. BRAM becomes the limiting factor because it would be impossible to achieve high performance if the data required by all stripes was loaded from DRAM. Our M-503 platform offers two 64-bit DDR interfaces running at 400MHz for a total of 12.8 GB/s of DRAM bandwidth. Ignoring the bandwidth required to load the data along the first row, each PE requires as many as four DP table values each cycle for Match Pair, Match Left, Match Right and Delete states. This means that again assuming a score width of w_s the bandwidth required for a single PE is 4 × FPGA speed × w_s. Using w_s of 21 bits [14], slightly more than Viterbi, and a speed of 100 MHz, this gives a requirement of about 1 GB/s of bandwidth for each PE. This result would limit our design to 12 PEs on the M-503.

When using the striped approach with BRAM there is sufficient DRAM bandwidth available and BRAM capacity becomes the limiting factor. Assuming a maximum sequence length of 6144 bases, reasonable given the length of CMs in Rfam, the BRAM requirement for the largest stripe becomes 4 × 4096 × w_s. Again using 21 bits, this yields a storage requirement of 504 Kb. Given the Virtex-6's 36 Kb BRAMs, this would require 14 BRAMs for each PE, allowing 24 PEs per FPGA after accounting for other BRAM usage. This solution is clearly superior, with over twice as many PEs on-chip compared to a design that stores the entire table in DRAM. Another advantage is that as the PE clock speed increases the DRAM bandwidth would be able to support even fewer PEs.

Another way the CYK design differs from Viterbi is in the input sequence handling. For Viterbi, the sequence can be streamed in, sent to the PEs and then discarded. Unfortunately, this is not the case for CYK. Because the use of stripes results in each stripe requiring the same parts of the sequence, the entire sequence must be stored for reuse after being streamed in. The sequence BRAM, shown on the block diagram, serves this purpose.

The last significant CYK module is the Controller. This module contains the Finite State Machine (FSM) responsible for the CYK algorithm operation. The Controller also contains BRAM that stores model-specific instructions. These consist of a number of fields generated by software and written to the FPGA prior to accepting sequence input. Entries in this table include DRAM and SRAM addresses and lengths for each stripe as well as state and other model information for the stripe. The CM itself can be large and is stored in off-chip SRAM. The FSM currently follows a simple schedule of loading the stripe information from the controller BRAM, loading appropriate DRAM values and then executing to completion while writing values back to DRAM. Future versions of the design could feature a more complex controller that performs some of these operations in parallel.

5.3 CYK Results

Running at 100 MHz, we estimate that the FPGA CYK implementation is limited by BRAMs available on chip, as seen in Table 2. This is the expected result given the discussion of BRAM utilization in the previous section. Future designs may contain additional parallel logic in an attempt to achieve better performance with the same memory utilization. Note that all results in this section are estimated based on functional Verilog simulation and synthesis.

Table 2. Estimated Logic Utilization of CYK Design

PEs	Slices	LUTs	BRAMs
6	35%	25%	35%
12	42%	32%	56%
24	51%	39%	96%

CYK performance shows much more unpredictable variation between different CMs than Viterbi, as seen in Figure 17. This is because the CYK algorithm's runtime depends on a number of factors other than model size, including the prevalence of Bifurcation states. In addition, the speedup from adding PEs is sublinear for small models due to the overhead of switching between stripes and computing scores for cells in the stripe that are not actually part of the DP triangle. This effect is substantially more pronounced in smaller models because the average height of a stripe is much less. Overall speedup is still very good, with a geometric mean speedup of over 70x for 24 PEs.

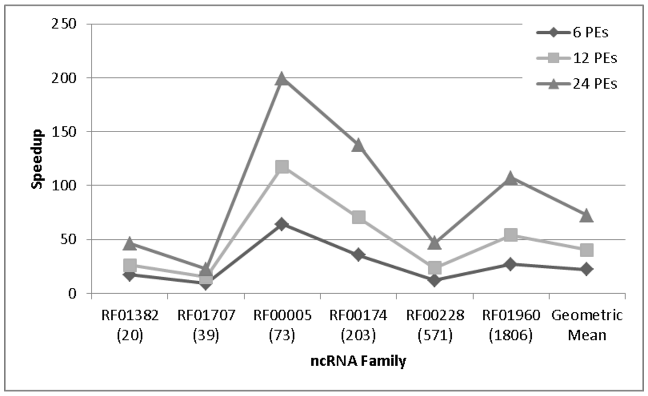


Figure 17. Estimated FPGA CYK Speedup vs. Infernal

To get an estimate of the performance of Infernal's final search algorithm, Inside, it is necessary to determine the number of Inside PEs that would fit on our FPGA. The large increase in size from a CYK PE to an Inside PE is due to the use of floating point (FP) addition. The CYK and Viterbi implementations avoid FP arithmetic by using log-odds scores. However, Inside requires addition of scores, instead of only multiplication, which is expensive in the logarithmic representation. For FPGAs, single precision FP addition requires approximately 20x more area than log multiplication (integer addition), which is still only half as expensive as log addition [12]. Based on these numbers, and our calculations showing that the CYK design becomes FPGA area limited at around 80 PEs, a very conservative estimate for Inside would allow for 4 PEs and one sixth the speedup of CYK. This estimate is very likely excessively pessimistic because the Infernal 1.0 implementation of Inside is also slower than CYK.

6. FPGA BASED SYSTEM

Although the performance of the individual FPGA implementations of Viterbi and CYK is important, the ultimate goal of this work is to accelerate Infernal 1.0 in a way that is useful to biologists and others in the field. To measure progress towards this goal, it is valuable to look at the performance of Infernal 1.0 as a whole. Infernal performance using the FPGA runtimes for Viterbi and CYK are shown in Figure 18. One advantage of a system that uses FPGA implementation to accelerate Viterbi and CYK, such as the one modeled here, is the possibility to reconfigure a single FPGA to run both filters. A multi-FPGA system could divide FPGAs between Viterbi and CYK as required given that some ncRNA families spend more time in one algorithm or the other, as seen in Figure 18. This is due to a number of factors, including the filtering fraction achieved by each filter and the asymptotically worse runtime of CYK, which begins to dominate for larger models. The ability to reconfigure the same hardware and allocate resources as required on a per-model basis gives the FPGA implementation an advantage over potential ASIC designs like [14].

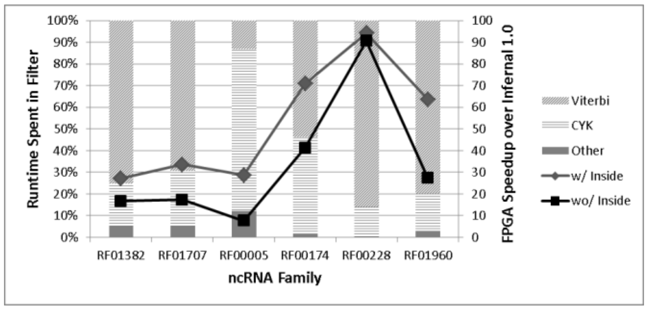


Figure 18. Estimated Infernal 1.0 FPGA System Speedup

The speedup here is less than that of the individual accelerators - particularly without Inside acceleration - for two major reasons. First, without an accelerated version of the Inside algorithm sequences that make it past both the Viterbi and CM filters must still run in software. Although a very small percentage of the sequence passes both filters, the Inside algorithm is much slower than CYK, so a substantial amount of time is still spent running Inside. This time is less than 10% of the total prior to acceleration, but dominates afterwards. The other reason for the reduction in speedup is that these results include all software tasks performed by the system, not just Inside and the two filters. Although for large enough sequences the time spent in these tasks becomes very small, averaging less than 1% of the total runtime for a 60 million base sequence, they cannot be ignored. It is also worth noting that the ncRNA families selected for this experiment tend to be among the more frequently occurring, so speedup would likely be better for a more typical family because Inside would have fewer possible matches to score. With estimated Inside acceleration, the expected overall system speedup is much better in the worst case. For tRNA (RF00005), a particularly unusual ncRNA, speedup improves from 8x to 29x with Infernal acceleration.

7. CONCLUSION

This work presented an FPGA accelerator for the biologically important ncRNA homology search problem. The accelerator features FPGA implementations of the two filtering algorithms used by the Infernal 1.0 software package. The FPGA version of the first of these algorithms, Viterbi, gets a speedup of over 200x. The second, CYK, achieves a speedup of 70x over the software version. When combined into an Infernal-like system, we anticipate that these accelerators alone can run 25x faster than pure software. This speedup is limited by the fact that our current system has no accelerator for the Inside algorithm used as the final stage in Infernal's search pipeline. Given the results of our other accelerators, we anticipate that an Infernal 1.0 system featuring an FPGA implementation of the Inside algorithm could achieve a total system speedup of over 47x. To our knowledge, this is the first system to accelerate multiple stages of an ncRNA homology search pipeline, which we achieved using FPGA reconfiguration.

8. ACKNOWLEDGMENTS

This work was supported in part by Pico Computing.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0718124.

9. REFERENCES

[1] Abbas, N. et al. 2010. Accelerating HMMER on FPGA using Parallel Prefixes and Reductions. (2010).

[2] Addess, K.J. et al. 1997. Structure and dynamics of the iron responsive element RNA: implications for binding of the RNA by iron regulatory binding proteins1. *Journal of molecular biology*. 274, 1 (1997), 72–83.

[3] Bachellerie, J.P. et al. 2002. The expanding snoRNA world. *Biochimie*. 84, 8 (2002), 775–790.

[4] Bartel, D.P. 2004. MicroRNAs: genomics, biogenesis, mechanism, and function. *Cell*. 116, 2 (2004), 281–297.

[5] Ciressan, C. et al. 2000. An FPGA-based coprocessor for the parsing of context-free grammars. *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*. (2000), 236–245.

[6] Eddy, S.R. 2002. Computational genomics of noncoding RNA genes. *Cell*. 109, 2 (2002), 137–140.

[7] Eddy, S.R. 1998. Profile hidden Markov models. *Bioinformatics*. 14, 9 (1998), 755.

[8] Eddy, S.R. and Durbin, R. 1994. RNA sequence analysis using covariance models. *Nucleic acids research*. 22, 11 (1994), 2079–2088.

[9] Esteller, M. 2011. Non-coding RNAs in human disease. *Nature Reviews Genetics*. 12, 12 (2011), 861–874.

[10] Forney Jr, G.D. 1973. The Viterbi algorithm. *Proceedings of the IEEE*. 61, 3 (1973), 268–278.

[11] Griffiths-Jones, S. et al. 2005. Rfam: annotating non-coding RNAs in complete genomes. *Nucleic acids research*. 33, suppl 1 (2005), D121–D124.

[12] Haselman, M. et al. 2005. A comparison of floating point and logarithmic number systems for FPGAs. *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*. (2005), 181–190.

[13] Mattick, J.S. 2009. The genetic signatures of noncoding RNAs. *PLoS genetics*. 5, 4 (2009), e1000459.

[14] Moscola, J. et al. 2010. Hardware-accelerated RNA secondary-structure alignment. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*. 3, 3 (2010), 1–44.

[15] Nawrocki, E.P. et al. 2009. Infernal 1.0: inference of RNA alignments. *Bioinformatics*. 25, 10 (2009), 1335–1337.

[16] Nawrocki, E.P. and Adviser-Eddy, S.R. 2009. Structural RNA homology search and alignment using covariance models. (2009).

[17] Oliver, T. et al. 2008. Integrating FPGA acceleration into HMMer. *Parallel Computing*. 34, 11 (2008), 681–691.

[18] Takagi, T. and Maruyama, T. 2009. Accelerating HMMER search using FPGA. *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. (2009), 332–337.

[19] Weinberg, Z. and Ruzzo, W.L. 2004. Faster genome annotation of non-coding RNA families without loss of accuracy. *Proceedings of the eighth annual international conference on Resaerch in computational molecular biology*. (2004), 243–251.

[20] Weinberg, Z. and Ruzzo, W.L. 2006. Sequence-based heuristics for faster annotation of non-coding RNA families. *Bioinformatics*. 22, 1 (2006), 35–39.

[21] Xia, F. et al. 2010. Fine-grained parallel RNA secondary structure prediction using SCFGs on FPGA. *Parallel Computing*. 36, 9 (2010), 516–530.

[22] Younger, D.H. 1967. Recognition and parsing of context-free languages in time n^3. *Information and control*. 10, 2 (1967), 189–208.