

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ACCELERATION OF PROFILE-HMM SEARCH FOR PROTEIN SEQUENCES
IN RECONFIGURABLE HARDWARE

by

Rahul Pratap Maddimsetty

Prepared under the direction of Dr. Jeremy D. Buhler

A thesis presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

May 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

ACCELERATION OF PROFILE-HMM SEARCH FOR PROTEIN SEQUENCES
IN RECONFIGURABLE HARDWARE

by

Rahul Pratap Maddimsetty

ADVISOR: Dr. Jeremy D. Buhler

May 2006

Saint Louis, Missouri

Profile Hidden Markov models are highly expressive representations of functional units, or motifs, conserved across protein sequences. Profile-HMM search is a powerful computational technique that is used to annotate new sequences by identifying occurrences of known motifs in them. With the exponential growth of protein databases, there is an increasing demand for acceleration of such techniques. We describe an accelerator for the Viterbi algorithm using a two-stage pipelined design in which the first stage is implemented in parallel reconfigurable hardware for greater speedup. To this end, we identify algorithmic modifications that expose a high level of parallelism and characterize their impact on the accuracy and performance relative to a standard software implementation. We develop a performance model to evaluate any accelerator design and propose two alternative architectures that recover the accuracy lost by a basic architecture. We compare the performance of the two architectures to show that speedups of up to 3 orders of magnitude may be achieved. We also investigate the use of the Forward algorithm in the first pipeline stage of the accelerator using floating-point arithmetic and report its accuracy and performance.

To Chandana, twelve hours away.

And the sun that never set.

Contents

List of Tables	v
List of Figures	vi
Acknowledgments	vii
1 Background and Significance	1
1.1 Biological Sequence Analysis	2
1.2 Classical Sequence Alignment Algorithms	3
1.3 Profile-HMM Search	7
1.4 Motivation and Related Work	13
2 Accelerator Design	17
2.1 Profile-HMM Search Acceleration Issues	17
2.1.1 A Two-Stage Design	18
2.1.2 Data Dependences in the Viterbi Algorithm	19
2.1.3 Parallelizing Hit Detection	21
2.2 Performance Model	25
2.2.1 Input Size and Parameters	27
2.2.2 Accuracy	27
2.2.3 Execution Time Speedup	29
2.2.4 Software Timing Estimates	31
2.2.5 Hardware Timing Estimates	31
2.2.6 Banded Computation and Edge Effects	33
3 1-pass and 2-pass Architectures	37
3.1 1-pass Architecture	37
3.1.1 Relaxation of Hit Detection Threshold	38
3.1.2 Derivation of Cell Update Count	39

3.2	2-pass Architecture	40
3.2.1	Unrolling the Plan7 Feedback Path	41
3.2.2	Derivation of Cell Update Count	43
3.3	Experimental Results	44
3.3.1	Methodology	44
3.3.2	Comparative Accuracy	46
3.3.3	Comparative Speedup	48
3.3.4	An Alternative Route to Estimating Speedup	50
4	Accelerating the Forward Algorithm for Hit Detection	53
4.1	Motivation	53
4.2	Design Issues	55
4.2.1	Recurrences in Probabilistic Form	55
4.2.2	Computing with Logarithms of Probabilities	56
4.2.3	Forward Algorithm with Floating-Point Probabilities	58
4.3	Experimental Results	60
5	Conclusion	64
	References	67
	Vita	70

List of Tables

2.1	Accuracy loss from eliminating Plan7 feedback path in Hit Detection	24
3.1	Comparison of accuracy of 1-pass and 2-pass accelerators	47
3.2	Performance of systolic array Smith-Waterman implementations . . .	51
4.1	Accuracy of 1-pass accelerator for the Forward algorithm	62

List of Figures

1.1	Growth of the computer-annotated TrEMBL protein database	3
1.2	Growth of the GenBank DNA sequence database	4
1.3	Pairwise alignment of two protein sequences	4
1.4	An example of a protein motif	5
1.5	Building a profile from the multiple alignment of a family of 5 related proteins	7
1.6	The Plan7 HMM structure	9
1.7	Example of a profile-HMM search	11
2.1	Profile-HMM search pipeline	19
2.2	Viterbi dynamic programming matrix for profile-HMM search	20
2.3	Parallel computation of the feedback-free Viterbi Matrix	23
2.4	Relationship between quantities in the performance model	26
2.5	Banded computation and edge effects due to parallel cell updates	34
3.1	1-pass and 2-pass HMM structures	40
3.2	Hit Detection in a 2-pass Architecture	43
3.3	Comparison of speedup of 1-pass and 2-pass accelerators	49
3.4	An alternative comparison of speedup of 1-pass and 2-pass accelerators	52
4.1	Comparison of speedup of single and double precision Forward-based accelerators	62

Acknowledgments

This Masters thesis marks the beginning in my life of what I hope will be a continuing process of contribution to the global body of scientific knowledge and technology. For 18 years of my life, I have been nurtured and shaped by an education system and academic community to which I owe a great deal of the encouragement and support that has made this work possible; and for all 23 years, I have had a family and friends to whom I owe every bit as much.

I thank my teachers at the Hyderabad Public School for helping me discover the joys of mathematics, physics and computer programming and the true meaning of an all-round education. I also thank the faculty at the Institute of Technology Madras for helping a student of Chemical Engineering follow his true passion for Computer Science. I thank the faculty at Washington University for the tremendous learning experience that these two years have been. In particular, I thank my advisor, Dr. Jeremy Buhler, for introducing me to, and guiding me through, the wonderful world of research. I also thank Dr. Roger Chamberlain, and my colleagues Arpith Jacob and Brandon Harris for the perspective I gained through numerous discussions with them for this work. I also thank the National Science Foundation for supporting my research under grant NSF ITR-427794.

I thank my parents for their endless faith and support; for giving me the gift of music and the arts; and for encouraging my every creative pursuit. I thank my brother for being my inspiration in every walk of life and for the extraordinary enthusiasm with which he shares his knowledge and perspective with me. Finally, the last two years have been an enriching time spent outside the familiarity of my own country. I thank the community at Washington University, the city of Saint Louis, and the people of the United States for making this the next best place to home.

Rahul Pratap Maddimsetty

Washington University in Saint Louis
May 2006

Chapter 1

Background and Significance

This work is concerned with the application of high performance computing to the study of biological sequences. We examine algorithms employed in profile-HMM search – a technique used to detect features in protein sequence in order to infer their biological function. We identify approaches to exploiting parallelism for the acceleration of one profile-HMM search algorithm with the aid of reconfigurable hardware in order to meet the demand for high-throughput computational tools for this task. To this end, we describe in detail two alternative accelerator architectures and compare their performance on large input using a performance model we developed. Finally, we also investigate the acceleration of another profile-HMM search algorithm in parallel reconfigurable hardware using the same performance model.

In this chapter we introduce the field of biological sequence analysis and remark on its increasing dependence on computational tools. In particular, we compare techniques that are applied in protein motif finding, such as alignment and profile-HMM search, and address the growing need to accelerate them. We describe the general approach taken by previous work in the acceleration of alignment algorithms and identify challenges specific to the acceleration of profile-HMM search.

1.1 Biological Sequence Analysis

Biological sequences are symbolic linear representations of the chemical structures of biological molecules. Hence they offer the potential to be analyzed with the help of computational tools in order to infer biological properties of the molecules they represent, rather than the molecules themselves being treated and studied in a laboratory. This approach is especially helpful because biological functions are conserved through evolution across different DNA sequences or proteins.

The goal of biological sequence analysis is to identify sequence elements whose recurrence in multiple sequences signifies similar biological function, and to characterize new sequences based on knowledge about the biological functions of sequence elements found to be conserved in them.

The growing interest of biologists in DNA and protein sequences has seen the attendant proliferation of engineering solutions to aid in their sequencing as well as the study of their physico-chemical properties. However, the greater the information generated by sophisticated equipment such as sequencers and spectrometers, the greater the need for computational tools to consolidate and analyze it.

The development of high-throughput software applications to carry out biosequence analysis tasks that were previously carried out manually, such as sequence assembly, has led to an explosive growth in the size of sequence databases such as GenBank [14] for DNA sequences and Swiss-Prot/TrEMBL [3] for protein sequences. Figure 1.1 shows the growth of the number of entries in the computer-annotated protein database, TrEMBL, between 1996 and 2006, reproduced from [7]. Growth since 2000 has been nearly exponential, doubling approximately every two years. Figure 1.2

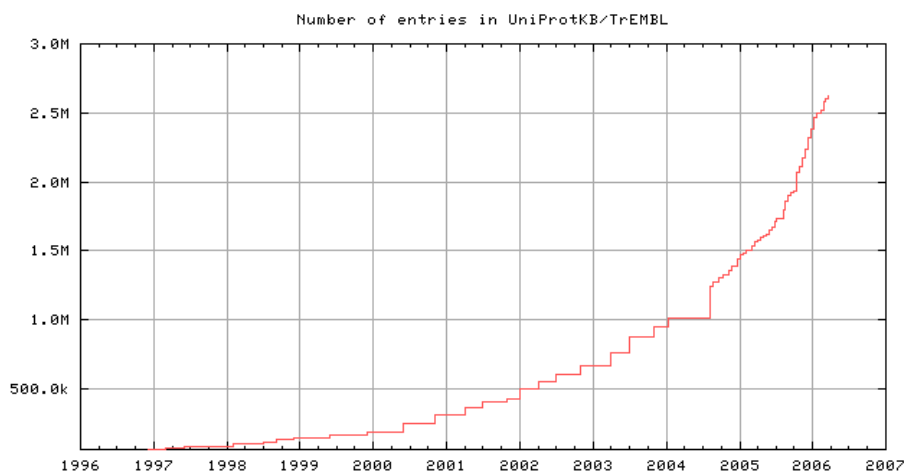


Figure 1.1: Growth of the computer-annotated TrEMBL protein database between 1996 and 2006.

shows the growth of the GenBank DNA sequence database between 1982 and 2004. The growth rate after the sudden surge around 1999 has been sustained ever since. It is clear, therefore, that biologists are increasingly dependent on computational techniques for the analysis and annotation of sequences in these databases.

1.2 Classical Sequence Alignment Algorithms

The majority of computational techniques that have been developed for biosequence analysis attempt to infer biological function or structure of a newly discovered or previously unstudied sequence, called the *query*, by performing a similarity search against *databases* of well-known, extensively classified and annotated sequences. Such a similarity search produces an *alignment*, i.e. seeks to align similar (and therefore, likely biologically conserved) substrings of the sequences being compared. Figure 1.3 illustrates the result of a similarity search between two short protein sequences by

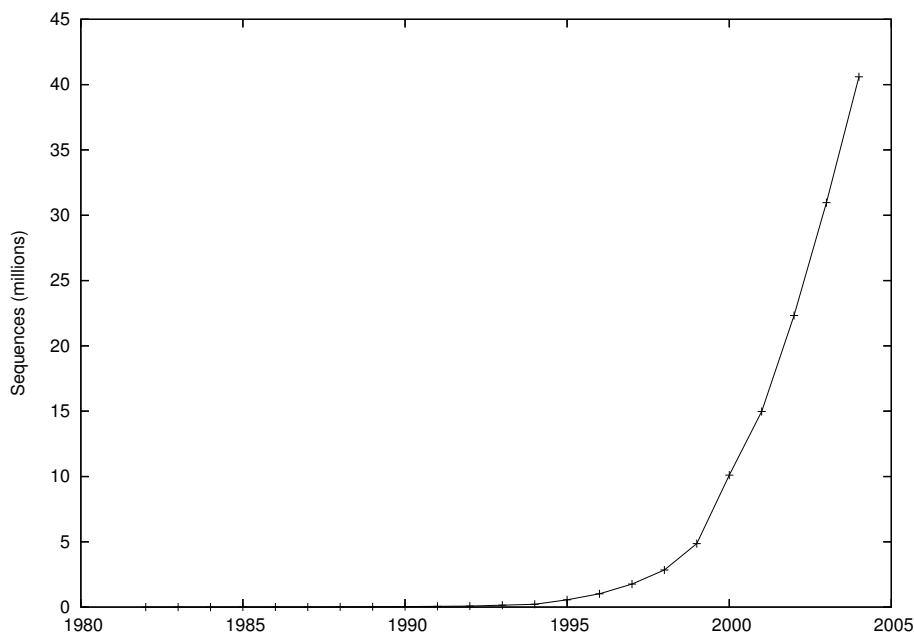


Figure 1.2: Growth of the GenBank DNA sequence database between 1982 and 2004.

Query: RAQEEMVENACNSDKLA
Database: TASESQEDMVCNS

Alignment

. . . . RAQE*E*MAENACNSDKLA
TASES . QEDMV . . . CNS

Figure 1.3: A pairwise alignment of two short protein sequences showing conserved regions in bold face. Substitutions in these regions are shown in italics. Dots indicate gaps caused by insertions or deletions.

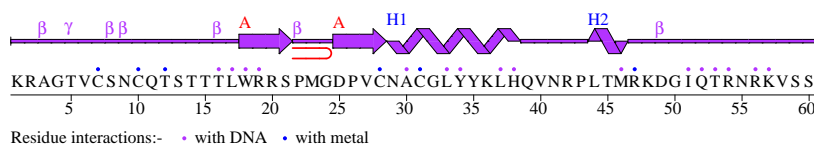
performing insertions and deletions over dissimilar regions in order to align the similar regions shown in bold face. Note that apart from identical symbols, the similar regions also contain substitutions (shown in italics in the query). Certain substitutions are more likely than others - an evolutionary fact reflected in the scoring system used to choose among alternative alignments.

The various biological functions of a protein reside in motifs that correspond to specific regions in its sequence. For instance, Figure 1.4 shows the sequence of the protein

A

Protein Sequence GATA1_CHICK	Position
MEFVALGGPDAGSPTPFPEAGAF LGLGGGERTEAGLLA	40
SYPPSGRVSLVPWADTGT LGTPQWVPPATQMEPPHYLELL	80
QPPRGSPHPSSG PLLPLS SGPPPCEARECVNCGATATPL	120
WRRDGTGHYLCNACGLYHRLNGQNRPLIRPKKRLLVSKRA	160
GTVC SNCQTSTTTLWRRSPMGDPVCNACGLYYKLHQVNRPL	200
LTMRKDG IQTRNRKVS SKGKKRRPPGGGNPSATAGGGAPM	240
GGGGDP SMPPPPPP PAAAPPQSDALYALGPVVLSGHFLPF	280
GNSGGFFGGGAGGYTAPPGLSPQI	304

B



C



Figure 1.4: (A) The full sequence of the protein GATA1_CHICK [22] with the location of the DNA binding GATA zinc finger domain shown in bold face. (B) The secondary structure exhibited by the 60 residues of the GATA zinc finger in GATA1_CHICK (reproduced from [19]). Residue interactions with DNA and metal are also shown. (C) Molecular structure of the zinc finger domain determined by Multidimensional NMR spectroscopy (reproduced from [4]).

GATA1_CHICK and the location, function, and domain structure of the instance of the DNA binding GATA zinc finger motif contained within it. However, the sequence of amino acids that makes up each motif is not an exact signature of its intended biological function but undergoes evolutionary changes. Figure 1.5 shows the same motif occurring in five different proteins. Although it is largely conserved across them, the exact sequence is not identical. Hence, the occurrence of a slightly mutated but otherwise highly similar sequence of amino acids in two proteins suggests that the two sequences are evolutionarily related and so may share a common biological function through the motif occupying that region. Sequence alignment, therefore, is widely used in protein motif finding.

Sequence alignment algorithms such as Needleman-Wunsch [15], Smith-Waterman [20], and the BLAST family of programs [1] have long been used for protein motif-finding by performing pairwise alignment of each query against every sequence in the database, thus identifying those sequences in the database that are most closely related to various regions of the query. This information can then be used to create a full structural and functional annotation of the motifs present in the query. These algorithms use dynamic programming with a simple scoring system based on fixed substitution matrices and gap penalties.

The scores of a substitution matrix express the similarity scores associated with substitution of a pair of symbols. When a symbol appears unchanged in both the query and the database, the substitution score is highest. When it is replaced by a different symbol, the substitution score is lower – and its value is directly related to the likelihood of the particular substitution. When a symbol appears at a particular position in one sequence but not in the other, there is said to be a gap, and this contributes negatively to the similarity score. The gap penalty is said to be *linear* if it is independent of whether the gap is newly created or an extension of an existing gap, and *affine* if it assigns a different penalty to the opening of a gap than to the extension of an already open one.

The following is a simplified form of the dynamic programming recurrence for these algorithms with a linear gap penalty. Each cell $V(i, j)$ contains the score of the best alignment of the first i symbols of the query, q , to the first j symbols of the database sequence, d . $S(q_i, d_j)$ is the score for substitution of symbol y in the database with x in the query. p_g is the gap penalty. A detailed discussion can be found in [5].

Protein	Motif Sequence
RA25_SCHPO	RENSVYL AKLAEQAERYEEM VENMKKVACSND . . KLSVE
BMH1_YEAST	REDSVYL AKLAEQAERYEEM VENMKTVAS SGQ . . ELSVE
1434_LYCES	REENSVYL AKLAEQAERYEEM IEFMEKVAKTADVEELTVE
143T_HUMAN	KTEL IQK AKLAEQAERYDD MATCMKAVTEQGA . . ELSNE
1433_XENLA AKLSEQAERYDD MAASMKA VTELGA . . ELSNE

Figure 1.5: A multiple alignment of a family of 5 related proteins which can be used as the basis for building a profile representing the family. The residues in bold face are conserved in all sequences and are therefore likely to appear in every instance of the motif. Dots indicate gaps – at these positions, a related motif is likely to contain randomly inserted residues or no residues at all.

$$V(i, j) = \max \begin{cases} V(i-1, j-1) + S(q_i, d_j), & \text{substitution} \\ V(i-1, j) - p_g, & \text{gap in database – insertion} \\ V(i, j-1) - p_g, & \text{gap in query – deletion} \end{cases} \quad (1.1)$$

1.3 Profile-HMM Search

While pairwise techniques directly compare one sequence to another one symbol at a time, another paradigm compares a query to a probabilistic representation of several proteins of the same family. Since all the members of a family are mostly similar to each other, it is possible to construct a common *profile* on which they may each be considered variations. Such a profile of the family will then be general enough that any related motif occurring in a new sequence will also be detectable as a variation on it. Figure 1.5 shows a multiple sequence alignment of the first few residues of a family of 5 related proteins. A multiple sequence alignment can be used to build a profile that describes each position of the motif.

A common kind of profile representation is the *consensus sequence*, which simply reflects the most commonly occurring residue at each position. However, several more expressive profile representations can capture more detailed statistics of each position of the motif. One such representation is called the profile-HMM and represents a motif as a hidden Markov model. Mathematically, a profile-HMM is represented as a state diagram where a directed edge from state q_i to q_j has an associated transition score, $a(q_i, q_j)$. Further, state q_i may be *emitting*, meaning that a transition into it outputs a symbol x with an associated emission score, $b(q_i, x)$. Figure 1.6 shows the Plan7 HMM structure used by the profile-HMM search program HMMer [6] to represent a motif of length $m = 4$ residues. There are always as many M states as the motif length, m ; I states are numbered from 1 to $(m - 1)$, and D states from 2 to $(m - 1)$.

Profile-HMM search is used to solve the protein motif finding problem. Computationally, it is the process of scoring a query sequence for its similarity to the motif represented by a profile-HMM and locating the various high-scoring instances of the motif in the sequence. Throughout the remainder of this work, we refer to the overall profile-HMM search computation as simply a *search*, where a unique individual search is identified by its input query sequence and profile-HMM pair. For instance, a high-scoring search is one in which the similarity score of the motif and the query sequence is determined to be high.

Profile-HMM search employs algorithms previously used in fields such as signal processing, speech recognition [18] and natural language processing. In proteins, the process involves determining the likelihood score of emitting the query sequence through

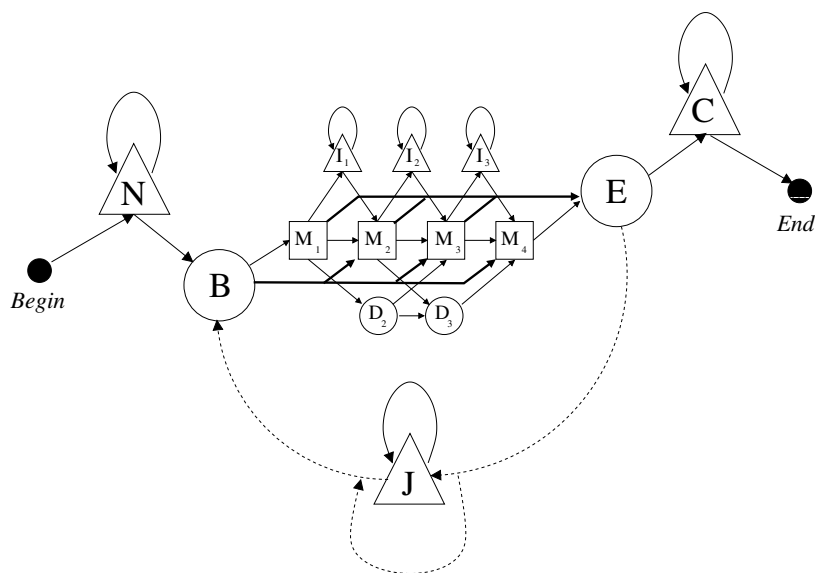


Figure 1.6: The Plan7 HMM structure used by HMMer for a motif of length 4 residues. The M_k , I_k and D_k states respectively model a substitution (or match), insertion or deletion at the k th motif position. The N and C states model randomly emitted N- and C-terminal residues, i.e. before and after significant copies of a motif. The B and E states model the beginning and end of each individual copy of a motif. The J state allows random emission of residues between two copies of a motif. Dashed lines represent the feedback path between multiple copies of a motif.

a state path (or several state paths) over each profile-HMM (representing a particular family of proteins) in the database. This proceeds by considering all state paths, beginning at the *Begin* state of the profile-HMM and ending at its *End* state, that emit precisely the sequence of symbols corresponding to the query. The overall score of each path is the sum of the scores for each state transition on the path, plus the emission scores for each symbol emitted by an emitting state. Figure 1.7 illustrates one out of several possible state paths over a profile-HMM that emit the query sequence. The score computation and motif location reported by profile-HMM search are also indicated.

The Viterbi algorithm [18] is used to determine the single best scoring path over the profile-HMM that emits the query sequence. The Forward algorithm [18] is used to determine the total score of all paths over the profile-HMM that emit the query sequence. While the recurrences of these algorithms are quite similar, they provide two fundamentally different measures of similarity between the query sequence and the database. For a major portion of this work, we shall concern ourselves with the Viterbi algorithm, dealing separately with the Forward algorithm in Chapter 4.

Although the dynamic programming recurrence for profile-HMM search bears outwardly resemblance to Equation 1.1 in that it considers matches (or substitutions), insertions and deletions, it has a more complicated internal structure due to the inadmissibility of certain state transitions and the distinction between emitting and non-emitting states. The shapes of the individual states as well as transition edges between them in Figure 1.6 reflect this structure – the circular *B*, *E*, and various *D* states are non-emitting. The triangular *N*, *C*, *J* and various *I* states emit non-motif symbols whereas the various square *M* states emit motif symbols. Further, passing

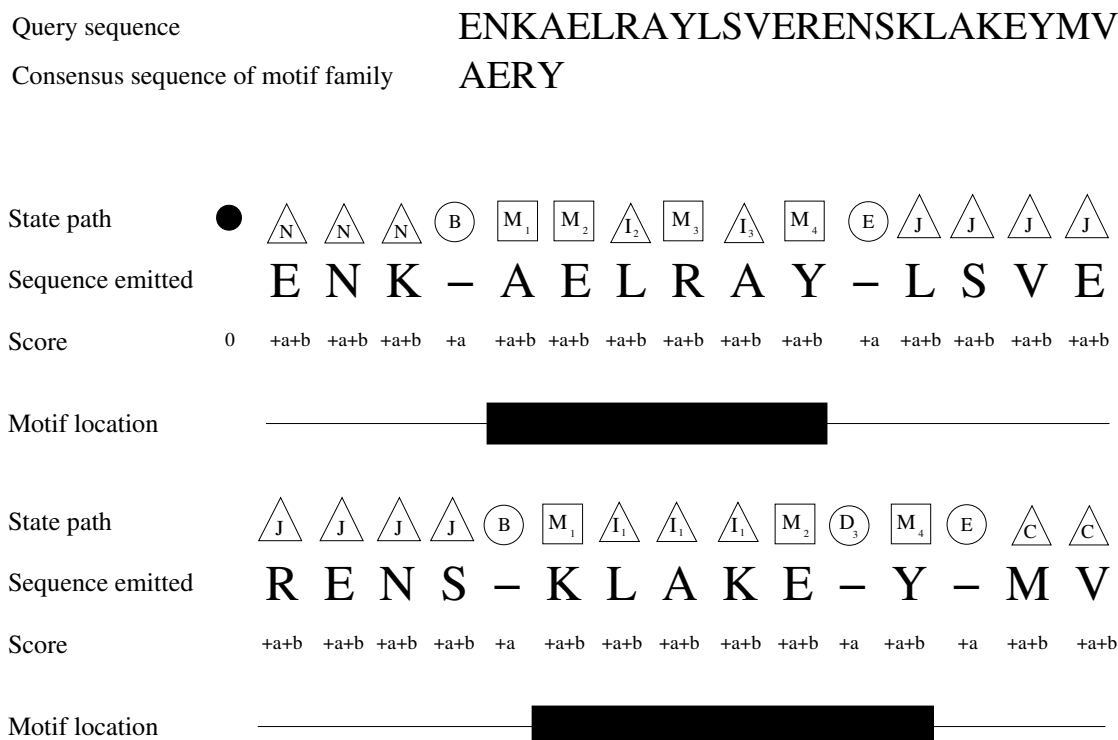


Figure 1.7: An example of a profile-HMM search showing a state path over a profile-HMM of size $m = 4$ that emits the given query sequence. A hyphen in the emitted sequence, Y , indicates a non-emitting state at that position in the state path, Q . Emitting states at position i in the state path contribute a transition term, $a(Q[i-1], Q[i])$, as well as emission term, $b(Q[i], Y[i])$ (together shown simply as $+a+b$), to the overall score. Non-emitting states contribute only the transition term, $a(Q[i-1], Q[i])$, shown simply as $+a$. Each region of the sequence emitted between successive B and E states is interpreted as an instance of the motif. Note that the characters emitted by Match states are not necessarily identical to the character at the corresponding positions of the consensus sequence of the motif family. Also note that transitions into the Insert states do not advance the subscript whereas those into the Match and Delete states do.

through the M_k and D_k states advances from the k th to the $(k + 1)$ th position of the motif's profile representation, while passing through the I_k state does not. This is seen from the fact that a transition into an I state does not advance the subscript whereas transitions into M and D states do. The state path of Figure 1.7 further clarifies this point.

Let Q_i be the set of states $\{q' | q' \rightarrow q_i \text{ is a valid transition}\}$. Then, the Viterbi score of the dynamic programming cell, $V(i, j)$, is the score of the best path that emits the first j symbols of the query sequence, s , and ends up in state q_i of the profile-HMM. Note that if, for instance, $q_i = M_k$ or I_k or D_k , then this means the profile-HMM search has already gone over k symbols from the profile representation of the HMM and needs to go over $(m - k)$ more symbols to find a whole motif. The Viterbi recurrence is given by:

$$V(q_i, j) = \max \begin{cases} \max_{q' \in Q_i} V(q', j - 1) + a(q', q_i) + b(q_i, s_j), & \text{if } q_i \text{ is an emitting state} \\ \max_{q' \in Q_i} V(q', j) + a(q', q_i), & \text{otherwise} \end{cases} \quad (1.2)$$

The Viterbi score for the entire query sequence s , of length l , against the profile-HMM is simply given by $V(End, l)$.

Profile-HMM search is favored by biologists for a variety of reasons. The probabilities that are computed for the various symbol emissions and state transitions depend on the protein sequences which were used in building the profile-HMM. Given more proteins of the same family, a more sensitive profile-HMM can be built, which can in turn more accurately identify unseen instances of the same motif. This is in

contrast with a scoring system using a fixed substitution matrix and gap penalty based on global protein sequence statistics. Also, profile-HMMs, on account of using probabilistic scoring, are better tuned to detecting weakly conserved yet biologically significant motif sequences than simple pairwise alignment.

1.4 Motivation and Related Work

Profile-HMM search based software tools such as HMMer and SAM [11] have gained popularity among biologists, but the rate of growth in their input sizes (i.e. biosequence and profile databases) outpaces the rate of improvement of processor speeds according to Moore's law. As of 2004, performing a complete comparison of the Swiss-Prot protein database (containing 1.6×10^5 sequences) against all 7.7×10^3 profile-HMMs built from the Pfam-A [2] seeded alignments could take nearly 50 days on a modern CPU running HMMer. This running time can increase by two orders of magnitude if the much larger, computer-curated, TrEMBL and Pfam-B databases are searched instead – making the task enormous even for large supercomputing clusters.

Specialized hardware architectures have been used to exploit coarse-grained parallelism in profile-HMM search accelerators such as JackHMMer [25], using network processors; and ClawHMMer [10], a streaming implementation for graphics processors. However, these achieve speedups of only up to an order of magnitude relative to a software implementation of HMMer.

Another approach to acceleration exploits fine-grained parallelism by decomposing the entire algorithm into stages and implementing the most time consuming (yet simpler

and/or inherently parallelizable) parts of these algorithms in parallel hardware, while non-parallelizable components are executed in software running on a general-purpose processor. This approach leads to a simple design whose speedup is easily measurable by Amdahl's law.

Previous work demonstrates the success of such an approach in the acceleration of Smith-Waterman [9, 16, 17, 23, 24] and BLAST [13] using FPGA hardware in achieving speedups of up to two orders of magnitude relative to software-only implementations running on general-purpose processors, making such a solution much faster and more cost-effective than a large supercomputing cluster or cluster of graphics or network processors. The Viterbi algorithm also exhibits a high-level structure that may be exploited by a similar acceleration strategy. It may be implemented using a two-stage pipeline – the first stage, implemented in parallel hardware, filtering out all low-scoring searches based on a preliminary similarity score computation; and the second, on a general purpose processor, performing a detailed similarity search on the input it receives from the first.

However, owing to specific difficulties detailed in Section 2.1, acceleration of profile-HMM search first requires simplifying the first stage in order to make it implementable in parallel hardware. In doing so, trading off some accuracy for speed is usually inevitable. The extent to which the first stage is simplified then affects performance on two fronts: the accuracy of the overall accelerator; and the overall speedup achieved by the accelerator relative to an unaccelerated software-only implementation.

In contrast, in the case of Smith-Waterman or BLAST all stages can be implemented fully accurately in hardware, but since a pipeline runs only as fast as its slowest

stage, the decision of whether or not to do so is concerned more with economics than challenge and accuracy.

In this work, we compare two approaches to the acceleration of profile-HMM search – the first a modification of a basic design based on HMMer [21]; the second a new design we propose in order to recover the accuracy lost by the first approach. Our design and comparative results are also based on experiments with HMMer. As this work did not lead to the development of an actual hardware accelerator, our conclusions are drawn from software simulations of an FPGA-based accelerator.

Although our accelerator designs are not tied to a specific platform for their deployment, we assume the use of FPGA hardware for primarily economic reasons. For the given application, with its relatively small user base, the startup cost associated with manufacturing ASICs is prohibitive, favoring the use of reconfigurable hardware such as FPGAs instead. Further, the reconfigurability of FPGAs facilitates easy upgrades of the application.

The contributions of this work are as follows:

- The analysis of errors produced by a basic accelerator design using the Viterbi algorithm and the development of a performance model to evaluate any accelerator design (Section 2.2).
- The proposal of a modification to the basic design to recover lost accuracy (Section 3.1).
- The proposal of an entirely new design to more efficiently recover lost accuracy (Section 3.2).

- The development of a software simulator to measure the accuracy and running time as well as program trace statistics of any accelerator design (Section 3.3).
- The investigation of the Forward algorithm for use in the hardware accelerated stage of the accelerator (Sections 4.2 and 4.3).

The remainder of this work is organized as follows. Chapter 2 examines the profile-HMM search acceleration problem in detail (specifically using the Viterbi algorithm in Hit Detection) and develops a performance model to evaluate proposed designs. Chapter 3 details the architectures, and compares experimental evaluations, of two designs for implementations of the Viterbi algorithm. Chapter 4 examines specific issues involved in accelerating the Forward algorithm for use in the Hit Detection stage. Chapter 5 concludes and suggests the direction of future work.

Chapter 2

Accelerator Design

In this chapter we describe the Viterbi algorithm for profile-HMM search as a two-stage computation – the first stage being implemented in parallel hardware. We examine the algorithm in detail and identify data dependences in the dynamic programming recurrence that hinder parallelism and suggest a strategy for simplification of the algorithm to make it parallel at the cost of some accuracy. Finally, we develop a detailed performance model for the evaluation of alternative accelerator architectures based on the two-stage design.

2.1 Profile-HMM Search Acceleration Issues

In this section we describe a two-stage design for a profile-HMM search accelerator and identify the key challenges involved in implementing the first stage in parallel hardware. We then suggest a simple modification to the Plan7 HMM structure that makes the algorithm parallelizable and quantify the loss of accuracy due to such a modification.

2.1.1 A Two-Stage Design

The core of the Viterbi algorithm, namely the computation of the dynamic programming matrix, may be replicated in two stages – Hit Detection, which computes the matrix and determines whether the search yields a significant score with respect to a threshold; and Path Generation, which also computes the matrix and, additionally, recovers the best scoring state path and outputs an alignment. This acceleration strategy for profile-HMM search makes use of the fact that most proteins do not share features with most families of sequences in the database (and so the vast majority of searches of query sequences against HMMs in the database are expected to be low-scoring). It places Hit Detection in parallel hardware, while performing Path Generation in software on a general-purpose CPU only for those searches that are deemed significant. Hence, the core computation is performed in very fast hardware for all input searches and repeated in software only for a very small fraction. Such an approach prevents the low-scoring majority from wasting valuable general-purpose CPU cycles, thereby speeding up the overall computation.

The high level structure of the pipeline is shown in Figure 2.1. The input for each individual search consists of a query sequence and a profile-HMM representing a family of motifs. A user-supplied threshold determines whether or not the score of the search is considered significant enough to report as a hit. For a hit, the output contains the score and the location of each detected copy of the motif in the query.

As mentioned in Section 2.1, the first stage, Hit Detection, is isolated from the rest of the computation in order to filter low-scoring searches out of the input to the Path Generation stage. Further, since the Hit Detection stage may run a different

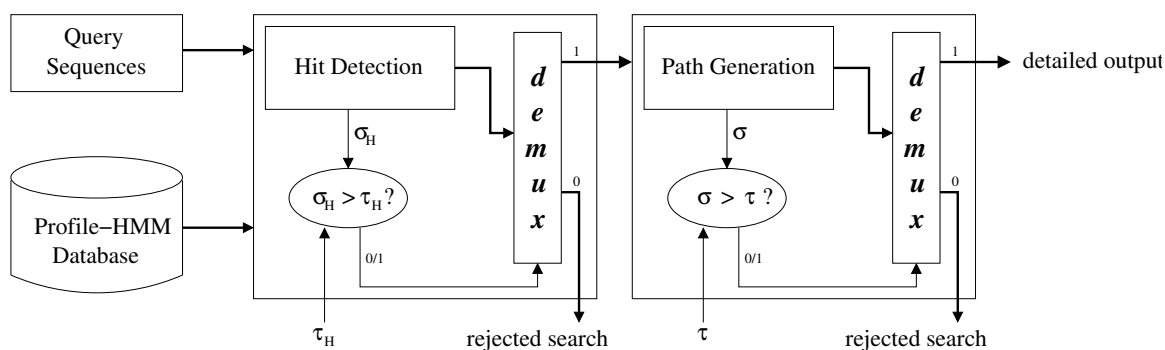


Figure 2.1: The profile-HMM search pipeline with a Hit Detection stage targeted for implementation in parallel hardware and a Path Generation stage running on a general-purpose processor. Potentially distinct thresholds τ_H and τ are used by each stage.

version of the Viterbi algorithm from the Path Generation stage, the pipeline could potentially determine two different internal scores σ_H and σ each compared to a different threshold, τ_H and τ .

In our experiments with the HMMer software package, we divided the standard implementation of the Viterbi algorithm into a Hit Detection stage and a Path Generation stage. Traces through the modified program for a search of 1200 protein sequences against the 7677 profile-HMMs of Pfam-A showed that more than 99% of CPU time was spent in the Hit Detection stage, clearly establishing it as the bottleneck and candidate for acceleration in parallel hardware.

2.1.2 Data Dependences in the Viterbi Algorithm

Figure 2.2 depicts the dynamic programming matrix computed by the Viterbi algorithm. The vertical axis represents motif position, while the horizontal axis represents amino acid position in the query sequence. Each cell of the matrix corresponds to a

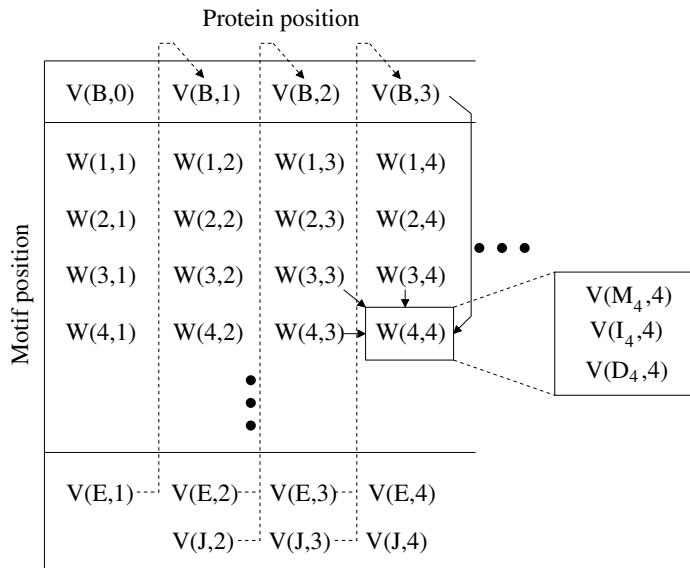


Figure 2.2: The dynamic programming matrix for the Viterbi algorithm for profile-HMM search. The $W(4,4)$ block has been magnified to show the 3 cells internally contained. Data dependences of the same block are shown with solid arrows. Dashed arrows indicate dependence on cells along the Plan7 feedback path.

Viterbi score, $V(q, j)$ (for a state path ending at state q and emitting the first j characters of the query sequence), computed using the recurrence in Equation 1.2. Further, for motif position i and query position j , the values of the three cells, $V(M_i, j)$, $V(I_i, j)$ and $V(D_i, j)$, are lumped together into the block $W(i, j)$. Figure 2.2 shows the internal composition of the $W(4,4)$ block.

As shown in Equation 1.2, the score of each term $V(q_i, j)$ of the recurrence is a maximization over state path extensions from all candidate *predecessor* states $q' \in Q_i$. Similarly, computing the cell corresponding to $V(q_i, j)$ maximizes over the scores of the predecessor cells corresponding to each $V(q', j - 1)$ plus transition and emission scores (if q_i is an emitting state) or $V(q', j)$ plus transition score (if q_i is non-emitting). It is also required that each cell in the matrix store a pointer to the maximizing predecessor cell from which the state path was extended into it. Then, tracing back

through the matrix from the final cell (i.e. $V(End, l)$) using these pointers yields the best scoring state path.

The set of permissible predecessor states, Q_i , for each state q_i , is evident from Figure 1.6. These translate into data dependences for computation of each cell on the values of predecessor cells. Figure 2.2 shows with solid arrows the dependence of the $W(4, 4)$ block on (the values of some cells within each of) the $W(3, 3)$, $W(3, 4)$ and $W(4, 3)$ blocks as well as the individual $V(B, 3)$ cell. Specifically, these dependences arise from the following permissible state transitions: $M_3, I_3, D_3 \rightarrow M_4$; $M_4, I_4 \rightarrow I_4$; $M_3, D_3 \rightarrow D_4$; $B \rightarrow M_4$.

2.1.3 Parallelizing Hit Detection

The dashed transitions of Figure 1.6 between the E and B states (possibly through the J state) are responsible for what we call a *feedback path* in the Plan7 HMM structure, because they make the E and J states valid predecessors of the B state. Thus, the score upon passing through the B state to begin a new instance of the motif is dependent on the scores after ending previous instances by passing through the E and possibly J states.

The dashed arrows flowing into each of the $V(B, j)$ cells in Figure 2.2 represent their dependence on the values of $V(E, j)$ and $V(J, j-1)$ – predecessor cells on the feedback path between multiple copies of a motif. For instance, $W(4, 4)$ depends on $V(B, 3)$ which in turn depends on $V(E, 3)$ and $V(J, 3)$. But these two cells are themselves dependent on $V(B, 2)$ and the entire column of blocks below it. Similarly, $V(B, 2)$ is dependent on the entire column to the left of it and so on. Thus, $W(4, 4)$ is

implicitly dependent on the entire first, second and third columns of the matrix. This forces a serial order for computing the cells of the Viterbi matrix thereby disallowing computation of several cells in parallel.

The simplest approach to overcoming this obstacle is to eliminate the feedback path altogether in the Hit Detection stage, in effect allowing only one copy of a motif to be detected in the query. In such a structure, the various $V(B, j)$ cells can be precomputed since the dashed arrows flowing into them in Figure 2.2 are now absent, implying that they have no dependences on other cells in the matrix. The implicit dependence of every $W(i, j)$ block on all columns of blocks to the left similarly vanishes such that only those dependences shown with solid arrows remain. Figure 2.3 shows how parallel hardware, such as a systolic array, can exploit this feedback-free structure to compute the scores of multiple blocks along an *anti-diagonal* of the matrix at once. For example, the first time step computes $W(1, 1)$ given the precomputed $V(B, 0)$; the second time step $W(1, 2)$ and $W(2, 1)$ (which only depend on $W(1, 1)$ and the precomputed $V(B, 0)$ and $V(B, 1)$); the third time step $W(1, 3)$, $W(2, 2)$ and $W(3, 1)$ and so on, till the j th time step computes the blocks $W(i, j - i + 1)$ for $1 \leq i \leq j$. The actual number of blocks that can be simultaneously computed is bounded by available hardware resources and may or may not equal the length j of the anti-diagonal for large j . The performance model of Section 2.2.5 considers this limitation in detail.

The natural disadvantage of an approach that eliminates the feedback path is that it fails to detect multiple copies of a motif in the query sequence. The reason it remains an effective Hit Detection strategy is that if a protein contains multiple copies of a motif, chances are quite high that at least one of those copies will be high-scoring. However, this is not always the case, and in query sequences where no single copy

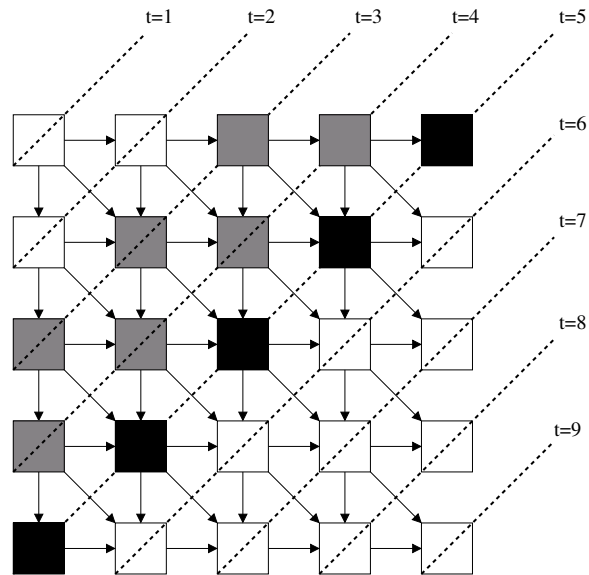


Figure 2.3: Computation of the feedback-free Viterbi matrix using a systolic array. Each square represents a block of dynamic programming cells as in Figure 2.2. Dotted anti-diagonal lines connect all the cells that are independent of each other and that may be computed in parallel in each time step. Arrows depict data dependences between blocks. In the 5th time step, black squares represent blocks being computed by the systolic array whereas gray squares represent stored values of predecessor blocks computed in the 3rd and 4th time steps.

Table 2.1: Loss of accuracy resulting from elimination of the feedback path from the Plan7 structure in Hit Detection.

E-value E	Hits detected when feedback path retained	Hits lost by eliminating feedback path	Estimated hits lost over entire Swiss-Prot database
10	77645	2612	72324
1	17367	562	15544
0.1	9628	246	6808
0.01	8130	208	5757
0.001	7577	173	4788

is sufficiently high-scoring even while all of them together are, such a Hit Detection stage will fail to correctly forward the search to the Path Generation stage.

Although the pipeline of Figure 2.1 uses thresholds τ_H and τ , which bear a direct relation to stringency, user-supplied thresholds are usually E-values, E_H and E , which bear an inverse relation to stringency, i.e. a lower E-value signifies a more stringent threshold. These are internally converted into thresholds, τ_H and τ using Karlin-Altschul statistical theory [12]. Table 2.1 shows for different E-value thresholds the number of searches (out of a comparison of 5898 randomly sampled proteins from Swiss-Prot against the 7677 HMMs built from Pfam-A) where a feedback-free Hit Detection stage failed to detect weak multicopy motifs in a query. Also shown is the projected number of missed motifs for a complete search of Swiss-Prot (1.6×10^5 sequences) against the same Pfam-A database. Although these searches constitute a small fraction of the total, their absolute number represents a sizeable loss of accuracy due to feedback-free Hit Detection. Chapter 3 discusses two approaches to recovering the lost accuracy due to these searches.

According to Figure 2.2, searching a query sequence of length l against a profile-HMM representing a motif of length m requires $O(ml)$ space. Since Hit Detection only computes the values of the cells of the matrix and does not need to recover a state path, it does not need to store the entire matrix. In fact, as revealed by the data dependences in the feedback path-free Hit Detection stage above, each anti-diagonal of blocks depends on at most only two previous anti-diagonals, meaning that no more than three anti-diagonals of the matrix need to be stored at any time. Hit Detection can, therefore, be performed in $O(m)$ space, thereby placing fewer demands on the hardware resources targeted for its deployment. The Path Generation stage, however, being a full implementation of the Viterbi algorithm, requires $O(ml)$ space as usual.

2.2 Performance Model

In order to evaluate the performance of a profile-HMM search accelerator design, it must be compared to a software-only unaccelerated implementation on two fronts: accuracy and execution time. Accuracy refers to the extent to which the two-stage accelerator output resembles that of unaccelerated software, irrespective of how biologically accurate the output of the unaccelerated software itself is. Execution time of the two-stage accelerator is reported as a speedup relative to the execution time of unaccelerated software running on a single general-purpose processor. Figure 2.4 depicts the relationship between the quantities in the performance model developed below and the two-stage accelerator design of Figure 2.1.

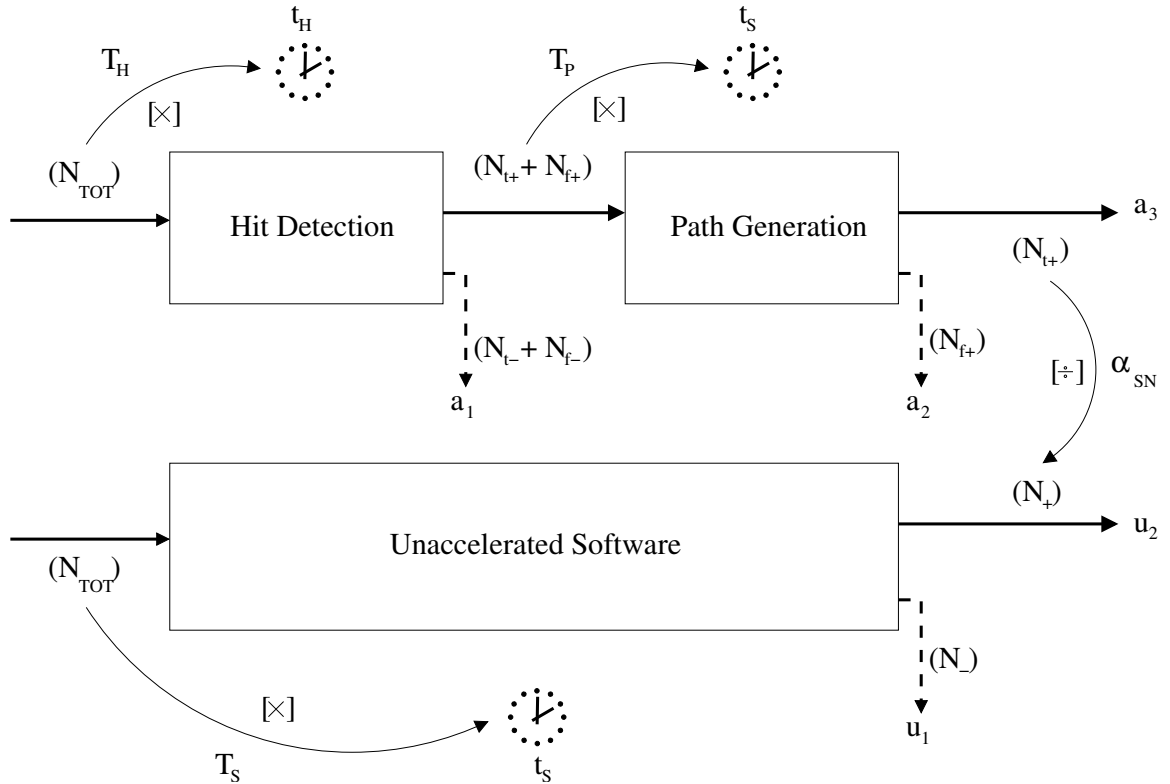


Figure 2.4: The relationship between various quantities introduced in the Performance model of Section 2.2. A straight edge represents a path through the system. Dashed edges represent paths that result in a negative output. The quantity in parentheses next to an edge indicates the number of searches passing through it. An arc represents derivation of the quantity next to it by performing the operation in square brackets on the operands at either end in the direction of the arrow.

2.2.1 Input Size and Parameters

The input to both the accelerator and unaccelerated software is the same, i.e. a set of query sequences and a database of profile-HMMs. We denote by N_S the number of query sequences, and by N_M the number of profile-HMM motifs in the database. The total number of searches, N_{TOT} , performed, is therefore

$$N_{TOT} = N_S \cdot N_M \quad (2.1)$$

The parameters to the search include the values of the different thresholds. Our results are reported for different E-value threshold combinations corresponding to the τ_H and τ thresholds (if distinct) for the accelerator, and the threshold corresponding to the same τ for unaccelerated search. Provided Hit Detection is implemented using a simplified version of the Viterbi algorithm or of the Plan7 structure (such as one eliminating the feedback path), each combination of thresholds results in different accuracy and speedup for the overall accelerator.

2.2.2 Accuracy

Since unaccelerated software is the reference, it produces no errors relative to itself. Also, since the Path Generation stage of the accelerator is identical to unaccelerated software in that it uses the same version of the Viterbi algorithm and Plan7 structure as well as the same threshold, it is not by itself responsible for any loss of accuracy in the pipeline. However, the Hit Detection stage, depending on how it simplifies the

algorithm and HMM structure, is prone to two kinds of errors: *false negatives* and *false positives*.

A false negative is produced when the simplified Hit Detection stage determines that the query sequence does not appear to contain any high-scoring copies of the motif in it, whereas unaccelerated software determines that it does in fact contain some. Since this search exits the accelerator through the edge marked a_1 in Figure 2.4, it does not make it to the Path Generation stage, which ordinarily could have detected all the copies of the motif in the query in the same way as unaccelerated software. We denote by N_{f-} the total number of searches that are false negatives.

A false positive is produced when the simplified Hit Detection stage determines that the query sequence appears to contain high-scoring copies of the motif in it, but unaccelerated software determines that it in fact contains none. Unlike false negatives, false positives are recoverable errors as they do not exit the accelerator before the Path Generation stage can correctly determine that they must exit the pipeline through edge a_2 . Therefore, the accelerator as a whole does not produce any false positives, although the Path Generation stage must waste CPU time in processing each false positive produced by Hit Detection. We denote by N_{f+} the total number of searches that are false positives.

Additionally, N_{t-} denotes the number of true negatives, i.e. searches correctly deemed low-scoring by Hit Detection; and N_{t+} denotes the number of true positives, i.e. searches correctly deemed high-scoring by Hit Detection and for which Path Generation produces detailed output via edge a_3 in Figure 2.4. Similarly, N_- and N_+

respectively denote the number of searches discarded via edge u_1 and reported in the detailed output via edge u_2 by unaccelerated software.

Since the number of searches entering must equal the number exiting, we must have

$$N_{TOT} = N_{t-} + N_{t+} + N_{f-} + N_{f+} = N_- + N_+ \quad (2.2)$$

We can now define our primary accuracy measure, the *sensitivity* of the overall pipeline, denoted by α_{SN} , as follows

$$\alpha_{SN} = \frac{N_{t+}}{N_+} \quad (2.3)$$

2.2.3 Execution Time Speedup

Let the average time taken to perform a full search in unaccelerated software be t_S (Section 2.2.4 shows how this quantity may be obtained). Then the total execution time, T_S , of unaccelerated software equals

$$T_S = N_S \cdot N_M \cdot t_S \quad (2.4)$$

Let the average time taken to perform Hit Detection be t_H (Section 2.2.5 shows how this quantity may be obtained). Then the total execution time, T_H , of the Hit Detection stage is

$$T_H = \frac{N_S \cdot N_M \cdot t_H}{n_{ST}} \quad (2.5)$$

where n_{ST} is the parallelism introduced by internal pipelining (if any) of Hit Detection. The value of n_{ST} is not necessarily equal to the number of stages but depends on how balanced the internal pipeline is. We separately specify n_{ST} for both the architectures detailed in Chapter 3.

Further, define f_{HIT} to be the fraction of total searches passed to Path Generation by Hit Detection. It can be seen from Figure 2.4 that

$$f_{HIT} = \frac{N_{t+} + N_{f+}}{N_{TOT}} \quad (2.6)$$

Since an individual search takes the same amount of time in Path Generation as in unaccelerated software, the total execution time, T_P , of the Path Generation stage must be

$$T_P = N_S \cdot N_M \cdot f_{HIT} \cdot t_S \quad (2.7)$$

Since the two stages are pipelined, the overall execution time of the accelerator, T_A , is the same as that of its slower stage

$$T_A = \max\{T_H, T_P\} \quad (2.8)$$

Finally, the speedup, X , achieved by the accelerator is given by

$$X = \frac{T_S}{T_A} \quad (2.9)$$

2.2.4 Software Timing Estimates

The time complexity of the Viterbi algorithm for searching a particular query s of length l and profile-HMM, μ with m Match states is $O(ml)$. Therefore, the running time can be approximated as a linear function $t(s, \mu) = \gamma ml + \delta$. We timed the Viterbi algorithm as implemented in the HMMer software package for a search of 1200 randomly sampled query sequences from Swiss-Prot against the 7677 profile-HMMs of Pfam-A on a Pentium 4 processor running Linux and found that such a linear approximation closely models the observed running time.

Therefore, the average execution time for a search in unaccelerated software, t_S , can be obtained as the value of the linear function when m equals the mean motif size over the database, and l equals the mean sequence length over the set of queries.

2.2.5 Hardware Timing Estimates

In a systolic array implementation, the available hardware resources are in the form of independent computational units that we call *cells*. For the purposes of our work, each computational cell of the systolic array computes the value of a dynamic programming cell. Our hardware timing estimates are based on FPGA designs.

As seen earlier, the size of the dynamic programming matrix is $O(ml)$. However, the exact number of cells, $C(s, \mu)$, that will need to be computed in order to compare s to μ depends on the simplified algorithm and HMM structure used for Hit Detection. We later derive this value for each of the two designs presented in Chapter 3.

The execution time, t_H , for Hit Detection is then given by

$$t_H = \frac{C}{R} \quad (2.10)$$

where C is the number of cell updates required for searching an average length sequence (over the set of queries) against an average size motif (over the profile-HMM database) and R is the cell update rate (measured in cell updates per unit time) realizable by the systolic array.

We now derive R as follows. Consider a hardware stage of total *area* A_{TOT} (area is a measure of the available computational resources). Let the functional units and logic required by each computational cell utilize a area units. Then the number of cells, A , that can be simultaneously computed is

$$A = \frac{\eta \cdot A_{TOT}}{a} \quad (2.11)$$

where η is the maximum fraction of the total area that may be utilized for computational cells alone to keep routing delays at a minimum. This value is typically in the range of 0.7 – 0.9. Further, the memory requirements of each computational cell make the value of A implicitly bounded by the available memory on board. Although we do not factor this restriction into the performance model, it is an important validity check for designs that appear to realize absurdly high parallelism.

Further, each cell requires a total of n_{CLK} clock cycles to compute its value. If the hardware is clocked at frequency f_{CLK} , then the overall cell update R is given by

$$R = \frac{A \cdot f_{CLK}}{n_{CLK}} \quad (2.12)$$

2.2.6 Banded Computation and Edge Effects

The anti-diagonal of Figure 2.3 is shown to span the entire width of the matrix. However, since the area A of the systolic array is often less than the length of the longest possible anti-diagonal of the dynamic programming matrix (which is $\min\{m, l\}$), the overall matrix ends up being computed as several bands, each A blocks wide, as shown in Figure 2.5A. The values of blocks on the boundary of each band must be stored so computation of the next band may proceed. Further, the reduced dependences of feedback-free Hit Detection allow the bands to be vertical or horizontal. This means that each band may either be a sub-matrix formed by a chunk of the motif A blocks wide against the entire length of the query (horizontal bands), or a sub-matrix formed by a chunk of the query A blocks wide against the entire motif (vertical bands).

In horizontal banding, the computation of each band requires loading a chunk of the motif (A positions wide) once into memory and streaming in the entire length of the sequence. Therefore, the entire motif is read once while the sequence is streamed several times (once for each horizontal band). On the other hand, vertical banding requires loading a chunk of the sequence (again, A positions wide) and streaming in the entire length of the motif. Therefore, the entire sequence is read once whereas the motif is streamed several times (once for each vertical band).

It must be noted that while each sequence position is simply a single character, the size in memory of a notional motif position is much larger – each position i of the motif consists of the three states M_i, I_i and D_i . The M_i state is characterized by its 4 integer transition scores to the M_{i+1}, I_i, D_{i+1} and E states and a distribution

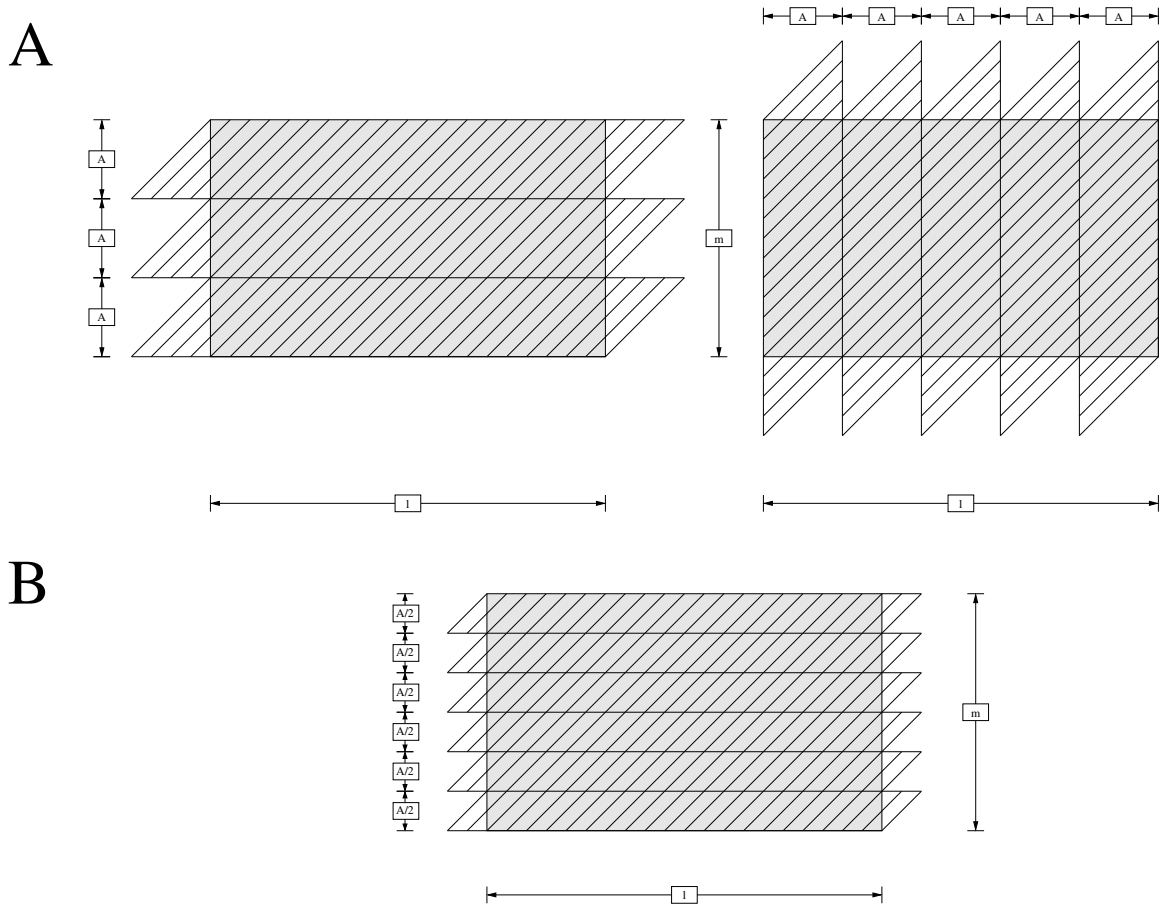


Figure 2.5: (A) Illustration of banded computation and a comparison of edge effects in horizontal and vertical banding. Each band is A blocks wide. Each outlying triangle represents wasted cell updates. The total number of wasted cell updates is proportional to mA for horizontal banding and to lA for vertical. Usually, $m < l$, so that horizontal banding wastes fewer cell updates. Additionally, horizontal banding generally requires fewer total memory accesses. (B) Effect of reducing A to proportionately reduce wasted cell updates due to edge effects. Here, reducing the number of concurrent cell updates by half also reduces the total number of wasted cell updates by half.

of emission scores for 20 amino-acids. Similarly, Figure 1.6 reveals that I_i contains 2 transition and 20 emission scores; and D_i contains 2 transition and no emission scores. Therefore, in total, the size in memory of a single motif position can be as high as that of 48 integers – far larger than 1 character for a single sequence position. Although HMMer uses 32 bit integers to represent all scores, our experiments reveal that using 16 bit integers gives the same results. Therefore, for 16 bit wide integers and 5 bit wide characters (sufficient to represent the protein alphabet) one position of a motif occupies approximately 154 times as much space in memory as a position of the sequence. This disparity is clearly in favor of horizontal banding (unless $l > 154m$ – an extremely rare case), as it requires reading in the motif only once into memory.

Figure 2.5A also illustrates the edge effects that significantly influence the actual cell update rate realized during the dynamic programming computation. Near the edges of the matrix, although the systolic array is capable of performing A cell updates simultaneously, the anti-diagonal requires fewer cells to be actually computed and so there are idle computational cells that cannot be utilized in any other way. The wasted cell updates are shown as triangles extending out of the matrix for each band. The total number of such wasted cell updates is proportional to $A \cdot m$ if the bands are horizontal or $A \cdot l$ if the bands are vertical. Since it is more common for m to be less than l , the wasted cell updates are fewer for a computation using horizontal bands. On account of this fact as well as the lower memory access cost, horizontal banding is taken to be the default choice in the remainder of our work.

Figure 2.5B shows how the number of wasted cell updates may be reduced by half through a reduction of the number of concurrent cell updates to $\frac{A}{2}$. This suggests that it is possible to minimize the edge effects without sacrificing original throughput

by realizing the possible concurrent cell updates A through n_P different searches running in parallel on the Hit Detection hardware¹, each performing $\frac{A}{n_P}$ concurrent cell updates. However, each search might then place its own additional disk I/O demands. A more quantitative analysis of the conflicting effects of wasted cell updates and the choice of n_P could help optimize hardware performance in the future.

The edge effects are easily accounted for in the performance model by adding the number of cells in the outlying triangles to the total cell updates, C , now making it a function of A and n_p as well. We estimate C for each of the designs detailed in Chapter 3 in this way.

¹This entails only a minor modification to the pipeline of Figure 2.1 where a suitably large buffer needs to be placed before the Path Generation stage.

Chapter 3

1-pass and 2-pass Architectures

In this chapter, we present two alternative profile-HMM accelerator architectures, based on the two-stage design, which use different approaches to eliminate the feedback path of the Plan7 HMM structure in their respective Hit Detection stages. For each architecture, we describe the strategy used by its Hit Detection stage as well as its effect on the key metrics of our performance model. Finally, we compare experimental evaluations of software simulations of these two architectures using the performance model and characterize cases where each is favorable over the other.

3.1 1-pass Architecture

In this section we introduce the notion of a 1-pass architecture based on the feedback-free Hit Detection stage suggested in Section 2.1.3 and describe a slight modification to the basic 1-pass architecture. We also derive the cell update count, C_1 , for this architecture to be substituted for C in our performance model.

3.1.1 Relaxation of Hit Detection Threshold

The simple approach to parallelizing Hit Detection introduced in Section 2.1.3 wherein the feedback path of the Plan7 HMM structure is altogether eliminated may be referred to as a 1-pass architecture as it detects only single copies of a motif in a query, i.e. makes one pass over the profile-HMM. We have seen such a Hit Detection stage to produce false negatives relative to an unaccelerated software implementation.

The easiest way to recover some of the false negatives is to make the Hit Detection threshold less stringent than that used by Path Generation (i.e. set $\tau_H < \tau$) in order to let searches with weak multicopy motifs through the Hit Detection filter. The same searches when run in the Path Generation stage with threshold τ will produce the same output as unaccelerated software.

Notice that in Table 2.1, the false negatives produced by a 1-pass Architecture without relaxation is a relatively small fraction of the total number of searches. Further, since the vast majority of searches are correctly classified as true negatives, it follows that true negatives must far outnumber false negatives. Therefore, for every false negative recovered through some relaxation of the Hit Detection threshold, it is expected that many more true negatives are also wrongly forwarded to the Path Generation stage, now making them false positives – a computational burden on Path Generation, though they do not affect the overall accuracy of the accelerator relative to unaccelerated software.

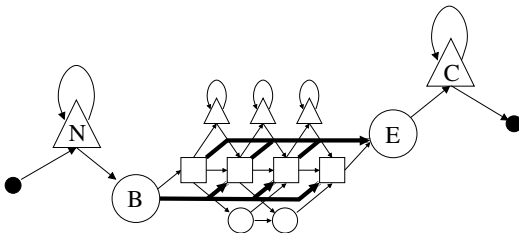
The 1-pass architecture with relaxation requires no modifications to the general accelerator pipeline of Figure 2.1 except the restriction that $\tau_H < \tau$. Figure 3.1A depicts

the modified Plan7 HMM with the feedback path absent, as employed by its Hit Detection stage. The value of n_{ST} for a 1-pass architecture is 1 as it is not internally pipelined. It is important to distinguish n_{ST} from n_P – while n_P is the number of parallel Hit Detection pipelines into which the systolic array is divided in order to reduce wasted cell updates due to edge effects, n_{ST} reflects the parallelism of each pipeline. Here, each of the n_P Hit Detection pipelines can perform only one search at once since the Hit Detection stage is undivided. Therefore, $n_{ST} = 1$.

3.1.2 Derivation of Cell Update Count

The bulk of the dynamic programming matrix of Figure 2.2 lies within the W -blocks. Each block contains 3 cells and there are ml such blocks. Since each block depends on a fixed number of blocks and cells, these updates take constant time. Their overall contribution, therefore, is $3ml$ cell updates. Since 1-pass architectures do away with the J state, the remaining cell updates involve the computation of the various B and E cells. Of these, the B cells are precomputed serially and are discounted here as they are not part of the parallel computation. As is evident from the direct transitions from each M_i state to the E state in Figure 1.6, each $V(E, j)$ cell (for $1 \leq j \leq l$) is dependent on m cells. Therefore, computing an E cell is not a constant time operation and so these account for ml cell updates, bringing the total to $4ml$ actual cell updates. Finally, in the more frequent case that $m < l$, edge effects account for an additional $4\frac{A}{n_P}m$ wasted cell updates (see Section 2.2.5). Therefore, we obtain C_1 , the number of cell updates made by the Hit Detection stage in a 1-pass architecture,

A



B

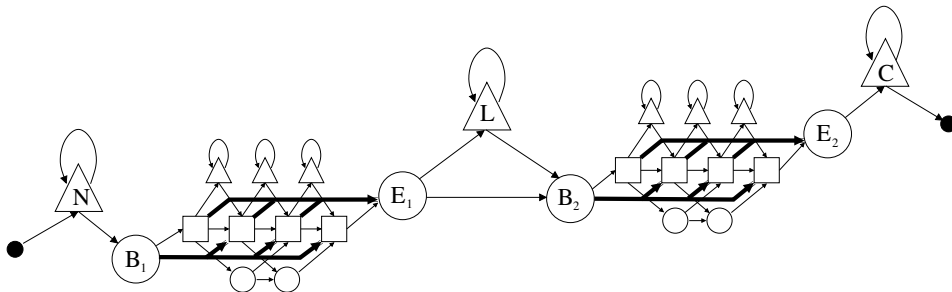


Figure 3.1: (A) The 1-pass HMM structure obtained by eliminating the feedback path from the Plan7 structure. (B) The 2-pass HMM structure obtained by creating two copies of the Plan7 motif separated by a linker state, L , to emit non-motif symbols between them.

to plug into Equation 2.10 as follows.

$$C_1 = 4 \left(l + \frac{A}{n_P} \right) m \quad (3.1)$$

3.2 2-pass Architecture

In this section we introduce the notion of a 2-pass architecture as a means to detect the existence of multiple copies of a motif in a query sequence and describe how the Hit Detection stage in this architecture may be internally pipelined. We also derive the cell update count, C_2 , for this architecture to be substituted for C in our performance model.

3.2.1 Unrolling the Plan7 Feedback Path

We have seen that the number of false negatives produced by a 1-pass architecture may be reduced by relaxing the Hit Detection threshold, τ_H . However, this strategy is of low specificity in that it introduces false positives. The 2-pass architecture is a more specific approach to identifying multi-copy motifs in a query without sacrificing parallelism in the Hit Detection computation. This is achieved by comparing two search scores – one where precisely one pass is made over the profile-HMM, and another where precisely two passes are made. If the two-pass score is greater than the one-pass score, it is likely that the query contains two or more copies of the motif, and the search is forwarded to the Path Generation stage *irrespective* of the one-pass score.

The HMM structure employed by a 2-pass architecture is shown in Figure 3.1B. It is obtained by unrolling the feedback path of the Plan7 structure precisely once, yielding two copies of the core states separated by a linker state, L , that emits non-motif symbols between the two instances of the motif. The direct transition from E_1 to B_2 allows the second copy of the motif to begin immediately after the first, without passing through the L state. The goal of the Hit Detection stage here is to compute 1-pass as well as 2-pass scores. Just as the value $V(E, l)$ gives the 1-pass score computed by the 1-pass architecture, the value $V(E_2, l)$ gives the 2-pass score computed by the 2-pass architecture. Further, the value $V(E_1, l)$ computed by the 2-pass architecture is identical to the value $V(E, l)$ computed by the 1-pass architecture. Therefore, the computation of the 1-pass score by the 2-pass architecture is contained within that of the 2-pass score and need not be separately implemented.

Another observation that follows from the 2-pass HMM structure is that the computation of the vector of values $V(E_1, i)$ for $1 \leq i \leq l$ given the vector $V(B_1, j)$ for $0 \leq j \leq l - 1$ is identical to the computation of the vector $V(E_2, i)$ given the vector $V(B_2, j)$ for $1 \leq i, j \leq l$. Therefore, the Hit Detection computation can be performed in three phases. Phase 1 receives the precomputed vector $V(B_1, j)$ for $0 \leq j \leq l - 1$ and computes the vector $V(E_1, i)$ for $1 \leq i \leq l$. Phase 2 receives as input the vector output by Phase 1 and computes the vector $V(B_2, i)$ for $1 \leq i \leq l$. Finally, Phase 3 receives as input the vector output by Phase 2 and computes the vector $V(E_2, i)$ for $1 \leq i \leq l$. As noted, Phases 1 and 3 are identical and may be implemented using two copies of the same hardware computational resource.

Phase 2 performs the bridging computation between Phases 1 and 3 using the following recurrence.

$$V(B_2, j) = \max \begin{cases} \max_{j' < j} V(E_1, j') + a(E_1, L) + \sum_{k=j'}^j b(L, s_k) + a(L, B_2) \\ V(E_1, j) + a(E_1, B_2) \end{cases} \quad (3.2)$$

The upper term maximizes over all state paths in which non-motif characters are emitted by the linker state between the two copies of the motif. Each of these candidate paths passes through state E_1 after emitting some $j' < j$ characters of the query sequence, and then makes a transition to and remains in the L state to emit characters at sequence positions $(j' + 1)$ through j , before finally making a transition into B_2 . The lower term, on the other hand, considers the one state path that does not pass through the L state at all but instead directly makes a transition from E_1 to B without emitting any characters from the sequence.

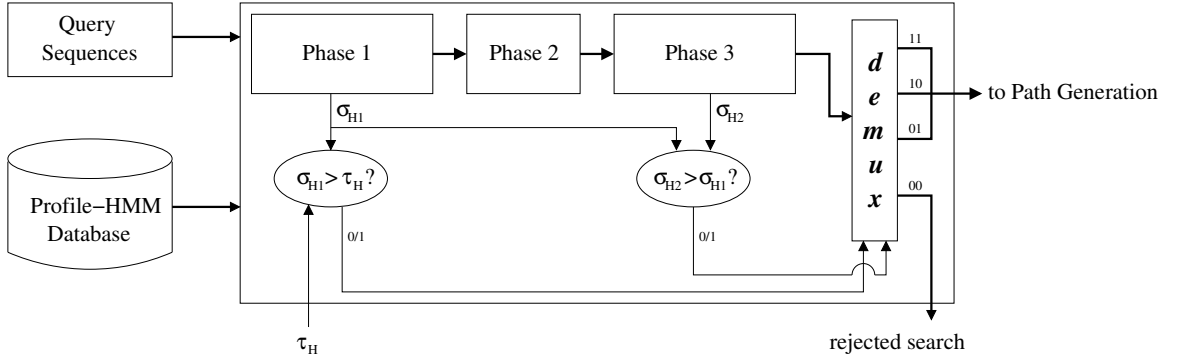


Figure 3.2: The three pipelined phases that make up the Hit Detection stage in a 2-pass architecture. The 2-pass heuristic forwards all searches which see any improvement from σ_{H1} to σ_{H2} to Path Generation, irrespective of whether σ_{H1} exceeds Hit Detection threshold, τ_H .

Since the computation of each phase only depends on the output of the phase preceding it, these three phases can be pipelined, thereby achieving the same throughput as a 1-pass Hit Detection stage. It must be noted that though there are 3 phases in the Hit Detection pipeline, Phase 2 is much smaller than the other two, so that the pipeline may effectively be considered as composed of two balanced stages – balanced because Phases 1 and 3 are identical. Therefore, each of the n_P Hit Detection pipelines that the systolic array is divided into runs two different searches at any given time. This leads to $n_{ST} = 2$ being substituted into Equation 2.5 for the 2-pass architecture. Figure 3.2 depicts the three-phase design of the Hit Detection stage in a 2-pass architecture. The one-pass and two-pass scores are referred to as σ_{H1} and σ_{H2} respectively.

3.2.2 Derivation of Cell Update Count

The computations of Phases 1 and 3 of 2-pass Hit Detection are identical to that of 1-pass Hit Detection. Therefore, each contributes $4(l + \frac{A}{n_P})m$ cell updates, including

wasted cell updates due to edge effects. Since Phase 2 is not highly parallel, we ignore edge effects, and consider only the $2l$ cell updates required to compute the $V(L, i)$ and $V(B_2, i)$ cells for l values of i . Therefore, we obtain C_2 , the total number of cell updates made by the Hit Detection stage in a 2-pass Architecture, to plug into Equation 2.10 as follows.

$$C_2 = 8 \left(l + \frac{A}{n_P} \right) m + 2l \quad (3.3)$$

3.3 Experimental Results

In this section we describe the methodology used to evaluate and compare the two accelerator architectures. We present results for accuracy and speedup relative to unaccelerated software. We also compare the results of each architecture to identify cases where each wins out over the other.

3.3.1 Methodology

We developed a software simulator based on the code of HMMer version 2.3.2 to implement the 1-pass and 2-pass architectures as well as the unaccelerated software implementation of the Viterbi algorithm. The simulator was compiled using the GNU C Compiler version 3.4.4 and executed on a 2.8GHz Pentium 4 CPU with 1GB RAM running Linux. The two accelerator architectures were then compared in terms of their accuracy and running time speedup relative to unaccelerated software using the performance model and timing estimates of Section 2.2. Cell update counts in Hit Detection for the two architectures are as developed in Sections 3.1.2 and 3.2.2 above.

Simulator data was gathered for a comparison of 5898 protein sequences randomly sampled from Swiss-Prot against the 7677 profile-HMMs of Pfam-A for several combinations of E-value thresholds, E_H and E , and extrapolated over the entire Swiss-Prot database (1.6×10^5 sequences) using the observed mean sequence length over Swiss-Prot and mean motif size over Pfam-A. We cross-validated our results by running the simulator on four different inputs, each containing an independent random sample of proteins (of the same size – 5898 sequences) from Swiss-Prot, and extrapolating the results for each input over the entire Swiss-Prot database.

Since we ran our evaluations on a software simulator rather than on a real accelerator with an FPGA-based Hit Detection stage, our speedup results are obtained from estimates of various values in the performance model based on published specifications of the Xilinx Virtex-4 XC4VLX100 FPGA [26].

From the specifications, we get $A_{TOT} = 49152$ slices and 4320KB Block RAM (240 blocks of 18KB each). We assume a conservative $\eta = 0.7$ for the fraction of area that may actually be used for computation. Each cell of the systolic array computes one block of the dynamic programming matrix which accounts for 4 cell updates (one each for the M, I, D and E states). Each cell is built from 11 16-bit integer adders (8 slices each), 6 16-bit integer comparators (16 slices each) and 25 16-bit registers (0.5 slices each), of which 8 are used for prefetching motif data. This approximately gives us $a = 200$ slices per cell of the systolic array or per 4 dynamic programming cells. Therefore, we get $A = 688$ concurrent cell updates according to Equation 2.11. For these unpipelined integer computations, $n_{CLK} = 1$ and the systolic array can be clocked at $f_{CLK} = 50\text{MHz}$. Finally, therefore, our Hit detection stage achieves a cell update rate of $R = 34.4 \times 10^9$ cell updates per second (CUPS) or 34.4 GCUPS.

It must be noted that the above assumptions have no effect on the accuracy of the accelerator – with respect to either sensitivity or false positive rate. Therefore, the accuracy results presented in the following section are independent of the assumed hardware area and speed.

3.3.2 Comparative Accuracy

We assume that the goal of an accelerator is to not only speed up profile-HMM search but also to do so without any overall loss of accuracy with respect to unaccelerated software. Therefore, for both accelerator architectures, and for each value of threshold, E , the least stringent (i.e. highest valued) E_H for which Hit Detection produces no false negatives was noted. For this critical value of E_H , the time spent in Path Generation provides a measure of the number of false positives produced by the accelerator.

Table 3.1 compares the critical values of E_H for the two architectures, and shows the effect of its lenience on false positive rate. It is clearly seen that at stricter (numerically lower) values of E , the 2-pass architecture achieves perfect sensitivity without having to relax the Hit Detection threshold as much as the 1-pass architecture. Although their respective critical values of E_H differ by up to three orders of magnitude (for $E = 0.001$), the more meaningful comparison is between the projected times in Path Generation, for this describes the amount of additional work performed by the Path Generation stage. The 2-pass architecture is seen in this regard to produce an order of magnitude fewer false positives, though the benefit is less pronounced at less stringent values of E .

Table 3.1: Comparison of critical values of threshold E_H at which 1-pass and 2-pass architectures achieve perfect sensitivity. The accompanying projected time spent in Path Generation is a measure of the number of false positives produced by each architecture.

E	One-pass		Two-pass	
	E_H for Sensitivity 1.0	Time in Path Generation (hours)	E_H for Sensitivity 1.0	Time in Path Generation (hours)
0.001	5	2.208	0.003	0.626
0.01	20	5.498	0.02	0.684
0.1	20	5.498	2	1.508
1	30	7.524	9	3.250
10	40	9.479	40	9.490

The values in Table 3.1 were obtained by extrapolating the results of four different inputs (of equal size) to the simulator over the entire Swiss-Prot database. The critical values of E_H were observed to be identical for all inputs. The Path Generation times shown are averaged over the four sets of results. The observed variation in Path Generation times across the four sets was $< 5\%$.

It must be noted that the two architectures produce false positives of different kinds. As is evident from Figure 2.1, the only false positives produced by the 1-pass architecture exit the Path Generation stage demux through the edge marked 0. These are insignificant hits that pass through the Hit Detection filter due to the relaxation of τ_H to allow the detection of weak multicopy motifs. However, Figure 3.2 reveals two distinct kinds of false positives. One of these is the same as that produced by the 1-pass architecture, and will exit the Hit Detection demux through the edge marked 10 if $\tau_H > \tau$. The other kind of false positives is introduced by the two-pass heuristic and includes searches with low one-pass scores but that see an absolute score improvement from σ_{H1} to σ_{H2} and are forwarded to Path Generation where they are found not to contain any weak multicopy motifs. Such searches, if any, exit the Hit

Detection demux through the edge marked 01. We found that the 2-pass architecture shows greater susceptibility to the latter kind, especially at less stringent thresholds – on average by a single order of magnitude. While the relative composition of 2-pass false positives is only of academic interest in that it does not affect performance under the model of Section 2.2 in any way, it may provide a basis for subsequent refinements of the two-pass heuristic.

3.3.3 Comparative Speedup

The projected running time of the Path Generation stage in each accelerator architecture is shown in Table 3.1. However, according to Equation 2.8, the overall running time of the accelerator pipeline equals that of its slower stage, or bottleneck. When the Hit Detection stage is the bottleneck, we say that the accelerator is *hardware-bound* and, similarly, that it is *software-bound* when the Path Generation stage is its bottleneck.

Figure 3.3 compares the speedup, X , achieved by the two architectures using the hardware parameters estimated in Section 3.3.1. In addition we also present the speedup realizable by use of a hypothetical superior hardware stage (assuming double the clock rate f_{CLK} and triple the total area A_{TOT}). The figure also quantifies the amelioration of edge effects by running multiple parallel searches in Hit Detection. As mentioned in Section 2.2.6, increasing n_P places additional demands on the external hard disk, from which the profile-HMMs and sequences must be read into memory, to an extent not accounted for by our performance model. Since disk access is serial,

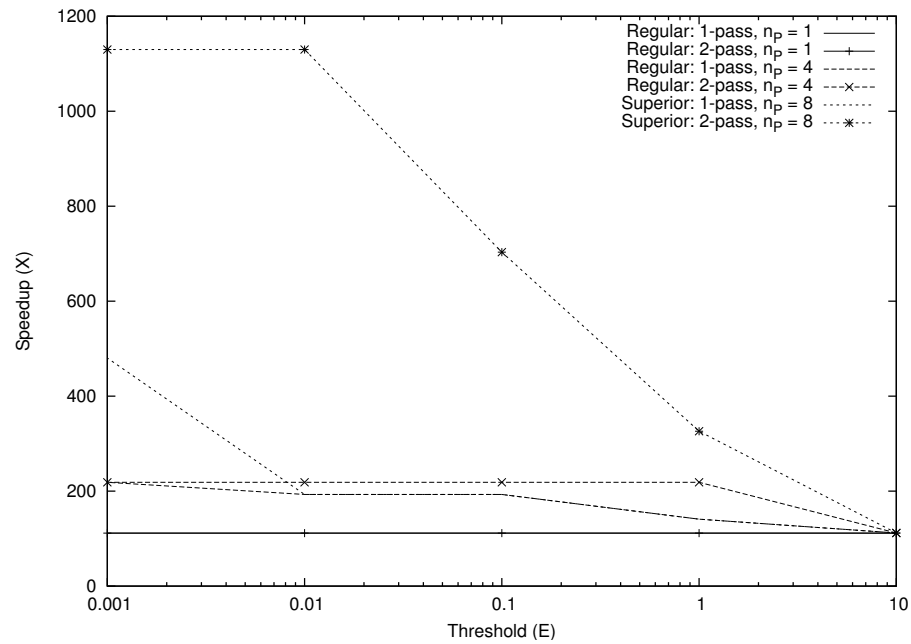


Figure 3.3: A comparison of the speedup achievable by 1-pass and 2-pass architectures. The curves labeled Regular assume the hardware timing parameters of Section 3.3.1. The curves labeled Superior assume a hypothetical six-fold hardware speed increase. The plot must be read from right to left in order to consider increasing stringency of threshold E .

we do not consider any higher values of n_P than 8 as the associated overhead may no longer be justifiably negligible.

Although Figure 3.3 shows threshold E increasing numerically from left to right, the plot must be read from right to left to move from less stringent to more stringent thresholds. It is easily seen that for $n_P = 1$, the 1-pass and 2-pass architectures are equivalent, both achieving a speedup of about 112 relative to unaccelerated software. Moreover, the flat shape of the two curves indicates that the accelerators are hardware bound in this regime – since T_H is independent of E whereas T_S generally is not, and a constant speedup indicates a constant execution-time bottleneck. Increasing n_P to 4 leads to a near twofold improvement in speedup for stricter values of E . The speedup of the 2-pass architecture is larger than that of the 1-pass architecture for most of this regime because the 1-pass architecture is software bound on account of its higher false positive rate (see Table 3.1) whereas the 2-pass architecture is hardware bound. At $E = 0.001$, however, they converge to a speedup of 218.

In the hypothetical case that we have available a superior hardware stage with thrice as much area and capable of running at twice the clock rate, the benefits of the 2-pass architecture are even more pronounced, with projected speedups as high as 1130 relative to unaccelerated software while the 1-pass architecture follows the current curve before surging up to a speedup of 487 at $E = 0.001$.

3.3.4 An Alternative Route to Estimating Speedup

Since the speedups of Figure 3.3 are the output of a theoretical performance model given input parameters obtained by guided assumptions about the state of FPGA

Table 3.2: Published performance of systolic array Smith-Waterman implementations (using both linear and affine gap penalties) on Xilinx FPGAs.

Group	FPGA (A_{TOT} , in slices)	A (updates/clock)	f_{CLK} (MHz)	R (GCUPS)
VanCourt and Herbordt [23]	XC2VP30 (13,696)	57 to 126	39 to 77	2.2 to 9.7
Oliver et al. [16, 17]	XC2V6000 (33,792)	119 to 252	44 to 55	5.2 to 13.9
West et al. [24]	XCV1000E (12,288)	152	25	3.8

technology, they must be treated as optimistic upper bounds. The actual speedups realized by real FPGA implementations of our architectures may in fact turn out significantly smaller due to routing and memory access issues. In the absence of published speed and area numbers for FPGA implementations of the Viterbi algorithm, we turned to the literature on systolic array implementations of the Smith-Waterman algorithm (using both linear and affine gap penalties). As earlier noted, the external structures of the Smith-Waterman and Viterbi recurrences (Equations 1.1 and 1.2) are sufficiently similar that the numbers in Table 3.2 may be considered a starting point for a more conservative estimate for Viterbi speedup using the 1-pass and 2-pass architectures. Specifically, cell updates may be equivalently defined for both algorithms so that the values of A presented for Smith-Waterman can easily derive estimates of the same for Viterbi.

On the basis of the values in Table 3.2, and adjusting for evolution of FPGA technology, we made direct estimates of A (rather than detailed estimates of the area a of each cell of the systolic array) and different values of n_P , with a constant $f_{CLK} = 100\text{MHz}$. Figure 3.4 shows the speedups of the 1-pass and 2-pass architectures for $A = 25, n_P = 1$; $A = 300, n_P = 6$; and $A = 600, n_P = 12$ – a wide range of relatively conservative estimates with respect to those in Figure 3.3. The shape of the curves reveals the same qualitative relationship between 1-pass and 2-pass speedups,

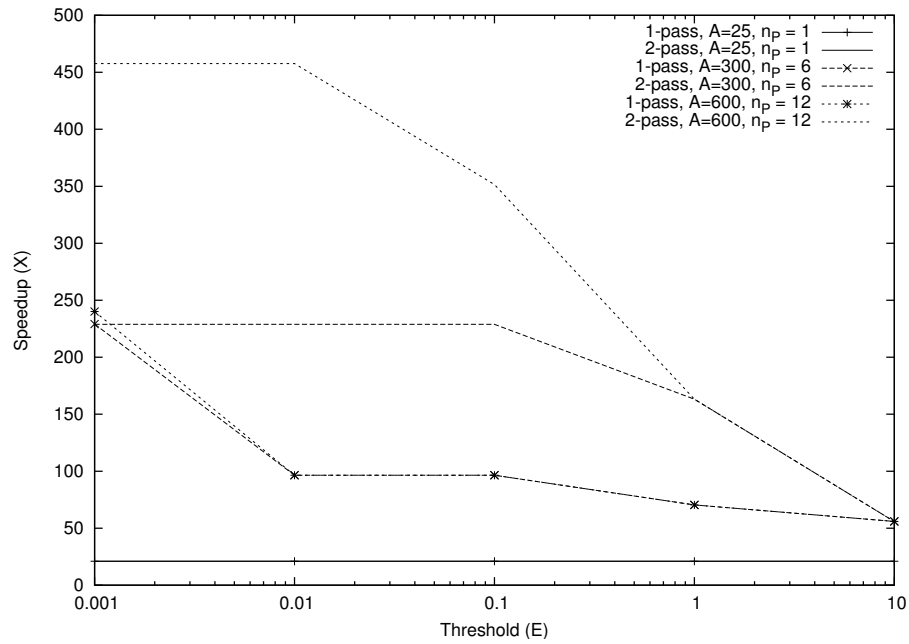


Figure 3.4: An alternative comparison of the speedup achievable by 1-pass and 2-pass architectures using estimates for A made on the basis of the published Smith-Waterman performance numbers of Table 3.2. A constant value $f_{CLK} = 100\text{MHz}$ is assumed for all curves in this figure. The plot must be read from right to left in order to consider increasing stringency of threshold E .

but shows numerically smaller speedups. Importantly, the figure expresses just as well the specific behavior of the two architectures over different regimes in which they are hardware bound and software bound, demonstrating that the performance model is a very useful tool for analyzing any accelerator architecture, irrespective of the actual state of FPGA technology.

Chapter 4

Accelerating the Forward Algorithm for Hit Detection

In this chapter we describe and evaluate an alternative approach to profile-HMM search that combines the Viterbi and Forward algorithms. We address the motivation for such an approach and outline the design issues specific to accelerating the Forward algorithm. Finally, we evaluate a 1-pass architecture based on this approach using the performance model developed in Section 2.2

4.1 Motivation

We introduced the Forward algorithm in Section 1.3 as a method to score the sum of probabilities of all state paths over a profile-HMM that emit the given query sequence. For this reason, it is sometimes argued that the Forward algorithm is more sensitive to weak matches than the Viterbi algorithm. Specifically, if two or more state paths over a profile-HMM are more or less equally likely to emit a query sequence, the

Viterbi algorithm compares only one of their scores to its threshold. In the case that the score of each individual state path falls below the threshold while their combined sum crosses it, the Viterbi score causes the search to be deemed insignificant whereas the Forward score causes it to be deemed significant.

The Forward algorithm, on account of summing over the probability of all state paths, does not keep track of any one, and so cannot be used to retrieve a final alignment, leaving that instead to the Viterbi algorithm. However, for simply answering the question of whether or not a query sequence exhibits similarity to a profile-HMM, the Forward algorithm may be applied just as well. This makes it a candidate for deployment in the Hit Detection stage, even as the Path Generation stage runs the Viterbi algorithm as before. We applied the 1-pass architecture of Section 3.1 to a two-stage accelerator design where Hit Detection is performed using the Forward algorithm over the HMM structure of Figure 3.1A. We then simulated this accelerator as we did the Viterbi accelerator and evaluated it using the same performance model.

A key point to note is that since they are compared to intrinsically different scores, the Viterbi and Forward thresholds are not meant to be the same. However, since HMMer 2.3.2 applies the same threshold to both scores, our simulated accelerator does the same. This is because for any given search the Forward score is numerically higher than the Viterbi score. Therefore, using the Viterbi threshold guarantees that no additional false negatives are produced, though it may introduce additional false positives that are detected in Path Generation.

4.2 Design Issues

In this section we explore the probabilistic roots of the Viterbi and Forward algorithm and the use of logarithms for computational convenience. We describe specific difficulties involved in computing logarithms of sums for the Forward algorithm and approaches to overcoming them.

4.2.1 Recurrences in Probabilistic Form

The Viterbi recurrence of Equation 1.2 is a maximization over integer terms. However, these integer terms are actually derived from the probabilistic form of the Viterbi recurrence. In order to compute the probability, $\Pr(q_i, j)$, that a state path is in state q_i after emitting the first j symbols of query sequence s , we consider the probabilities associated with extending the state path from each valid predecessor state, $q' \in Q_i$ to q_i . If q_i is an emitting state, this is the product of the probability $\Pr(q', j - 1)$ of being in the predecessor state after emitting $(j - 1)$ symbols, the probability $\Pr(q_i|q')$ of making a transition to q_i , and the probability $\Pr(s_j|q_i)$ of emitting the j th symbol from q_i . Similarly, if q_i is non-emitting, it is the product of $\Pr(q', j)$ and $\Pr(q_i|q')$. This leads to the probabilistic form of the Viterbi recurrence below.

$$\Pr(q_i, j) = \max \begin{cases} \max_{q' \in Q_i} \Pr(q', j - 1) \cdot \Pr(q_i|q') \cdot \Pr(s_j|q_i), & \text{if } q_i \text{ is an emitting state} \\ \max_{q' \in Q_i} \Pr(q', j) \cdot \Pr(q_i|q'), & \text{otherwise} \end{cases} \quad (4.1)$$

Similarly, since the Forward algorithm sums the same probabilities rather than maximizing over them, we get the probabilistic form of the Forward algorithm. Note that since there is no *choice* being made by the algorithm in a dynamic programming sense, the Forward algorithm, unlike the Viterbi algorithm, is not a combinatorial optimization algorithm.

$$\Pr(q_i, j) = \begin{cases} \sum_{q' \in Q_i} \Pr(q', j-1) \cdot \Pr(q_i|q') \cdot \Pr(s_j|q_i), & \text{if } q_i \text{ is an emitting state} \\ \sum_{q' \in Q_i} \Pr(q', j) \cdot \Pr(q_i|q'), & \text{otherwise} \end{cases} \quad (4.2)$$

4.2.2 Computing with Logarithms of Probabilities

The various transition and emission probabilities are real numbers between 0 and 1. In the case of the Viterbi algorithm, several such probabilities are multiplied in the course of obtaining the final score, leading to the danger of floating-point underflow errors. This is commonly averted by storing and computing logarithms of probabilities rather than raw probabilities themselves, thereby converting the various multiplications in Equation 4.1 into sums of logarithms. Not only does this make the computation invulnerable to underflows but also significantly faster – since logarithms are implemented as fixed-point quantities, they are computationally equivalent to integers, and therefore the core of the computation is equivalent to integer addition, which is much less expensive than floating-point multiplication. The Viterbi algorithm, as a result, is frequently performed using *log-odds ratios*, which normalize

logarithms of probabilities with respect to a background distribution, as doing so provides a better idea of how much more likely a particular state path is than a random one (see [5] for a more detailed discussion). In fact, the integer Viterbi scores as well as the transition and emission scores in Chapters 2 and 3 are log-odds ratios.

In the case of the Forward algorithm, however, using logarithms no longer provides the same convenience as it does for the Viterbi algorithm. This is because while multiplication translates to addition in the log domain, addition does not translate to any simple operation. Mathematically, suppose $c = \log(x)$ and $d = \log(y)$. Then,

$$\log(xy) = c + d$$

but

$$\log(x + y) = \log(e^c + e^d)$$

This necessitates the floating-point exponentiation operation for the Forward algorithm. One common workaround, employed by HMMer, uses the result of the following algebra.

$$\begin{aligned} \log(x + y) &= \log(e^c + e^d) \\ &= \log(e^c(1 + e^{d-c})) \\ &= \log(e^c) + \log(1 + e^{d-c}) \\ &= c + \log(1 + e^{d-c}) \end{aligned}$$

or equivalently,

$$\log(x + y) = d + \log(1 + e^{c-d})$$

Therefore, the log of the sum is the addition of either c or d with another term dependent only on the difference between them. In essence, HMMer precomputes and stores a lookup table of $\log(1 + e^h)$ indexed by $h = |c - d|$ over a part of the range of c and d ¹. The core of the Forward algorithm is then simply reduced to table lookups and additions. While this is slower than the Viterbi computation in log space, it is still an acceptable solution for computing the Forward score in log space in software. However, performing the same algorithm in parallel hardware requires a sufficiently high number copies of the lookup table in highly multiplexed on-board memory so that all cells of the systolic array can look up their own copy of the table with minimum delay caused by sharing. As noted in Section 2.2.5, the availability of on-board memory constrains the value of A . Our designs for the Viterbi accelerator store the HMM transition and emission scores required by each cell of the systolic array in memory. Clearly, additionally duplicating a table with 20000 integers in memory forces A to be very low.

4.2.3 Forward Algorithm with Floating-Point Probabilities

Since performing the Forward computation in log space is impractical in parallel hardware, we investigated the option of using floating-point representations of the probabilities (or odds ratios). In order to quantify the threat posed by underflow we performed a floating-point Forward algorithm search of 1250 randomly sampled proteins from the Swiss-Prot database against the 7677 profile-HMMs built from Pfam-A, and counted the occurrence of negative infinity, infinity and NaN (not a number) scores – the possible resulting scores when a floating point exception occurs

¹For large $|c - d|$ the larger term dominates the sum and a table lookup is unnecessary.

during the Forward computation. We received floating-point exceptions in nearly 21% of the cases when single precision was used, while only in 0.24% of the cases when double precision was used.

One way to deal with floating-point exceptions is to treat the search as a hit and forward it to Path Generation, thereby potentially producing even more false positives than before. Even though it then appears that using single precision may produce more than 20 times as many false positives, the ultimate effect on overall speedup depends only on whether the extra burden on the Path Generation stage causes it to become the bottleneck. Another key point is that our underflow estimates are based on searches made using the original Plan7 HMM structure, i.e. with feedback path intact. Since the actual Hit Detection stage which performs these floating-point computations uses a feedback-free HMM structure, there is usually a smaller scope for underflow as a single pass over the motif means fewer multiplications than multiple passes. In the rest of our analysis, we discount the floating point exceptions and their contribution to the false positive rate, instead reintroducing them when considering the speedups estimated by our simulator in Section 4.3 below.

Since the Forward-based Hit Detection stage is based on the 1-pass architecture of Section 3.1 and also uses the same HMM structure, the dynamic programming matrix is identical to that in Figure 2.2 and the cell update count is the same as that in Equation 3.1. In fact, the only difference introduced is with respect to the hardware timing parameters (such as those estimated in Section 3.3.1) that are substituted in the performance model. As in Section 3.3.1, our estimates assume a Hit Detection stage implemented in FPGA hardware. As before, we assume $A_{TOT} = 49152$ slices, 4320KB Block RAM and $\eta = 0.7$. Our area and clock rate estimates of floating

point adders and multipliers implemented in FPGA hardware are based on results presented in [8].

Each cell of the systolic array once again computes a block of the dynamic programming matrix that accounts for 4 cell updates (corresponding to the M, I, D , and E states) per time step. Each cell is built from 1 pipelined floating point multiplier and 1 pipelined floating point adder and 25 floating point registers. In the case of single precision (32-bit), the multiplier has 4 stages and the adder has 6. Including registers, the approximate area of each cell is then 550 slices, and it is clocked at $f_{CLK} = 100\text{MHz}$, with the number of clock cycles for the pipelined computation being $n_{CLK} = 33$. This leads to $A = 250$ concurrent cell updates and $R = 0.76$ GCUPS.

In the case of double precision (64-bit), the multiplier and adder both have 6 stages each. Including registers, the approximate area of each cell is 1200 slices, and it is clocked at 50MHz, with the number of clock cycles for the pipelined computation being $n_{CLK} = 39$. This leads to $A = 115$ concurrent cell updates and $R = 0.147$ GCUPS.

4.3 Experimental Results

In this section we present the accuracy and speedup realized by the 1-pass Forward accelerator with current FPGA technology as well as based on projections for the future. We also suggest changes to the way the Forward algorithm is currently implemented that may provide scope for further improvement.

As in the case of the Viterbi accelerator, we used a software simulator with a 1-pass Forward-based Hit Detection stage to search 5898 randomly sampled proteins from Swiss-Prot against the 7677 profile-HMMs built from Pfam-A and extrapolated our results over the entire Swiss-Prot database (1.6×10^5 sequences) using the observed mean sequence length over Swiss-Prot and mean motif size over Pfam-A.

Table 4.1 shows the critical values of E_H for each value of threshold E at which the accelerator achieves perfect sensitivity (in the absence of floating-point exceptions) relative to unaccelerated software. As was similarly noted in Section 3.3.1, the hardware timing estimates have no effect on the accuracy results of Table 4.1. Since these critical values of E_H are in most cases smaller relaxations of E than those for the 1-pass architecture in Table 3.1, it does appear that the Forward algorithm is more sensitive for the purpose of Hit Detection when the same threshold is used. However, the accompanying Path Generation times are significantly higher than those in Table 3.1, signifying the rather low specificity of the Forward algorithm when the same threshold is applied. We suggest that future work on theoretically determining an appropriate distinct threshold for the Forward score may alleviate the problem, though it may simultaneously affect the critical value of E_H .

Finally, Figure 4.1 shows the speedup achieved by single and double precision Forward-based accelerators relative to unaccelerated software using hardware parameters estimated for current FPGA technology as well as for some hypothetical superior hardware stage as in Section 3.3.3 with thrice the total area and running at twice the clock rate. As before, we consider $n_P = 1, n_P = 4$ and $n_P = 8$.

Table 4.1: Critical values of threshold E_H at which the 1-pass architecture for a Forward algorithm accelerator achieves perfect sensitivity (in the absence of floating-point exceptions). The accompanying projected time spent in Path Generation is a measure of the number of false positives produced.

E	E_H for Sensitivity 1.0	Time in Path Generation (hours)
0.001	2	3.571
0.01	2	3.571
0.1	2	3.571
1	5	6.578
10	30	30.969

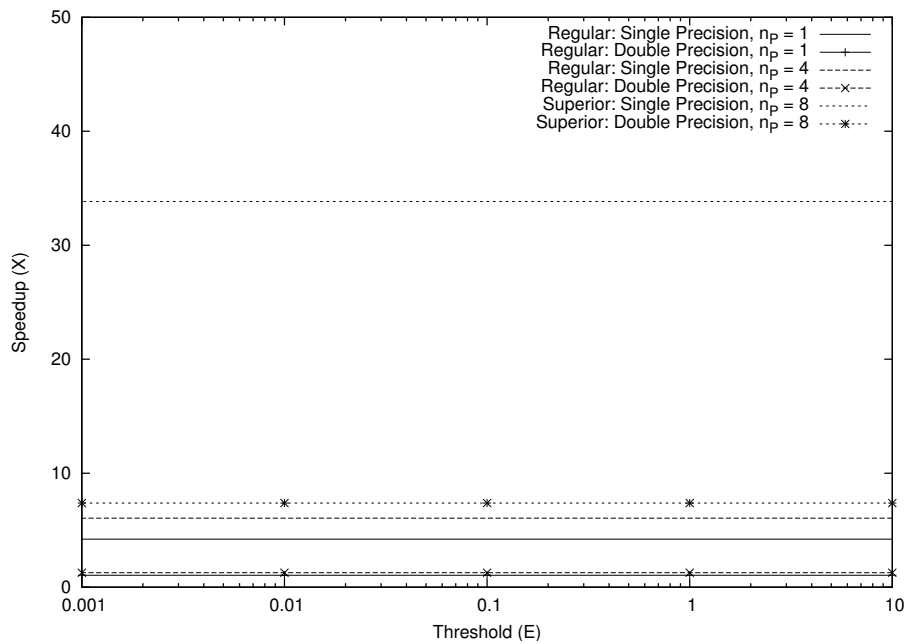


Figure 4.1: A comparison of the speedup achievable by single and double precision Forward-based accelerators. The curves labeled Regular assume the hardware timing parameters of Section 4.2.3. The curves labeled Superior assume a hypothetical six-fold hardware speed increase. The plot must be read from right to left in order to consider increasing stringency of threshold E .

At this point, we reintroduced the effect of floating-point exceptions. Our simulator results revealed that even after multiplying the single precision accelerator's Path Generation time by 22 and that of the double precision accelerator by 1.24 (to account for the maximum possible additional false positives), they remain clearly hardware bound and hence achieve the same speedup as before, justifying the omission of these effects from our analysis.

The flat shape of all curves in Figure 4.1 reveals that an accelerator using a floating-point Forward-based Hit Detection stage is hardware bound and likely to remain so in the future. This especially argues against a 2-pass architecture for floating-point Forward-based Hit Detection as its only possible contribution could be to reduce Path Generation time, which is of little use as long as Hit Detection is the clear bottleneck stage. The speedups achievable by both single and double precision are much smaller than by the 1-pass and 2-pass Viterbi architectures. Only single precision can potentially realize double-digit speedup given the same hypothetical superior hardware stage (with thrice as much total area and capable of twice the clock rate), and so, we suggest that future work in the direction of single precision optimizations specific to the Forward computation may help altogether eliminate its tendency to underflow (thereby removing the need to consider double precision) and achieve greater speedup.

Chapter 5

Conclusion

Our work reveals the great potential offered by acceleration of profile-HMM search in protein sequences with the aid of parallel hardware. Although existing accelerator architectures can realize significant speedup, the simplicity of their designs sacrifices accuracy for speed. Our 1-pass architecture, although introducing only the conceptually minor modification of Hit Detection threshold relaxation, achieves perfect sensitivity relative to unaccelerated software while achieving high speedups. The 2-pass architecture, an entirely new idea contributed by this work, achieves perfect sensitivity with greater efficiency than the 1-pass architecture and, therefore, allows greater overall speedups. The advantage of this architecture may become more pronounced should specialized hardware evolve so that Hit Detection stage is no longer the bottleneck and the time spent in Path Generation becomes key to the performance of the accelerator.

Through alternative hardware timing parameter estimates, we show that the two architectures we described can achieve speedups of between 2 and 3 orders of magnitude. Given the variety of considerations that go into estimating hardware timing and the frequently large gap between design and implementation, our performance

model may be used to obtain speedup estimates for a more diverse range of hardware area and speed assumptions, thereby allowing end users to evaluate each accelerator architecture according to parameters best suited to the resources available to them. We also suggest that investigation into the current limitations of the performance model and work towards overcoming them will provide more accurate estimates in the future.

We suggest that future work towards modeling in greater detail the effects of memory access patterns and the availability of highly multiported memory on board the hardware stage as well as routing and miscellaneous delays on the value of A may help provide more accurate estimates of speedups using our performance model. We also suggest a more thorough investigation of disk access issues governing the value of n_P . Eventually, as we have demonstrated clearly the benefits of the 2-pass architecture, its translation into an actual implementation will produce a significantly faster profile-HMM search accelerator (than currently available implementations similar to our 1-pass architecture) and will be useful to the biosequence analysis community.

Our investigation of acceleration of the Forward algorithm is only preliminary in that it establishes an advantage in terms of accuracy that when used in Hit Detection and favors the use of floating-point arithmetic over table lookup-based integer arithmetic. However, there needs to be a theoretically, or otherwise at least empirically, founded method for conversion between Viterbi and Forward score thresholds. Further work in the direction of eliminating single precision underflow can pave the way for more Forward-based accelerator designs. Since no other protein motif finding algorithm performs floating-point arithmetic, we were unable to provide an alternative set of

speedup estimates of the kind we provided for the Viterbi accelerator based on published Smith-Waterman accelerator performance numbers.

Even though we targeted the Forward algorithm for acceleration in parallel hardware, our findings suggest that it may be worthwhile for standard profile-HMM search software packages to consider the Forward score as a filter. In also initiating the discussion of its acceleration, we believe work on software and hardware implementations may progress alongside one another rather than hardware catching up to software, thereby benefiting both areas of research.

References

- [1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–10, October 1990.
- [2] Alex Bateman, Lachlan Coin, Richard Durbin, Robert D. Finn, Volker Hollich, Sam Griffiths-Jones, Ajay Khanna, Mhairi Marshall, Simon Moxon, Erik L. L. Sonnhammer, David J. Studholme, Corin Yeats, and Sean R. Eddy. The Pfam protein families database. *Nucleic Acids Research*, 32:D138–41, 2004.
- [3] 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.
- [4] G. M. Clore, J. G. Omichinski, and A. M. Gronenborn. Solution structure of the specific DNA complex of the zinc containing DNA binding domain of the Erythroid Transcription Factor Gata-1 by Multidimensional NMR, 1993. <http://www.rcsb.org/pdb/>.
- [5] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, New York, 1998.
- [6] Sean Eddy. HMMER: Sequence analysis using profile hidden Markov models, 2004. <http://hmmer.wustl.edu>.
- [7] European Bioinformatics Institute. TrEMBL protein database statistics, 2006. http://www.ebi.ac.uk/swissprot/sptr_stats/index.html.
- [8] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *Proc. 11th Reconfigurable Architectures Workshop*, New Mexico, USA, April 2004.
- [9] D. T. Hoang. Searching genetic databases on Splash 2. In *Proc. of IEEE Workshop on Field-Programmable Custom Computing Machines*, pages 185–192, 1993.
- [10] Daniel Reiter Horn, Mike Houston, and Pat Hanrahan. ClawHMMER: a streaming HMMer-search implementation. In *Proc. IEEE Supercomputing 2005*, Seattle, WA, 2005.

- [11] R. Hughey and A. Krogh. Hidden Markov models for sequence analysis: extension and analysis of the basic method. *CABIOS*, 12:95–107, 1996.
- [12] Samuel Karlin and Stephen 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, March 1990.
- [13] Praveen Krishnamurthy, Jeremy Buhler, Roger Chamberlain, Mark Franklin, Kwame Gyang, and Joseph Lancaster. Biosequence similarity search on the mercury system. In *Proc. 15th Int'l Conf. Application-Specific Systems, Architectures and Processors*, pages 365–75. IEEE, September 2004.
- [14] National Center for Biological Information. Growth of GenBank, 2005. <http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html>.
- [15] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–53, March 1970.
- [16] Tim Oliver and Bertil Schmidt. High performance biosequence database scanning on reconfigurable platforms. In *Proc. of 4th IEEE Int'l Workshop on High Performance Computational Biology*, April 2004.
- [17] Tim Oliver, Bertil Schmidt, and Douglas 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, February 2005.
- [18] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77:257–86, 1989.
- [19] Research Collaboratory for Structural Bioinformatics. Protein Data Bank entry 1gat. <http://www.rcsb.org/pdb/>.
- [20] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–97, March 1981.
- [21] Timelogic DeCypherHMM solution, 2004. http://www.timelogic.com/decypher_hmm.htm.
- [22] UniprotKB/Swiss-Prot. UniProtKB/Swiss-Prot entry P17678 (GATA1_CHICK) erythroid transcription factor. <http://tw.expasy.org/uniprot/P17678>.
- [23] Tom VanCourt and Martin 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, September 2004.

- [24] Benjamin West, Roger D. Chamberlain, Ronald S. Indeck, and Qiong Zhang. An FPGA-based search engine for unstructured database. In *Proc. of 2nd Workshop on Application Specific Processors*, pages 25–32, December 2003.
- [25] Ben Wun, Jeremy Buhler, and Patrick Crowley. Exploiting coarse-grained parallelism 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.
- [26] Xilinx Corporation. Virtex-4 multi-platform FPGA, 2006. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm.

Vita

Rahul Pratap Maddimsetty

- Date of Birth** January 6, 1983
- Place of Birth** Vishakhapatnam, India
- Degrees** B.Tech. Chemical Engineering, July 2004, from Indian Institute of Technology Madras
- Publications** R. Maddimsetty, J. Buhler, R. Chamberlain, M. Franklin, and B. Harris. “Accelerator Design for Protein Sequence HMM Search.” *to appear in Proceedings of the 20th ACM International Conference on Supercomputing (ICS06)*, Cairns, Australia, June 2006.

May 2006

Short Title: Acceleration of Profile-HMM Search Maddimsetty, M.S. 2006