# Accelerator Design for Protein Sequence HMM Search

**Rahul P. Maddimsetty**
**Jeremy Buhler**
**Roger D. Chamberlain**
**Mark A. Franklin**
**Brandon Harris**

Rahul P. Maddimsetty, Jeremy Buhler Roger D. Chamberlain, Mark A. Franklin, and Brandon Harris, "Accelerator Design for Protein Sequence HMM Search," in *Proc. of 20th ACM Int'l Conference on Supercomputing*, June 2006.

Dept. of Computer Science and Engineering
Washington University
Campus Box 1045
One Brookings Dr.
St. Louis, MO 63130-4899

# Accelerator Design for Protein Sequence HMM Search

Rahul P. Maddimsetty, Jeremy Buhler*, Roger D. Chamberlain,
Mark A. Franklin, and Brandon Harris
Department of Computer Science and Engineering
Washington University, St. Louis, MO 63130, USA

## ABSTRACT

Profile Hidden Markov models (HMMs) are a powerful approach to describing biologically significant functional units, or *motifs*, in protein sequences. Entire databases of such models are regularly compared to large collections of proteins to recognize motifs in them. Exponentially increasing rates of genome sequencing have caused both protein and model databases to explode in size, placing an ever-increasing computational burden on users of these systems.

Here, we describe an accelerated search system that exploits parallelism in a number of ways. First, the application is functionally decomposed into a pipeline, with distinct compute resources executing each pipeline stage. Second, the first pipeline stage is deployed on a systolic array, which yields significant fine-grained parallelism. Third, for some instantiations of the design, parallel copies of the first pipeline stage are used, further increasing the level of coarse-grained parallelism.

A naïve parallelization of the first stage computation has serious repercussions for the sensitivity of the search. We present a pair of remedies to this dilemma and quantify the regions of interest within which each approach is most effective. Analytic performance models are used to assess the overall speedup that can be attained relative to a single-processor software solution. Performance improvements of 1 to 2 orders of magnitude are predicted.

## Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*pipeline processors*; J.3 [**Computer Applications**]: Life and Medical Sciences; H.3.3 [**Information Systems**]: Information Search and Retrieval

## General Terms

Design, Performance, Algorithms

## Keywords

protein motif, hidden Markov model, HMMER

*Corresponding author, *jbuhler@cse.wustl.edu*.


*ICS06* June 28-30, Cairns, Queensland, Australia.


## 1. INTRODUCTION

Determining the biological functions of DNA and protein sequences is a computationally challenging problem. Over the last decade, the advent of high-throughput techniques for DNA sequencing has caused an exponential growth in the size of biosequence databases (e.g., NCBI's GenBank [11]). Before these sequences are used in biological research, they must first undergo *annotation*, a computational process that recognizes and labels parts of a sequence with known functions or similarity to other, well-studied sequences. The explosion of new sequences has placed progressively greater demands on the computational tools used for annotation, with current techniques often taking hours or days to process batches of newly acquired sequence.

The last decade has also seen steady improvement in computer power. Basic computing technology has roughly doubled in speed every 1.5-2 years. At the same time, specialized architectures now permit relatively easy direct hardware implementation of applications. Field-programmable gate arrays (FPGAs), while typically operating at lower clock frequencies than standard processors, exploit both application parallelism (direct and pipelined) and direct functional logic implementation to outperform general-purpose processors on selected applications.

This paper studies an important, widely used biosequence annotation algorithm based on the mathematical formalism of profile hidden Markov models [10], subsequently referred to as HMMs. We describe the design of a specialized hardware-software pipeline tailored to execute this algorithm – Viterbi decoding [15] of a protein against an HMM – one to two orders of magnitude faster than standard processor-based approaches while producing an equivalent result.

HMMs are used in bioinformatics to represent a sequence pattern, or *motif*, common to several evolutionarily related protein sequences. The HMM formalism describes this pattern using a probabilistic model. As more proteins have been functionally characterized, the number of known motifs has grown, so that the Pfam-A and Pfam-B motif databases [1] now contain HMMs for $10^4$ and $2 \times 10^5$ motifs, respectively.

Biologists compare newly obtained protein sequences to HMMs in a database to determine whether the proteins contain known motifs. Key to this process is the Viterbi decoding algorithm, which evaluates how well any portion of a given protein matches a given motif. Because new protein sequences are usually obtained through whole-genome sequencing and gene finding, it is often necessary to to identify motifs in an organism's entire proteome, ranging from $5 \times 10^3$ proteins for bacteria to $2 \times 10^4$ for human.

General-purpose CPU-based search tools exist to perform motif searches [5, 8], but these tools have limited speed with large protein sets and HMM databases. One such tool, the HMMER software [5]

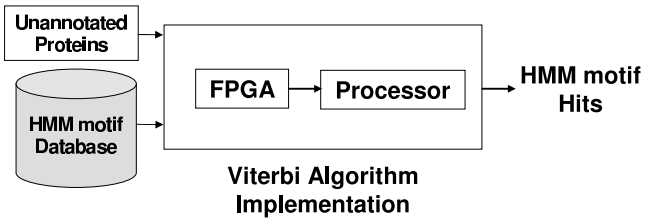


**Figure 1: schema of a special-purpose FPGA-based accelerator for protein motif search.**

requires $10^{-3}$ to $10^{-2}$ seconds on a modern CPU to compare a typical model and protein, and hence 1-500 CPU-days for the above problem sizes. These costs drive demand to increase the performance of HMM search using specialized architectures, such as chip multiprocessors [21], graphics processors [7], or FPGAs, as in the commercial implementation of [17] and this paper.

The work presented here investigates a pipelined design, following the schema shown in Figure 1, that uses both specialized and general-purpose processors to exploit the fine- and coarse-grained parallelism in the Viterbi algorithm. Accessing this parallelism is challenging because the HMMs used in practice induce data dependencies that serialize the algorithm. High-performance search solutions therefore use only a simplified approximation to these HMMs, which lowers the search's sensitivity to biologically meaningful motifs. We address this problem, giving two distinct designs that achieve useful parallelism (and hence high performance) while maintaining high sensitivity. The choice of which design to use in practice depends on the speed and available parallelism of the special-purpose hardware. We quantitatively estimate the performance of each design to project its speedup over a single-processor software solution, obtaining projected speedups of 1-2 orders of magnitude.

The remainder of the paper is organized as follows. Section 2 provides background on use of HMMs for protein motif finding. Section 3 presents the design for a HMM search accelerator, identifies the key serialization problem, and measures the loss of sensitivity incurred by a naïve solution. Section 4 describes the two alternative designs for removing serialization with minimal sensitivity loss. Section 5 compares the two designs and estimates their abilities to speed up HMM search. Finally, Section 6 summarizes our contributions and concludes.

# 2. APPLICATION: HMM SEARCH

This section reviews how HMMs model motifs common to multiple proteins and how search tools use the Viterbi algorithm to compare a protein to an HMM. More detailed background may be found in [4]. We note that use of HMMs to recognize patterns in sequential data is common to many domains besides bioinformatics, including computer vision [19], speech processing [15], and computer security [14]; all these domains are potential targets for search acceleration using techniques like those described here.

## 2.1 HMMs for Protein Motifs

To identify the function of an unannotated protein, biologists search for strings of amino acids in its sequence (i.e., motifs) that resemble the sequences of proteins with known function. The amino acids that make up proteins constitute a twenty-letter alphabet; hereafter, we refer to them abstractly as *symbols*.

Because a motif's exact sequence is rarely critical to its biological function, it may be encoded by slightly different sequences of symbols in different proteins. For example, Figure 2 shows five

| Protein | Motif Sequence |
|---|---|
| RA25_SCHPO | RENSVYL**AKL**A**EQA**E**RY**EEMVENMKK**V**ACSND . . K**L**SV**E** |
| BMH1_YEAST | REDSVYL**AKL**A**EQA**E**RY**EEMVENM**K**T**V**ASSGQ . . E**L**SV**E** |
| 1434_LYCES | REENVYL**AKL**A**EQA**E**RY**EEM**I**EFM**E**K**V**AKTADVEE**L**TV**E** |
| 143T_HUMAN | KTEL**I**QK**AKL**A**EQA**E**RY**DDMATCM**K**A**V**TEQGA . . E**L**SN**E** |
| 1433_XENLA | . . . . . **AKL**S**EQA**E**RY**DDMAASM**K**A**V**TELGA . . E**L**SN**E** |

**Figure 2: a protein motif as it appears in five different proteins. Bold-faced symbols are invariant across all 5 instances; a dot indicates a missing symbol in an instance.**

proteins with a common motif. All instances of the motif have some symbols in common (shown in bold), but evolutionary change may cause some of an instance's symbols to vary, may delete some of its symbols, or may insert new, irrelevant symbols into its sequence. Computational motif finding tools must allow for such variations. Furthermore, several distinct instances of a motif, each with its own variations, may appear sequentially in one protein.

An HMM summarizes the observed variations in a motif across a group of proteins (in the example of Figure 2, across five proteins). An HMM $\mathcal{M}$ is a finite state diagram in which each directed edge from state $q_i$ to state $q_j$ is assigned an integer *transition score* $\tau(q_j \mid q_i)$, associated with movement from $q_i$ to $q_j$. States are either *non-emitting* or *emitting*. If $q_i$ is an emitting state, passing through it emits one symbol; each symbol $a$ has an *emission score* $\epsilon(a \mid q_i)$ associated with this state.

Given an HMM, one can align a protein $s$ to it as a sequence of symbols. Alignment begins at the HMM's initial state $q_0$ and traces a path to the final state that passes through $|s|$ emitting states (producing $|s|$ symbols), plus perhaps some non-emitting states (producing no symbols). Given this path, the score of the aligned sequence is the sum of the scores for all the path's state-to-state transitions, plus the emission scores for each symbol from the state that emitted it. Roughly speaking, sequences emitted via higher-scoring paths correspond to motif instances that are more likely to appear in real proteins and are therefore more "desirable;" a precise probabilistic interpretation is given in [4].

Figure 3 shows the structure of a "Plan7" HMM used by the HMMER software. A motif of length $m$ ($m = 4$ in the figure) contains $m$ "match states" $M_1 \ldots M_m$, where $M_i$ emits the motif's $i$th symbol. A parallel sequence of non-emitting "deletion states" states $D_2 \ldots D_{m-1}$ allows any substring of the motif to be skipped, while another parallel set of "insertion states" $I_1 \ldots I_{m-1}$ can emit symbols between any two motif positions. The non-emitting states $B$ and $E$ act as the model's initial and final states. Finally, state $J$ emits non-motif symbols between successive instances of the motif. The model can therefore describe a string of multiple motif instances in one protein. Note that there exists a feedback path from state $E$ to $B$, which presents hardware architectural difficulties that we address later in the paper.

We note that the Plan7 model as shown in Figure 3 does not align symbols before a protein's first motif instance or after its last instance. An alignment can therefore cover any *substring* of a protein. To simplify exposition, this work omits the (computationally simple) model elaboration needed to score the unaligned ends of a protein.

## 2.2 The Viterbi Algorithm

To determine whether protein sequence $s$ contains a motif that matches model $\mathcal{M}$, HMM search finds the highest-scoring path through $\mathcal{M}$ that matches $s$. If the score $\mathrm{L}(s, \mathcal{M})$ of this path is high enough, the match is reported to the user. In such a case, $\mathcal{M}$ is said to *hit* $s$, and the path indicates which symbol of the protein (if any) corresponds to each position of the motif.

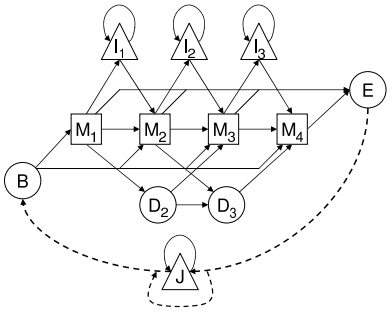


**Figure 3: Plan7 HMM structure for a motif of length $m = 4$. Dashed lines indicate feedback path.**

The Viterbi algorithm [15] calculates $\mathrm{L}(s, \mathcal{M})$ by the following dynamic programming recurrence. Let $\lambda(q, j)$ be the highest score for any path through $\mathcal{M}$ from initial state $q_0$ to a later state $q$ that emits the string of symbols $s[1..j]$. Then for any $q$ and $j$,

$$\lambda(q,j) = \begin{cases} P_e(q,j) & \text{if } q \text{ is emitting} \\ P_n(q,j) & \text{otherwise.} \end{cases} \quad (1)$$

where

$$P_e(q,j) = \max_{q' \in \mathcal{M}} \left[\lambda(q', j-1) + \tau(q \mid q') + \epsilon(s[j] \mid q)\right]$$
$$P_n(q,j) = \max_{q' \in \mathcal{M}} \left[\lambda(q', j) + \tau(q \mid q')\right].$$

The Viterbi recurrence states that the best path ending at state $q$ ends with a single move to $q$ from *some* predecessor state $q'$. If $q$ is emitting (top case), it is assumed to emit the last symbol $s[j]$; otherwise (bottom case), $s[1..j]$ was already emitted by the time the predecessor $q'$ was reached. Any path through $\mathcal{M}$ must end at the unique end state $q_e$ of the model after emitting all of $s$; hence, $\mathrm{L}(s, \mathcal{M}) = \lambda(q_e, |s|)$ is the score of the best path through $\mathcal{M}$ emitting $s$.

For the Plan7 model of Figure 3, the maximizations of Equation (1) are taken over only a constant number of predecessor states; hence, $\mathrm{L}(s, \mathcal{M})$ can be computed in time $\Theta(|s||\mathcal{M}|)$, where $|\mathcal{M}|$ is the number of states in $\mathcal{M}$. If only this score is desired, the computation requires $O(|\mathcal{M}|)$ space; however, reconstructing the path of this best motif uses space $\Theta(|s||\mathcal{M}|)$.

The highest-scoring path is considered a hit only if its score exceeds a threshold $\rho$. This threshold is set as a function of $s$, $\mathcal{M}$, and a user-supplied *E-value* parameter $\mathcal{E}$ that controls how stringently potential hits are filtered[1]. *Lower* $E$-values result in *higher* (i.e., more stringent) thresholds. $\mathcal{E}$ typically ranges from a permissive high of 10 to a stringent low of $10^{-3}$ or less.

# 3. AN HMM SEARCH ACCELERATOR

In this section, we describe how to decompose HMM search into a pipeline of operations. We identify the bottleneck operation and show how this operation must be simplified to expose its parallelism for acceleration in hardware. Simplifying the algorithm and mapping it to efficient hardware while producing nearly the same result as software alone are key challenges addressed in this work.

## 3.1 Two-Stage Pipeline Design

We decompose the comparison of protein $s$ to motif $\mathcal{M}$ into two stages. The first stage, *hit detection*, runs (a limited version of) the Viterbi algorithm to compare $\mathcal{M}$ to $s$, generating a score $\sigma_1$. This score is compared to a threshold $\rho_1$ to determine if the motif hits $s$; if not, the pair is discarded. If $\mathcal{M}$ hits $s$, this pair is passed to a second stage, *path generation*, that runs Viterbi again, this time both producing a score $\sigma$ *and* identifying the location(s) of the motif in $s$. Because the two stages use different variants of the Viterbi algorithm, $\sigma$ may be different from $\sigma_1$ and may be compared to its own threshold $\rho$; if $\sigma \geq \rho$, the hit is reported to the user. Figure 4 illustrates this two-stage design.

[1]Threshold $\rho$ is derived from $\mathcal{E}$ using Karlin-Altschul statistical theory [9].

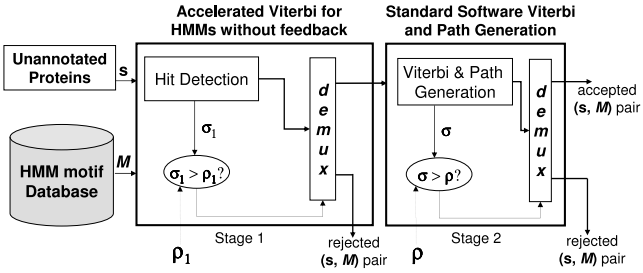


**Figure 4: architecture of a two-stage HMM search pipeline.**

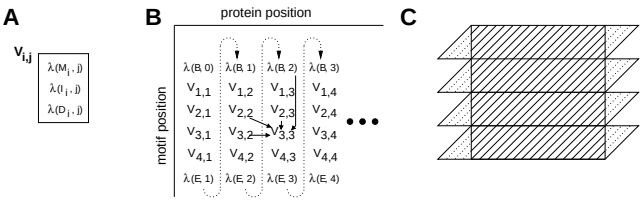


**Figure 5: organization of the Plan7 Viterbi recurrence as a dynamic programming matrix. (A) values grouped into one matrix block $V_{i,j}$; (B) example matrix illustrating data dependencies between blocks; (C) division of a large matrix into horizontal bands.**

Dividing search into hit detection and path generation exposes an opportunity to place each stage on the most appropriate computing resource. Hit detection must be run for *every* pair $(s, \mathcal{M})$, making it a computational bottleneck; in a comparison of 1200 proteins against 7700 motifs in software, we found that search spent over 99% of its time in this stage. However, because hit detection inspects only the score of a potential hit, it can be implemented using only $O(|\mathcal{M}|)$ space (tens of kilobytes). In contrast, path generation runs on only the small fraction of pairs $(s, \mathcal{M})$ that reach it but requires $O(|\mathcal{M}||s|)$ space (megabytes) to locate the motif in the protein. We therefore perform hit detection using a special-purpose hardware architecture that has limited storage but offers the potential for dramatic parallelism, while placing path generation on a slower but more flexible general-purpose processor. The hit detection architecture can economically be realized on an FPGA platform.

## 3.2 Hit Detection by Hardware Dynamic Programming

The special structure of Plan7 HMMs leads to a fruitful approach to accelerating the Viterbi algorithm. We organize the values to be computed by the algorithm into a dynamic programming matrix $V$. Rows of the matrix correspond to positions $i$ in a motif, while columns correspond to amino acid positions $j$ in a protein. Figure 5 illustrates this matrix for the example model of Figure 3.

For a motif of length $m$ and a sequence $s$ of length $\ell$, the matrix $V$ contains $m\ell$ *blocks*. As shown in Figure 5A, each block $V_{i,j}$

holds the values $\lambda(M_i, j)$, $\lambda(I_i, j)$ and $\lambda(V_i, j)$ computed by the Viterbi algorithm. The structure of the model and Equation (1) together imply that the values in $V_{i,j}$ can be computed given the value $\lambda(B, j-1)$ and the three blocks $V_{i-1,j-1}$, $V_{i,j-1}$, and $V_{i-1,j}$. These four dependencies are shown for block $V_{3,3}$ in Figure 5B by solid arrows flowing into it. They correspond to four sets of transitions in the model: from $B$ to $M_3$; from $M_2$, $I_2$, and $D_2$ to $M_3$; from $M_3$ and $I_3$ to $I_3$; and from $M_2$ and $D_2$ to $D_3$. Transitions to $M_3$ or $I_3$ emit a symbol and so advance $j$ from 2 to 3.

The feedback path in Figure 3 introduces additional data dependencies (shown as dotted lines) into the computation. The value $\lambda(B, j-1)$ depends on $\lambda(E, j-1)$ via the path $E \rightarrow B$; it may also depend on $\lambda(E, k)$ for $k < j-1$ via paths that detour through the $J$ state. To satisfy all dependencies, including those induced by the feedback path, the matrix $V$ *must* be filled one block at a time, in column-major order. This restriction eliminates most opportunities to compute multiple matrix blocks at once.

To make the Viterbi algorithm amenable to hardware acceleration, we delete the feedback path from the model, thereby removing the dependencies indicated by the dashed lines. The values $\lambda(B, j)$ can now be precomputed for all $j$, and the remaining data dependencies are all downwards and to the right. This more localized dependency structure permits simultaneous computation of an entire diagonal band of blocks at once. The first step of computation computes $V_{1,1}$; the second computes both $V_{1,2}$ and $V_{2,1}$; the third computes all of $V_{1,3}$, $V_{2,2}$, and $V_{3,1}$, and so forth, with the $d$th step computing $V_{i,d-i+1}$ for all rows $i$.

The feedback-free Viterbi recurrence for Plan7 models is similar to that of the well-known Smith-Waterman algorithm [16] for computing the edit distance between two biosequences. Various groups, including our own, have previously used systolic array designs to to accelerate the Smith-Waterman algorithm [6, 12, 13, 18, 20, 22]. Unlike these implementations, however, the Viterbi algorithm uses different scoring parameters for each step of the dynamic programming algorithm, as well as requiring a larger number of computations per cell of the dynamic programming matrix.

The feedback-free recurrence can be computed in only $m + \ell$ steps (proportional to the input size), compared to $m\ell$ steps for the original recurrence. This speedup requires sufficient hardware resources to compute an entire diagonal band of the matrix at once. These resources consist of $\min(\ell, m)$ copies of a logic circuit to compute one block, organized as a systolic array. In practice, real protein sequences are too long to fit a full $\ell$ copies of this basic circuit on today's FPGA platforms, so the computation may be split into horizontal bands as shown in Figure 5C. Bands are processed serially, with the values in the last row of each band used to initialize the computation in the first row of the next.

The diagonal lines of Figure 5C illustrate the blocks that can be concurrently computed by the systolic array (i.e., in parallel within the hardware). At each end of the band, the systolic array will extend beyond the actual array bounds, as illustrated in the figure. This unnecessary (wasted) computation impacts the overall performance of the algorithm and will be explicitly considered in the performance model described in Section 5.

If the size of the systolic array is large compared to the model size $m$, the above edge effect causes the implementation to become inefficient. To mitigate this effect, one may partition the hardware resources into multiple independent hit detection pipelines, each with a smaller array size, that all run in parallel. We will consider this trade-off between fine- and coarse-grained parallelism as part of our performance model.

## 3.3 Accelerated Hit Detection as a Filter

| E-value $\mathcal{E}$ | Total motif instances with Plan7 models | Instances lost using one-pass hit detection | Est. instances lost over Swiss-Prot database |
|---|---|---|---|
| 0.001 | 7577 | 173 | 4788 |
| 0.01 | 8130 | 208 | 5757 |
| 0.1 | 9628 | 246 | 6808 |
| 1 | 17367 | 562 | 15544 |
| 10 | 77645 | 2612 | 72324 |

**Table 1: loss of sensitivity using one-pass hit detection with $\rho_1 = \rho$ in comparison of Pfam-A to Swiss-Prot.**

Removing the feedback path from the Plan7 model significantly changes its assumptions about motifs. The feedback path expresses the fact that a single sequence can contain multiple instances of a particular motif. This expressiveness is important: many biologically important motifs in real proteins do occur in multiple instances. In contrast, removing the feedback path allows only one instance of the motif to appear in a protein. We therefore refer to such reduced models as *one-pass*, in contrast to the full *multipass* models with feedback.

In our search pipeline, hit detection uses Viterbi with one-pass models as a filter to discard most of its input, while path generation uses multipass models to check the filter's output. Because hit detection does not use the same models as existing software (i.e., HMMER), it may produce false positives – pairs $(s, \mathcal{M})$ where software would report that $s$ does not contain $\mathcal{M}$ – and may incur false negatives – pairs for which software would have found $\mathcal{M}$ in $s$, but which are discarded. The path generation stage recognizes and discards any false positives from the filter but cannot detect its false negatives.

We measured the false negative rate of our pipeline on a comparison of 7677 motif models from Pfam-A to a set of 5898 proteins randomly sampled from the Swiss-Prot protein database [2]. HMMER 2.3.2 was used to generate two copies of each motif model, one with a feedback loop and one without, which were respectively used by path generation and by a software simulation of hit detection. The results were then compared to those of the HMMER software itself, and the false negative rate was quantified for values of $\rho$ corresponding to E-values between 10 and $10^{-3}$. In these experiments, we set $\rho_1 = \rho$.

Table 1 illustrates the loss of sensitivity, measured as the number of missed motif instances, incurred when hit detection is restricted to one-pass models. Although the fraction of misses is small, their absolute number is significant. Extrapolation of our results to the full Swiss-Prot database ($1.6 \times 10^5$ proteins) implies that thousands of significant motif instances would be missed, even for stringent E-values. These lost motifs represent missed opportunities to recognize potential biological functions in proteins. The next section therefore explores strategies to reduce the false negative rate in hit detection to nearly zero, without reducing its throughput or unduly increasing the number of false positives processed by path generation.

# 4. REDUCING FALSE NEGATIVES IN HIT DETECTION

In this section, we consider two approaches to recover the sensitivity lost by using one-pass models in hit detection. The two methods make different tradeoffs between the amount of hardware resources used and the computational burden placed on software path generation.

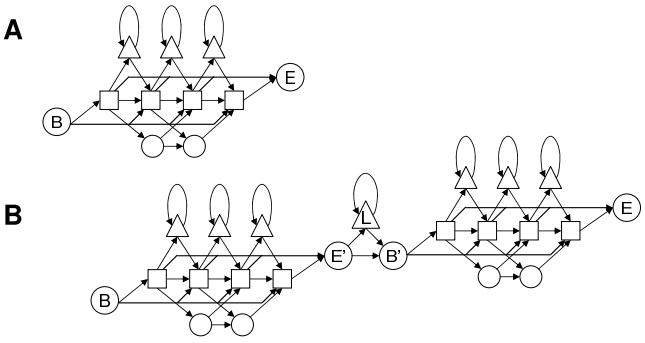


**Figure 6: (A) one-pass simplification of Plan7 model of Figure 3. (B) two-pass simplification, obtained by unrolling the the feedback path of $\mathcal{M}$ once. Linker state $L$ emits any non-motif symbols between the two instances of the motif.**

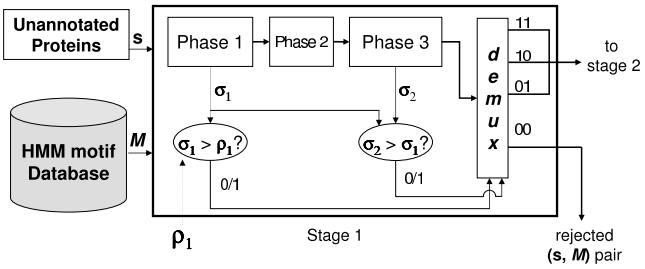


**Figure 7: Modification of hit detection to implement the two-pass heuristic. The three computational phases are described in Section 4.2. A motif $(s, \mathcal{M})$ is passed on to path generation if $\sigma_1 > \rho_1$ (as before) or if the two-pass score $\sigma_2$ is greater than $\sigma_1$.**

## 4.1 Two Ways to Recover Sensitivity

The pipeline of Figure 4 discards a pair $(s, \mathcal{M})$ when the score $\sigma_1$ of the highest-scoring path for protein $s$ through the *one-pass* version of model $\mathcal{M}$ fails to reach the user-supplied threshold $\rho_1$. One approach to improving sensitivity is to make threshold $\rho_1$ less stringent by setting $\rho_1 < \rho$.

Reducing $\rho_1$ can recover false negatives caused by a motif $\mathcal{M}$ that occurs several times, each time with a low score, in one protein $s$. The full Viterbi algorithm computes the total score of all instances of $\mathcal{M}$ in $s$, but the one-pass algorithm scores only the single best instance, which may be responsible for only a fraction of the total score. Reducing $\rho_1$ compensates for this deficiency. On the other hand, reducing $\rho_1$ also increases the number of false-positive pairs $(s, \mathcal{M})$ that pass hit detection but would not pass if the full multipass model were used with the original threshold $\rho$. These false positives are discarded in the path generation stage, but they add to that stage's computational cost.

A second approach to reduce false negatives is to modify hit detection to detect specifically those cases in which the one-pass algorithm is likely to err. False negatives occur when a protein $s$ contains two or more instances of a motif $\mathcal{M}$, but no one instance scores highly enough to pass hit detection's threshold $\rho_1$. To address this problem, we therefore compute the total score of the best *two* instances of motif $\mathcal{M}$ in $s$. If this score exceeds the one-pass score, we believe that $s$ contains multiple instances of $\mathcal{M}$, and so we pass $(s, \mathcal{M})$ through to path generation *regardless* of its one-pass score. We call this approach *two-pass* hit detection.

Two-pass hit detection minimizes false negatives while permitting a lesser reduction in hit detection's threshold $\rho_1$ compared to $\rho$. It therefore eliminates many false positives that would otherwise be incurred by a large reduction in $\rho_1$, reducing the burden on path generation.

## 4.2 Implementing Two-Pass Hit Detection

Two-pass hit detection may be viewed as running the Viterbi algorithm on an extended version of a one-pass Plan7 HMM. The difference between the original one-pass model and this extended version is illustrated in Figure 6. The two-pass model of Figure 6B consists of two copies of the one-pass model of Figure 6A, joined by a *linker* $L$ that permits any number of symbols to be skipped between the two motif instances.

A key observation for efficiently implementing two-pass hit detection is that it is computationally equivalent to running one-pass hit detection twice. Looking more closely at Figure 6B, the score $\max_{1 \le j \le |s|} \lambda(E', j)$ *is* the one-pass score, so it is not necessary to implement the one-pass algorithm separately. Moreover, we can divide the Viterbi algorithm on a two-pass model into three phases: computation of $\lambda(E', j)$ (the first pass), computation of $\lambda(B', j)$ (the linker), and computation of $\lambda(E', j)$ (the second pass), each for all $1 \le j \le |s|$. Each phase's computation depends only on the result of the previous phase, so the three phases can be pipelined.

The logical structure of efficient two-pass hit detection is shown in Figure 7. Phases 1 and 3, the passes through the two motif instances, are implemented using the dynamic programming design of Section 3.2, while phase 2 uses a simplified version of the same design to score all possible paths from $E'$ to $B'$. The one-pass score is available after phase 1, while the two-pass score is available after phase 3. Pipelining the three phases lets them run simultaneously on different models, and no one phase executes more slowly than the computation of the one-pass score alone, so the overall throughput is identical to that of one-pass hit detection.

## 4.3 Sensitivity of One- versus Two-Pass Hit Detection

To assess the behavior of one-pass and two-pass hit detection, we extended our software simulation of hit detection to implement the two-pass algorithm and compared both its sensitivity and its cost in path generation to those of one-pass hit detection.

Sensitivity of hit detection on a set of input pairs $(s, \mathcal{M})$ is measured relative to the output of the HMMER 2.3.2 software with full Plan7 models, which we take to be our standard of correctness. Any pair $(s, \mathcal{M})$ designated as a hit by HMMER but not by our search pipeline is considered a false negative. Because path generation performs the same computation as HMMER, the full pipeline does not produce false positives relative to HMMER, but the false positive rate is reflected indirectly in the computational cost of path generation.

We evaluated one- and two-pass hit detection on a comparison of four sets of 5898 randomly selected sequences from Swiss-Prot against 7677 models from Pfam-A. Evaluation used a 2.8 GHz Intel Pentium 4 CPU running Linux. We measured the behavior of both implementations while varying score thresholds $\rho$ (overall stringency of the search) and $\rho_1$ (stringency of hit detection). Following standard practice for HMMER, these these thresholds were not set directly but were derived (using HMMER's own code) from $E$-values $\mathcal{E}$ and $\mathcal{E}_1$. For a variety of $E$-values, we empirically determined for each hit detection strategy the smallest (most stringent) parameter $\mathcal{E}_1$ for which we observed sensitivity of 1.0, that is, no false negatives relative to HMMER. For these parameters, we measured time spent by the pipeline in unaccelerated path generation.

Table 2 gives the parameters and corresponding average times over the four sets. Variation between sets was less than 5%.

We achieved software-equivalent sensitivity using substantially more stringent (lower) thresholds $\mathcal{E}_1$ for two-pass than for one-pass hit detection. Whereas the one-pass method can recognize weak multi-instance motifs only by raising $\mathcal{E}_1$ enough to detect at least one instance of the motif, the two-pass method can recover motifs in which no one instance is recognized by itself. The two-pass method's more stringent threshold translates to less work for path generation, as indicated by this stage's lower observed running time. The difference between the two implementations becomes more pronounced when the user's threshold $\mathcal{E}$ is itself fairly stringent; that is, when the detected motifs are strong.

# 5. RESULTS

## 5.1 Modeling Accelerator Performance

In this section, we model the performance of accelerators for HMM search based on the one- and two-pass hit detection pipelines of Figures 4 and 7. We assume that hit detection is accelerated using the systolic array design of Section 3.2, while path generation is implemented in software.

### 5.1.1 Hit Detection Performance Model

We first estimate for the one-pass implementation $t_H$, the mean time (over a collection of proteins and models) to perform hit detection for one protein $s$ of length $\ell$ and one model $\mathcal{M}$ of size $m$. We quantify execution time in terms of *cell updates*, where a cell is defined as the computation of a single $\lambda$ value from Equation (1). Note that the blocks $V_{i,j}$ of Figure 5 each contain 3 cells. Each cell update requires constant time, and a systolic array of size $A$ can perform $A$ updates in parallel. Hence, we model the time to process $(s, \mathcal{M})$ as $t_H = \frac{C}{R}$, where $C$ is the average number of cell updates required, and $R$ is the number of updates per unit time.

We estimate the rate of cell updates, $R$, as $A \cdot f_{CLK}$, where $A$ is the number of concurrent cell updates as above, and $f_{CLK}$ is the clock rate. $A$ is proportional to total implementation area.

The number of cell updates $C$ includes both the necessary computation to process $(s, \mathcal{M})$ and the wasted cell updates due to the edge effects illustrated in Figure 5C. The computation to process $(s, \mathcal{M})$ depends on the protein and model sizes. A Plan7 model $\mathcal{M}$ for a motif of length $m$ contains $3m - 3$ core states ($M_i$, $I_i$, and $D_i$) and three non-core states $\{E, J, B\}$. The one-pass version of the model used in hit detection need not perform any computation for states $J$ or $B$. For all remaining states $q \neq E$, $\lambda(q, j)$ can be updated in constant time per $j$; $\lambda(E, j)$ alone requires $\Theta(m)$ operations per $j$. If we treat each update of $E$ as a collection of $m$ updates, then the total number of cell updates is $4m$ per symbol of $s$.

The wasted computation at the end of each band is proportional to the length $A$ of the systolic array. As noted at the end of Section 3.2, this waste can be reduced by partitioning the hardware resources into multiple, smaller pipelines. Let $n_p$ be the number of such pipelines; then the total wasted computation is $4A/n_p$ cell updates per symbol of $s$.

Combining the costs of necessary and wasted computation above, we derive a cell update count for the one-pass model of

$$C_1 = 4\left(m + \frac{A}{n_p}\right)\ell.$$

For two-pass hit detection, the cell update count is computed for the extended model of Figure 6B. In this model, states $E$ and $E'$ require $\Theta(m)$ updates for each symbol of $s$, while states $L$ and $B'$, like the two model cores, are updated once per symbol. We charge edge-effect cycles separately to each of phases 1 and 3 (which assumes that they are implemented by separate arrays) but neglect these cycles for phase 2, since it is not deeply pipelined. The total number of cell updates is therefore

$$\begin{aligned} C_2 &= \left[4\left(m + \frac{A}{n_p}\right) + 4\left(m + \frac{A}{n_p}\right) + 2\right]\ell \\ &= 8\left(m + \frac{A}{n_p}\right)\ell + 2\ell. \end{aligned}$$

### 5.1.2 Path Generation and Combined Performance Models

Let $t_S$ be the mean time to process a pair $(s, \mathcal{M})$ in path generation. To estimate $t_S$, we use a linear model $t_S = \alpha m\ell + \beta$, which is empirically justified by our observations of path generation times on our test workstation.

We also need the fraction $f_{hit}$ of work that hit detection passes on to path generation. For a one-pass implementation, this is simply the fraction of pairs $(s, \mathcal{M})$ that pass the hit detection threshold, while for a two-pass implementation, it also includes pairs that yield a higher score after two passes than after one. In our tests, $f_{hit}$ empirically ranged between $10^{-4}$ and $10^{-2}$, with lower values corresponding to more stringent thresholds $\mathcal{E}_1$.

We estimate overall accelerator performance as follows. Consider an all-to-all comparison of $N_{seq}$ sequences to $N_{mod}$ models. The total time in hit detection is $T_H = t_H \cdot N_{seq}N_{mod}$, while the total time in path generation is $T_S = t_S \cdot f_{hit}N_{seq}N_{mod}$. Because these two stages run on different computational resources, we pipeline them. The total cost of the computation is therefore $T_a = \max(T_S, T_H)$. For comparison, $T_0 = t_S \cdot N_{seq}N_{mod}$ is the time for an unaccelerated comparison without a separate hit detection stage. The ratio $T_0/T_a$ is therefore the speedup obtained through acceleration.

### 5.1.3 Parameter Estimates

We use as our benchmark the comparison of Swiss-Prot to Pfam-A, which determines mean values for sequence length $\ell$ and model size $m$, as well as $N_{seq}$ and $N_{mod}$. The values $\alpha$ and $\beta$ for path generation are inferred from the same set of comparisons, using linear regression from path generation times measured for HMMER 2.3.2 on a 2.8 GHz Intel Pentium 4 workstation with 1 GB RAM, running Linux 2.4.

We estimate our cell update rates for hit detection from a survey of recent literature on systolic arrays for biosequence comparison. The systolic array design technique has been deployed on FPGAs by a number of groups to implement the Smith-Waterman algorithm for biosequence comparison, which has a dynamic programming structure similar to that used in HMM search and is therefore an appropriate basis for performance estimation. Table 3 lists published performance numbers for these previous designs (including our own [20]) and details the technology used, the concurrency $A$ achieved, the clock frequency $f_{CLK}$, and the resulting cell update rate $R$. The range of performance values represents variation in the hardware generation used as well as the specific formulation (affine or linear) of the problem solved.

Given the availability of the newer Virtex 4 line, with its faster clock rates and greater capacity (49,152 slices in the XC4VLX100), cell update rates of $R = 5$ to 20 GCUPS (billion cell updates/second) should be readily achievable in a single FPGA part. Because the systolic array is linear, with limited stage-to-stage communication, we may cascade multiple parts to increase the concurrency $A$.

| $\mathcal{E}$ | One-pass | | Two-pass | |
|---|---|---|---|---|
| | $\mathcal{E}_1$ for Sensitivity 1.0 | Time in Path Generation (hours) | $\mathcal{E}_1$ for Sensitivity 1.0 | Time in Path Generation (hours) |
| 0.001 | 5 | 2.161 | 0.003 | 0.625 |
| 0.01 | 20 | 5.417 | 0.02 | 0.657 |
| 0.1 | 20 | 5.417 | 2 | 1.488 |
| 1 | 30 | 7.440 | 9 | 3.220 |
| 10 | 40 | 9.410 | 40 | 9.411 |

**Table 2: comparison of thresholds $\mathcal{E}_1$ required to obtain perfect sensitivity on test computation with one- and two-pass hit detection.**

| Group | FPGA (slices) | Concurrent cell updates per clock, $A$ | Clock Freq. $f_{CLK}$ (MHz) | Cell update rate, $R$ (GCUPS) |
|---|---|---|---|---|
| VanCourt and Herbordt [18] | XC2VP30 (13,696) | 57 to 126 | 39 to 77 | 2.2 to 9.7 |
| Oliver et al. [12, 13] | XC2V6000 (33,792) | 119 to 252 | 44 to 55 | 5.2 to 13.9 |
| West et al. [20] | XCV1000E (12,288) | 152 | 25 | 3.8 |

**Table 3: published performance of systolic-array Smith-Waterman implementations on Xilinx FPGAs. One slice $\approx$ 2 4-input LUTs + 2 FFs + carry logic. We show total slices available in each part, not the number used by the design. GCUPS: $10^9$ cell updates per second.**

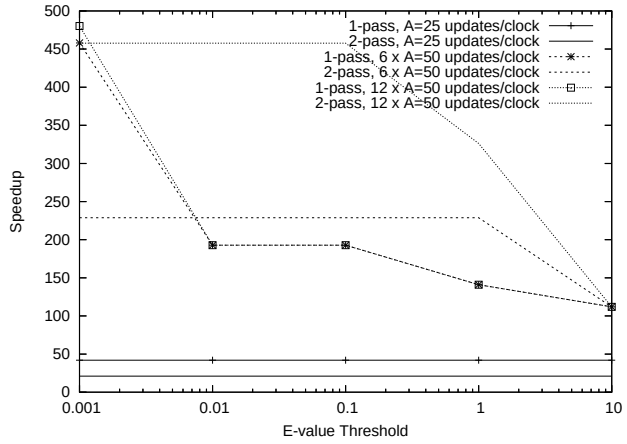


**Figure 8: Speedup of accelerated HMM search over software-only version. Execution times for a range of E-values $\mathcal{E}$ are estimated by performance modeling for various levels of FPGA resources. For each design, we give the number of independent hit detection pipelines and the number $A$ of concurrent updates associated with the systolic array. All designs assume $f_{CLK} = 200$ MHz. Software times are for HMMER 2.3.2 on a 2.8 GHz Intel Pentium 4 system with 1 GB RAM, running Linux 2.4.**

## 5.2 Estimated Speedup

Figure 8 presents the overall speedup estimated from our performance model for HMM search. Different curves represent different assumed cell update rates for the hardware and different implementations (one-pass or two-pass) of hit detection. The points on each curve give speedups for specified E-values $\mathcal{E}$, assuming that the corresponding $\mathcal{E}_1$ is set as in Table 2.

While the $X$ axis is plotted with threshold $\mathcal{E}$ increasing from left to right, it is more helpful to examine the points of each series in order of increasingly stringent hit detection (i.e., from right to left). We begin by examining the most conservative assumption for FPGA resources, 25 concurrent cell updates (i.e., $A = 25$). For these two curves, the pipeline throughput is completely dominated by the hit detection (i.e., hardware) stage, and therefore the one-pass design, with its lower FPGA resource requirements, outpaces the two-pass design.

The next two curves plot the predicted performance with FPGA resources sufficient to support 300 concurrent cell updates (organized as 6 parallel copies of a length $A = 50$ systolic array). For this pair of curves, as we move away from the least stringent threshold on the right towards more stringent thresholds, the one-pass design becomes path generation (i.e., software) limited. This allows the two-pass design, which puts lesser demands on the path generation stage, to outperform the one-pass design. For the most stringent threshold, $\mathcal{E} = 0.001$, the one-pass design again outperforms the two-pass design, as the two-pass design is again limited by hardware resources while the processing requirements placed on the software stage have decreased relative to the less stringent thresholds.

The trend continues when the available FPGA resources are sufficient to support 600 concurrent cell updates (organized as 12 parallel copies of a length $A = 50$ systolic array). Here, the additional hardware resources are effectively exploited by the two-stage design, while the one-pass design is software-limited over most of the range of E-values.

Generalizing the above observations, the results of the figure indicate that there are two regimes for design of hit detection. For lower FPGA resources (readily achievable using only one FPGA), the cost of hardware hit detection dominates that of software path generation. Hence, for a fixed amount of area, the one-pass implementation, which can achieve greater parallelism, outpaces the two-pass implementation, despite the latter's reduction in the cost of path generation. However, as the hardware becomes relatively faster, the cost of path generation becomes a more significant part of the overall running time. For higher FPGA resources (likely requiring multiple FPGAs), path generation costs dominate, throwing the advantage to two-pass hit detection. Overall, the expected speedup over software alone ranges from tens to hundreds of times.

An important caveat in estimating accelerator performance is that both sequences and pHMM parameters must be delivered to the accelerator fast enough to sustain the estimated rate of com-

putation. To determine the feasibility of delivering data to the FPGA, we consider an implementation in which protein sequences are cached on-chip, and the (much larger) block of pHMM parameters is streamed through it. The pHMMs in Pfam have an average size $m = 215$, and a pHMM of size $m$ requires $98m$ bytes for its parameters. Hence, the total size of the 7767 models in Pfam is 159 MB.

An average-sized protein from Swiss-Prot can be compared to an average-sized pHMM from Pfam in 0.003 s on our baseline software platform. Hence, the entire Pfam database can be processed in 23.3 s, implying that the pHMM database is consumed at a rate of 6.82 MB/s. To run 200-fold faster than this, an accelerator would have to deliver pHMM parameters to the FPGA at 1.33 GB/s. This could be achieved by, e.g., streaming the model database from one or more DRAM memories attached to the FPGA. If, as we have suggested above, the hardware is organized in $n_p > 1$ pipelines that compare the same model to $n_p$ proteins in parallel, only hundreds of megabytes per second must be transferred to obtain comparable speedup. In this case, the pHMM parameters could be delivered from the host processor over a PCI-X bus [3].

# 6. CONCLUSIONS

Useful acceleration of HMM motif search requires substantially faster computation with minimal loss of sensitivity versus an unaccelerated implementation. We have described a hardware-software pipeline to parallelize the key bottleneck in the search computation. Our design includes two alternative ways to achieve high performance with essentially no loss of sensitivity in our experiments, even after simplifying the Plan7 HMM structure. Our two strategies allocate work differently to the hardware and software stages of the search engine, so they are suited to implementations with different relative hardware and software speeds. At the same time, we find that in high-performance search implementations, it is important to balance coarse- and fine-grained parallelism in the hardware stage to obtain the highest overall throughput. Our FPGA-based designs are projected to achieve greater than 100-fold speedup over software HMM search implementations.

The speed and area limitations of current-generation FPGA hardware imply that single-chip designs can likely achieve cell update rates of 5-20 GCUPS. At this speed, and with a single modern general-purpose CPU performing path generation, hit detection remains a performance bottleneck, and the less area-intensive one-pass design is preferred. However, given an order of magnitude better FPGA performance, the CPU becomes the bottleneck, and the two-pass implementation delivers better performance by reducing the load on path generation. This level of FPGA performance is likely achievable today through a combination of careful design for fast clocking and multiple-FPGA solutions.

Our work suggests several avenues for future study. First, we plan to elaborate our high-level design to produce a more complete implementation of FPGA-based hit detection, which we can use to validate our performance model. A complete design will address not only the fundamental issues discussed here but also the costs associated with storing data on the FPGA and delivering it to the systolic array blocks. Second, the trade-off between fine- and coarse-grained parallelism should be optimized for best overall performance. We chose a somewhat arbitrary maximum size for each systolic array; higher performance may be possible by optimizing this size for a given overall chip area. Finally, we wish to reapply the lessons learned from this work to accelerate HMM-based search applications in other problem domains, such as knowledge extraction from databases of stored video, audio, or text.

# 7. ACKNOWLEDGMENTS

This work was supported by NSF awards ITR-427794 and DBI-0237902.

# 8. REFERENCES

[1] A. Bateman, L. Coin, R. Durbin, R. D. Finn, V. Hollich, S. Griffiths-Jones, A. Khanna, M. Marshall, S. Moxon, E. L. L. Sonnhammer, D. J. Studholme, C. Yeats, and S. R. Eddy. The Pfam protein families database. *Nucleic Acids Research*, 32:D138–41, 2004.

[2] B. Boeckmann, A. Bairoch, R. Apweiler, M. C. Blatter, A. Estreicher, E. Gasteiger, M. J. Martin, K. Michoud, C. O'Donovan, I. Phan, S. Pilbout, and M. Schneider. The Swiss-Prot protein knowledgebase and its supplement TrEMBL in 2003. *Nucleic Acids Research*, 31:365–70, 2003.

[3] R. Chamberlain and B. Shands. Streaming data from disk store to application. In *Proc. 3rd Int'l Workshop on Storage Network Architecture and Parallel I/Os*, pages 17–23, St. Louis, MO, 2005.

[4] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, New York, 1998.

[5] S. Eddy. HMMER: Sequence analysis using profile hidden Markov models, 2004. http://hmmer.wustl.edu.

[6] D. T. Hoang. Searching genetic databases on Splash 2. In *Proc. of IEEE Workshop on Field-Programmable Custom Computing Machines*, pages 185–192, 1993.

[7] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMER: a streaming HMMer-search implementation. In *Proc. IEEE Supercomputing 2005*, Seattle, WA, 2005.

[8] R. Hughey and A. Krogh. Hidden Markov models for sequence analysis: extension and analysis of the basic method. *CABIOS*, 12:95–107, 1996.

[9] S. Karlin and S. F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc. Nat'l Acad. Sci.*, 87(6):2264–2268, Mar. 1990.

[10] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden Markov models in computational biology: applications to protein modeling. *Journal of Molecular Biology*, 235:1501–31, 1994.

[11] National Center for Biological Information. Growth of GenBank, 2005. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[12] T. Oliver and B. Schmidt. High performance biosequence database scanning on reconfigurable platforms. In *Proc. of 4th IEEE Int'l Workshop on High Performance Computational Biology*, Apr. 2004.

[13] T. Oliver, B. Schmidt, and D. Maskell. Hyper customized processors for bio-sequence database scanning on FPGAs. In *Proc. of ACM/SIGDA 13th Int'l Symp. on Field-Programmable Gate Arrays*, pages 229–237, Feb. 2005.

[14] D. Outston et al. Application of hidden Markov models to detecting multi-stage network attacks. In *Proc. 36th Hawaii Int'l Conf. on System Sciences*, pages 334–44, 2003.

[15] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77:257–86, 1989.

[16] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*,

147(1):195–97, Mar. 1981.

[17] Timelogic DeCypherHMM solution, 2004. http://www.timelogic.com/decypher_hmm.htm.

[18] T. VanCourt and M. C. Herbordt. Families of FPGA-based algorithms for approximate string matching. In *Proc. of 15th IEEE Int'l Conf. on Application-Specific Systems, Architectures, and Processors*, pages 354–364, Sept. 2004.

[19] J. Vlontzos and S. Kung. Hidden Markov models for character recognition. *IEEE Transactions on Image Processing*, 1(4), 1992.

[20] B. West, R. D. Chamberlain, R. S. Indeck, and Q. Zhang. An FPGA-based search engine for unstructured database. In *Proc. of 2nd Workshop on Application Specific Processors*, pages 25–32, Dec. 2003.

[21] B. Wun, J. Buhler, and P. Crowley. Exploiting coarse-grained paralellism to accelerate protein motif finding with a network processor. In *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques*, pages 173–84, St. Louis, MO, 2005. IEEE.

[22] Y. Yamaguchi, T. Maruyama, and A. Konagaya. High speed homology search with FPGAs. In *Proc. of Pacific Symp. on Biocomputing*, pages 271–282, 2002.