BMC Bioinformatics

RESEARCH Open Access

An FPGA-based hardware accelerator supporting sensitive sequence homology filtering with profile hidden Markov models

Tim Anderson[1] and Travis J. Wheeler[2*]

*Correspondence: twheeler@arizona.edu

[1] Department of Computer Science, University of Montana, Missoula, MT, USA
[2] R. Ken Coit College of Pharmacy, University of Arizona, Tucson, AZ, USA

Abstract

Background: Sequence alignment lies at the heart of genome sequence annotation. While the BLAST suite of alignment tools has long held an important role in alignment-based sequence database search, greater sensitivity is achieved through the use of profile hidden Markov models (pHMMs). Here, we describe an FPGA hardware accelerator, called HAVAC, that targets a key bottleneck step (SSV) in the analysis pipeline of the popular pHMM alignment tool, HMMER.

Results: The HAVAC kernel calculates the SSV matrix at 1739 GCUPS on a ~ $3000 Xilinx Alveo U50 FPGA accelerator card, ~ 227× faster than the optimized SSV implementation in *nhmmer*. Accounting for PCI-e data transfer data processing, HAVAC is 65× faster than nhmmer's SSV with one thread and 35× faster than nhmmer with four threads, and uses ~ 31% the energy of a traditional high end Intel CPU.

Conclusions: HAVAC demonstrates the potential offered by FPGA hardware accelerators to produce dramatic speed gains in sequence annotation and related bioinformatics applications. Because these computations are performed on a co-processor, the host CPU remains free to simultaneously compute other aspects of the analysis pipeline.

Keywords: FPGA, Sequence alignment, Profile hidden Markov model

Introduction

After assembling the genome of an organism, it is standard practice to annotate the contents of that genome by comparing it to a library of known sequences. When the organism is evolutionarily distant from other sequenced genomes, as is common in the context of environmental metagenomic samples, high quality annotation depends on maximizing sensitivity in that comparative analysis. To date, high sensitivity in sequence comparison is achieved through sequence alignment, in which the letters of two sequences are arranged to identify regions of similarity. In the context of sequence alignment, models of mutational probability are used to compute a measure of the significance of the resulting alignment. Here, we focus on alignment methods for sequences that are highly

© The Author(s) 2024. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/. The Creative Commons Public Domain Dedication waiver (http://creativecommons.org/publicdomain/zero/1.0/) applies to the data made available in this article, unless otherwise stated in a credit line to the data.

divergent; see Sahlin et al. [1] for a review of methods for rapidly matching nearly identical sequences (as in the context of read mapping [2–4]).

Sequence alignment has been the target of intense design advances for algorithmic and statistical inference methods over several decades, resulting in sensitive and remarkably fast approaches for sequence annotation. For many years, the dominant tool in the space of high-volume sequence alignment was BLAST [5], with support from carefully-designed scoring models like position specific scoring matrices [6, 7]. In the decades since the introduction of BLAST, advances in sensitivity have come primarily in the form of position specific scoring matrices [8–10] and eventually profile hidden Markov models (pHMMs) [11]. Thanks to robust strategies for training and scoring [12, 13] and their representation of position-specific probabilities of observing letters, insertions, and deletions, pHMMs have remained the state-of-art for sensitive sequence annotation [14, 15].

Orthogonal to this development of sensitive models has been an ever-present push for greater speed, motivated by the exponential growth of modern sequence databases. These speed gains are generally achieved by either (i) filtering candidate alignment data with less computationally expensive algorithms, or (ii) devising faster implementations of the basic algorithms. The most popular alignment-based annotation tools achieve their speed by the first strategy, avoiding data analysis through various fast methods for predicting whether a sequence has the potential to produce a high score when exposed to a relatively expensive alignment algorithm [5, 16–20]. These approaches typically depend on indexing either the target sequences, query sequences, or both, and using the resulting indices to identify promising "seeds" for more intensive processing.

Any work-avoidance strategy runs the risk of lost sensitivity due to avoiding candidates containing true positive matches. In sequence alignment, index-based seed finding methods address these sensitivity/speed trade-offs through careful selection of data structures and parameterization. While recent advances retain BLAST-like sensitivity with 30–100$\times$ speed gains [21], the sensitivity of full-featured pHMMs is still unrivaled.

The alternative acceleration strategy (*apply the same core algorithm, but faster*) typically depends on some form of hardware acceleration. One such strategy leverages the SIMD (Single instruction, multiple data) vector instructions available on all modern CPUs [22]. SIMD sequence alignment implementations [23–26] have achieved impressive speed gains. This technique serves as a core part of the acceleration strategy used in popular profile alignment tools [21, 27, 28].

Hardware acceleration for sequence alignment has also been developed in the context of specialized hardware such as Graphics Processing Units (GPUs [29]) and field programmable gate arrays (FPGAs [30]). Here, we introduce an FPGA-base hardware accelerator that speeds up the key bottleneck stage of the HMMER pipeline by as much as 60x. Before presenting detailed methods and results, we first provide a brief introduction to the relevant aspects of HMMER, followed by a light introduction to hardware acceleration.

Profile HMMs and the HMMER pipeline

A profile HMM is a generative model of a family of sequences, with model parameters learned from multiple members of that family. Figure 1A shows the Plan-7 architecture

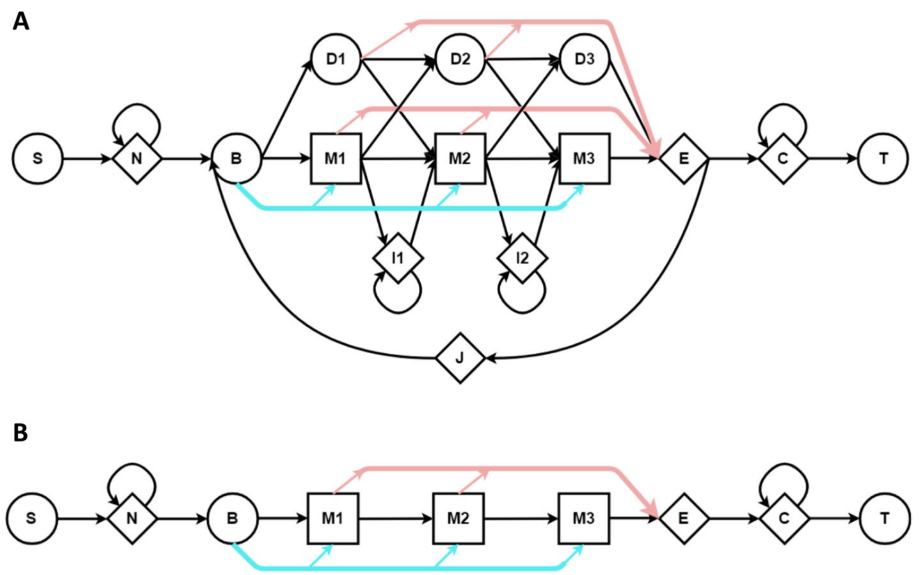

Fig. 1 The standard and SSV state models for HMMER3 pHMMs. In **A**, the core model utilized by HMMER3 Viterbi and Forward algorithm consists of states for observed positions in the modeled family (M), states for insertions relative to those positions (I), and silent states corresponding to the loss or deletion of those positions (D). The Jump state (J) enables a match between the target genome sequence and two disconnected (or even repeated) regions of the aligned model. The other states (S,N,B,E,C, and T) and path-skipping edges (blue/red) are required for proper scoring statistics [31]. **B** shows the reduced model utilized by the SSV algorithm implemented in HMMER3 and HAVAC, in which I, D, and J states are removed; this reduces data dependencies between cells of the Dynamic Programming matrix, and corresponds to alignments between query and target that contain only consecutive aligned positions

used in HMMER3 [31]. The architecture includes a core model made up of match states (which emit letters with position-specific probability at conserved positions in the family), insert states (which emit letters inserted between conserved positions), delete states (which silently bypass conserved positions), and position-specific transitions between these states, along with a few additional states and associated transitions that model letters not related to the core family—see [32] for further detail. The high sensitivity of pHMMs is due to (i) these position-specific probabilities, and (ii) application of the Forward algorithm [33] for computing support for the relationship between query HMM and target sequence.

Computation of a sequence alignment amounts to discovering a path through a QxT 2-dimensional matrix, where Q and T are the lengths of the query model and target sequence respectively. In the context of pHMMs, the Viterbi algorithm identifies a most-probable path through that matrix, and computes support for the relationship between Q and T from the single corresponding alignment. This is functionally equivalent [34] to the Smith-Waterman algorithm [35] that is approximated by BLAST and other faster tools mentioned above. Meanwhile, the Forward algorithm computes the sum of the probabilities of all possible paths (all alignments), and uses this as the basis for measuring support for relatedness. The run time complexity of both algorithms is $\Theta(QT)$, but Forward is much slower than Viterbi due to increased constant factors [36].

HMMER3 [36] produced a >100× speedup over the prior release, despite utilizing the relatively slow Forward algorithm, thanks to development of a pipeline consisting of

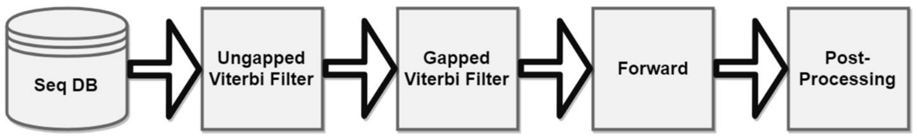

Fig. 2 HMMER Pipeline. The major stages of the HMMER pipeline. The Ungapped Viterbi, Gapped Viterbi, and Forward stages of the pipeline function as filters, reducing the number of queries that are passed onto subsequent stages

Table 1 The major stages of the HMMER3 pipeline, the percentage of random sequences that will be processed by a given pipeline stage, and the approximate percentage of total runtime required by each stage in a typical search

Pipeline stage	Ungapped Viterbi (%)	Gapped Viterbi (%)	Forward filter (%)	Align & post-process (%)
% random Ssequences processed	100	2	0.1	0.001
Total % Runtime	70	3	20	7

faster pHMM alignment filters that approximate the Forward score (Fig. 2). The key idea is that the Forward algorithm is only applied to a small number of candidate sequence matches that are allowed to pass these earlier, faster filters. Specifically: query/target pairs are aligned using a SIMD vectorized implementation [23, 25, 37] of the Viterbi algorithm, using reduced precision 16-bit integers in place of floating-point scores; this Viterbi stage approximates the score that will be achieved when running Forward, and is by default parameterized such that $\sim 1/1000$ random sequences are expected to pass the filter. This Viterbi filter is, in turn, preceded by an even simpler filter that compares the query to the target using a scoring scheme with further reduced precision (8-bit integers) and a variant of the model that does not allow for gaps (insertions or deletions). In the context of nucleotide alignment, the HMMER search tool nhmmer [15] calls this stage the Single Segment ungapped Viterbi (SSV) algorithm (see Fig. 1B). SSV produces a very rough approximation of the score that will be produced when computing the full Forward alignment, and is parameterized such that ~1/50 random sequences are expected to pass the filter. The SSV and Viterbi approximations are not particularly accurate, but are close enough to be generally useful; empirical evidence [36] suggests that nearly all matches that are reported by an unfiltered Forward implementation will also survive the Viterbi filter at $p \leq 0.001$ and the SSV filter at $p \leq 0.02$ [36].

Algorithm 1 describes the basic SSV algorithm, which computes the maximum ungapped score for matching a sequence to a query pHMM. HMMER3 parallelizes its SSV implementation using 16-way striped SIMD vector instructions [37]. The score returned from this algorithm can be compared against a threshold score (by default, the score required to produce a P-value of 0.02) to determine whether the given sequence is a sufficiently good match to warrant more robust calculation. Table 1 shows the average percentage of random sequences that will be processed by each stage in the pipeline and the total runtime spent in each stage in a typical use case. Even though SSV is the fastest stage in the pipeline, it still accounts for the majority of the runtime because every sequence must be evaluated by SSV, while only the best matching 2% of sequences are evaluated by Viterbi, and so on.

Algorithm 1 The Single Segment Ungapped Viterbi Algorithm

```
procedure SSV(model_emissions, sequence)
    $T \leftarrow$ |sequence|                                  ▷ Target
    $Q \leftarrow$ |model_emissions|                           ▷ Query
    $M \leftarrow$ 2D array of size $(T, Q)$
    $highest\_score \leftarrow 0$
    for $i \leftarrow 0$ to $T$ do
        $M[i, 0] \leftarrow 0$                       ▷ Initialize M[i, 0] to zero
    end for
    for $j \leftarrow 0$ to $Q$ do
        $M[0, j] \leftarrow 0$                       ▷ Initialize M[0, j] to zero
    end for
    for $i \leftarrow 1$ to $T$ do
        $symbol \leftarrow$ sequence[i]
        for $j \leftarrow 1$ to $Q$ do
            $score \leftarrow$ model_emissions[j, symbol]
            $M[i, j] \leftarrow$ M[i-1, j-1] $+ score$
            if $M[i, j] > highest\_score$ then
                $highest\_score \leftarrow M[i, j]$
            end if
            if $M[i, j] < 0$ then
                $M[i, j] \leftarrow 0$
            end if
        end for
    end forreturn $highest\_score$
end procedure
```

Hardware acceleration

An alternative approach to improving the runtime performance of the sequence homology search algorithms has been to employ hardware acceleration. ClawHMMER [38] was the first work to accelerate the pHMM Viterbi algorithm using Graphics Processing Units (GPUs). CUDAMPF [39] later used GPUs to accelerate ungapped Viterbi, gapped Viterbi, and Forward/Backward in HMMER. Other works have shown that Field Programmable Gate Arrays (FPGAs) can be used to further improve the pararallel performance of dynamic programming-based sequence alignment algorithms, including numerous FPGA hardware accelerators for the Smith-Waterman algorithm [40–43]. FPGAs are hardware devices that can be configured by Hardware Description Language (HDL) code to implement arbitrary digital logic circuits. Unlilke CPUs and GPUs, FPGAs have no prestructured computational architecture and can be optimized for a specific computational task. For example, an FPGA can be used to implement a chain of Processing Elements (PEs) that feed their outputs directly into the inputs of the next PE in the chain. These structures are called pipelines. Systolic arrays are a special type of pipeline with regular structure that synchronize data passing through the PEs in a controlled manner, similarly to their biological namesake. Systolic arrays tightly couple the control logic to the data being processed in order to compute small sections of a larger computational process on a regular basis. Both pipelines and systolic arrays can be used to efficiently pass data along in a highly parallel computational environment.

In serial implementations of sequence alignment, as in Smith-Waterman or in Viterbi and Forward with the standard pHMM model (Fig. 1A), it is common to perform calculations in row-major order, so that all calculations on one row are performed before moving on to the next row. Straightforward parallel implementations in which a batch of cells on one row are computed concurrently suffer from a data dependency pattern in which each cell depends on the previous cell in the row as well; multiple cells cannot be computed concurrently without either speculative calculation techniques as in [44] or parallelizing computation along the dynamic programming matrix's anti-diagonal [45].

An additional dependency challenge is presented by the existence of the pHMM J state, which may increase search sensitivity by allowing multiple passes through the core model. The inclusion of the J state results in a data dependency between each cell in the DP matrix and every cell corresponding to the previous sequence character, and as a result the maximum score from a given row must be identified before any cells in the subsequent row can be computed. Maddimsetty et al. [46] implement a two-pass hit detection algorithm by appending a duplicate copy of the model to the end of the model in lieu of the J state. This effectively allows two separate homologous areas to accumulate score as if there is a single-use J state loop. Oliver et al. [47] implements Viterbi on the full HMMER model including the J state, opting to parallelize over multiple discrete sequence/model pairs instead of along a single DP matrix anti-diagonal.

Abbas et al. [48] implemented a single accelerator that runs both a Multiple-Segment Ungapped Viterbi (MSV) filter and Viterbi on the same hardware. The MSV filter is similar to SSV, but it retains the HMMER pHMM J state. Abbas et al. implement the MSV filter with a max-reduction tree across a static number of cells from previous model states, allowing the accelerator to approximate the J state in a reasonable amount of parallelizable work, at the cost of a potential drop in sensitivity. Nowak et al. [49] developed an FPGA implementation of profile-based SSV modeled on the HHBlits [50] implementation. Their FPGA was limited to profile blocks of at most 220 positions, provided 128-way parallelism, and achieved a 1.91× speed improvement compared to the HHBlits function.

Here, we introduce a new hardware accelerator that implements SSV as a standalone nucleotide sequence homology filter, demonstrating vastly greater speed than similar previous accelerators. The Hardware Accelerated single-segment Viterbi Additional Coprocessor (HAVAC), is designed to exist as a standalone nucleotide sequence homology filter that can in principle be incorporated into a pHMM alignment pipeline in which SSV filter is run on the FPGA accelerator while downstream Viterbi and Forward alignment algorithms can be run simultaneously on the host CPU. In the standard implementation, SSV returns a maximum score that is later compared against a threshold score to determine if there was a matching region in the overall sequence. As alternative, HAVAC utilizes the parallel nature of reconfigurable hardware to check every cell's score against a threshold, and generate hit reports that detail the sequence and model positions of any hits. This extra information can be useful for downstream applications to determine areas of likely homology. The relative computational simplicity of the SSV model allows for better parallel performance than its more intricate counterparts. We also present a method for scaling a profile HMM's emission scores to leverage an 8-bit full adder's carry bit to check if a model's state score passes a significance threshold without requiring an 8-bit comparator. HAVAC returns the (i, j) pairs of DP matrix cells that pass the given threshold to allow later stages in a sequence homology search pipeline to localize search around areas of likely homology. HAVAC supports fasta-formatted sequence files and HMMER3-formatted model files. HAVAC is open-source licensed under BSD-3 and is available at https://github.com/TravisWheelerLab/HAVAC.

Methods

The HAVAC hardware accelerator was implemented on a Xilinx Alveo U50 FPGA accelerator card. The Alveo U50 is a PCI-e card that is simple to install and is modestly priced ($2,965 at time of writing). The Alveo U50 contains a custom-built Ultrascale+ FPGA with approximately 874K lookup tables (LUTs) and supports two 4GiB banks of High Bandwidth Memory (HBM). The HAVAC hardware design was implemented using Vitis High-Level Synthesis (HLS), a tool for synthesizing designs from compliant C/C++ codebases along with FPGA-specific #pragma instructions. Vitis HLS can allow for significantly faster development compared to traditional hardware description languages (HDLs), at a small cost to implemented resource utilization and performance [51]. The HAVAC host (CPU) driver code was implemented in C++ using the Xilinx Runtime Library (XRT).

Profile HMM scores

As in the nhmmer SSV filter, HAVAC computes the scores of the Dynamic Programming (DP) matrix using 8-bit integer emission scores. For a given pHMM, a threshold score t is generated representing the minimum score required for a target database sequence to pass the requested P-value target (default: $P \leq 0.02$) using the pHMM's precomputed gumbel distribution parameters [31] with adjustments to account for SSV's removed state transitions. HAVAC uses the threshold to generate a scaling factor τ on the pHMM's emission scores. The purpose of τ is to reproject the emission scores such that an accumulated score passes the query-specific threshold if and only if the score reaches 256.

$$\tau = 256/t$$

The purpose of using τ to re-project the emission scores is twofold. The primary reason is to simplify the hardware accelerator's cell Processing Elements (PEs). By enforcing a threshold score of 256, the cell PEs can check their score against the threshold by using an 8-bit full adder's carry bit, eschewing the need for a full 8-bit comparator to check for hits. As a result, cell PEs require fewer resources to implement, which allows more PEs to fit into the hardware to improve parallelism. As an added bonus, using τ to reproject the emission scores allows for better use of the 8-bit integer space, allowing for slightly more precision in the emission scores when compressed down to 8-bit integers.

The HMMER3 pHMM format stores emission scores as negative log-likelihood values, and these scores must be converted to bits before being projected using τ. To convert a negative log-likelihood score s to a τ-reprojected score s' in bits, the score is represented as a single-precision floating-point value and extracted from negative log-likelihood space, divided by the background distribution of equal probability for each of 4 nucleotides, converted to bits, and then multiplied by τ.

$$s' = log2(e^{-s}/(1/4)) * \tau$$

The above equation involves performing exp() and log() function calls on each emission score, which are slow on modern hardware. We eliminate the need to perform these expensive functions by simplifying as follows.

$$s' = log2(4e^{-s}) * \tau$$
$$s' = (log2(4) - s * log2(e)) * \tau$$
$$s' = 2\tau - s\tau log2(e)$$

In the resulting equation, the terms 2τ and $\tau log2(e)$ can be computed as constants over all match states of the model; this means that all match states can be computed with a single multiplication and subtraction. Each re-projected emission score s′ is then rounded to the nearest integer, and cast to signed 8-bit integers. Each individual pHMM in the input.hmm file is projected in this way using the same P-value and the data are appended together and written to the FPGA's HBM memory banks.

SSV processor design

Define $M(i, j)$ as the score of the maximum scoring ungapped Viterbi alignment ending at the *i*th letter of the sequence and the *j*th position of the pHMM. For a sequence of length L and a pHMM of length K, this requires computation of LK cells. The HAVAC kernel on the FPGA is comprised mainly of a systolic array of n cell PEs that each individually compute a single cell of the DP matrix each cycle. If $L > n$, then the sequence is procedurally broken into length-n segments. Each cell PE in the systolic array uses a different letter from the sequence to find the match score for the current pHMM position and adds it to the score computed by the previous cell PE on the previous cycle. Initially, the systolic array computes the cells $M(1, 1)$ to $M(n, 1)$ in the DP matrix. The model position is then incremented and the systolic array computes $M(1, 2)$ to $M(n, 2)$, and so on until a full pass through the pHMM has been completed. Figure 3 shows how the sequence is broken down into segments of length n, and how the SSV processor computes rows of length n down the DP matrix. If $L > n$, then a new segment of the sequence is loaded into the cell PEs and another pass through the pHMM begins. These passes through the pHMM compute columns of width n through the DP matrix until the entire matrix has been calculated. Sequences that are not a multiple of N in length are padded with random data to reach that target; SSV matches to these random sequences will be rare, and are easily filtered out by host-side driver.

When the processor is running, n-length sequence segments are read asynchronously from memory and inserted into a hardware FIFO until the entire sequence is consumed. Similarly, the pHMM is read asynchronously from memory in full for each sequence segment, and inserted into its own FIFO. When the processor begins to compute a pass through a sequence segment, a full segment is consumed from the sequence FIFO to set each cell's sequence symbol. Once the sequence is loaded into the cell PEs, the full pHMM is consumed from the FIFO, one model position at a time. Figure 4 shows a

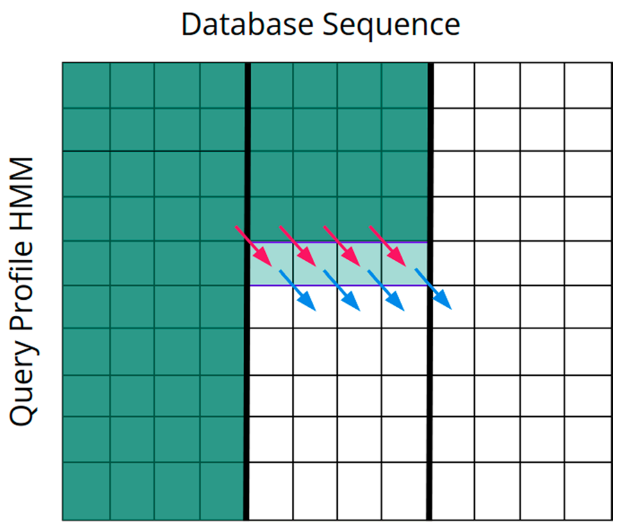

Fig. 3 SSV dynamic programming matrix. The Dynamic Programming matrix for the SSV model. Here, the sequence is comprised of 3 segments of length $N = 4$. The light colored row shows the cells being computed this cycle. The fuchsia arrows indicate the data from the cells $M(i-1, j-1)$. The blue arrows indicate the cells that will use the data generated this cycle to compute $M(i+1, j+1)$

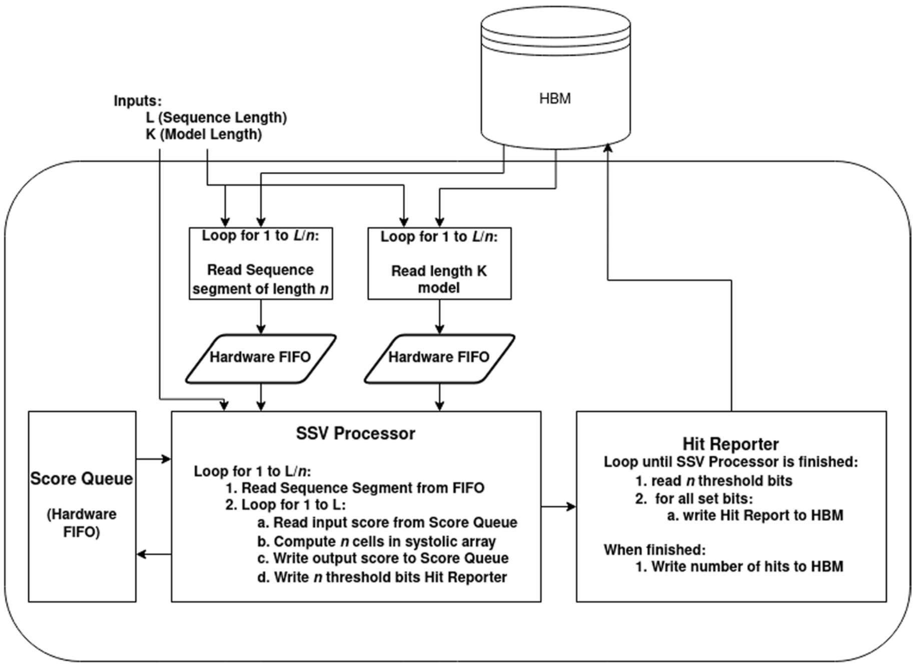
HBM
Inputs:
L (Sequence Length)
K (Model Length)
Loop for 1 to L/n:
Read Sequence segment of length n
Loop for 1 to L/n:
Read length K model
Hardware FIFO
Hardware FIFO
Score Queue
(Hardware FIFO)
SSV Processor
Loop for 1 to L/n:
1. Read Sequence Segment from FIFO
2. Loop for 1 to L:
a. Read input score from Score Queue
b. Compute n cells in systolic array
c. Write output score to Score Queue
d. Write n threshold bits Hit Reporter
Hit Reporter
Loop until SSV Processor is finished:
1. read n threshold bits
2. for all set bits:
a. write Hit Report to HBM
When finished:
1. Write number of hits to HBM

Fig. 4 HAVAC hardware design overview Diagram of the overall design of the HAVAC hardware accelerator. Sequence length and model length are provided as inputs. The hardware then asynchronously loads length n segments of the sequence from HBM. For each of those segments, the model is loaded from HBM. These data are fed to the SSV processor, which calculates the cells of the DP matrix. The score queue module facilitates the transfer of scores from the last column of one sequence segment pass to the first of the next. The vector of bits representing any threshold hits are passed to the hit reporter module, which checks for any hits, determines the model and sequence positions of the hits, and writes the hits to HBM

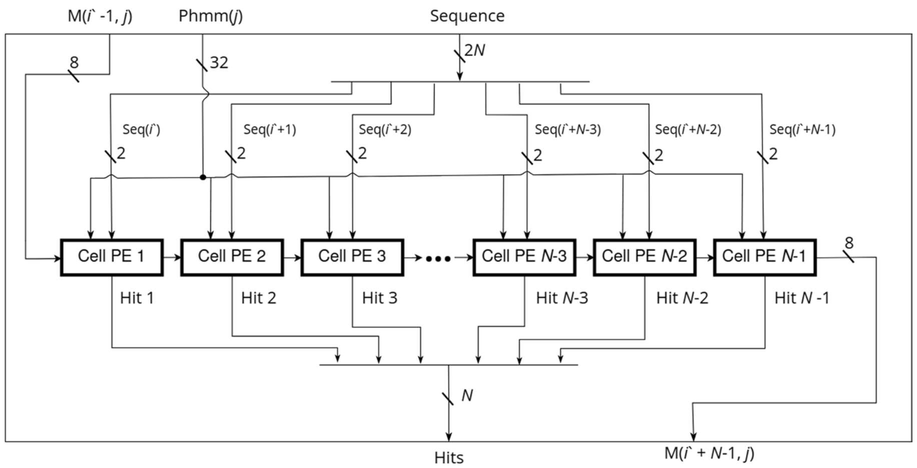

Fig. 5 Systolic array of *N* cell PEs. Each cell PE receives the score that was computed in the previous PE on the previous cycle. All cells use the same pHMM vector on a given cycle, and advance to the next vector on the subsequent cycle. Each cell uses a different symbol from inside the contiguous sequence segment to select the correct emission score for the PE's corresponding DP matrix cell

simplified view into the overall HAVAC hardware accelerator. Figure 5 shows the systolic array of cell PEs that compute the cells of the DP matrix.

For any given sequence segment, the leftmost cell of the systolic array of cell PEs requires input scores from the previous sequence segment column (except the first segment, which requires inputs of 0 for each cell in the first column of the matrix). Likewise, the scores generated by the rightmost cells in the segment column must be used as inputs for the next segment column of the matrix. A score queue module that implements a hardware FIFO structure is used to enqueue the outputs of the rightmost cell in a segment column, and dequeue scores as needed by the next segment column. The score queue module uses a series of 36kb block RAM elements to implement the hardware FIFO. Since hardware block RAMs have a definitive capacity, the score queue module acts as a limiting factor on the total length of phmm vectors that HAVAC can support. HAVAC supports a maximum of 1,048,576 (1024*1024) model positions for a single SSV query. This is large enough to hold the entirety of the Rfam database of RNA families [52](4108 models) twice over.

Hardware cell processing elements

The HAVAC cell PE is designed to minimize the resources required to calculate the score at a given cell. The cell's sequence symbol c is used as the *select* on the current pHMM vector $P(j)$ to obtain the cell's signed 8-bit pHMM emission score. This value is then summed with the score computed by the previous cell PE on the previous cycle, $M(i-1,j-1)$, to obtain an unsigned 8-bit intermediate sum value $T(i, j)$ and a carry bit from the 8-bit full adder. This carry bit is then used along with the sign bit from the emission score to determine if the threshold has been passed and if the final cell's result, $M(i, j)$, should be reset back to zero. There are two situations where the result of the summation should be discarded: if a negative match score was added to $M(i-1,j-1)$ (the carry bit was clear), or if a positive emission score was added to $M(i-1,j-1)$ (the carry bit was set). In the first case the result underflowed the unsigned 8-bit result, and

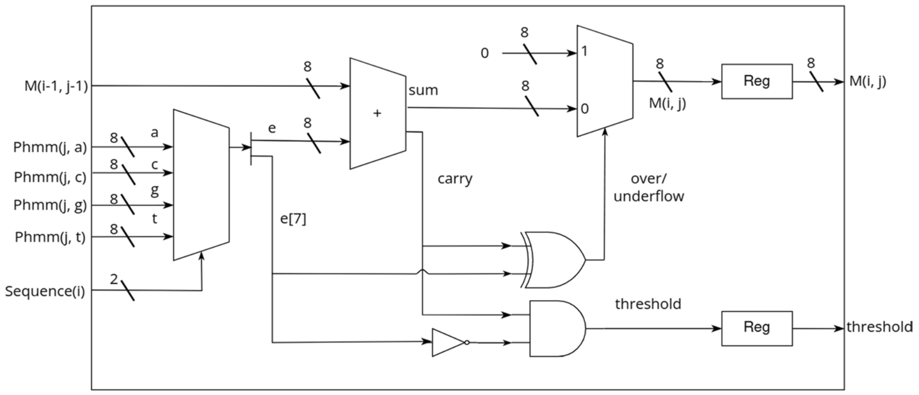

Fig. 6 Logic Diagram of the Cell PE. The pHMM vector's four 8-bit scores corresponding to each nucleotide are de-multiplexed by the sequence symbol, and then added to the score from the previous cell PE. Simple bitwise operations then determine if an overflow or underflow occurred, and reset the score or report a hit accordingly

in the latter case the result overflowed the maximum representable score of 256. In this second case, the given cell has passed the score threshold and should be reported. Figure 6 shows the logic that implements a cell PE.

$$T(i,j) = M(i-1,j-1) + P(j,c)$$
$$M(i,j) = \begin{cases} 0, & \text{if } T(i,j) < 0 \text{ or } T(i,j) > 256 \\ T(i,j), & \text{otherwise} \end{cases}$$

Hit reporting

HAVAC reports any cell that passes the 256 threshold as an (*i*, *j*) position pair in the sequence and pHMM. HAVAC finds these (*i*, *j*) pairs by taking the threshold bit from all *N* cell PEs and systematically partitioning the threshold hit bits to find and report any that were set for a given cycle. These bits represent the cells $i' < i < i' + N$ where i' is the first position of the *N*-length current segment of sequence. First, all *N* threshold hit bits are *bitwise-or* reduced; if the result is 1, the threshold hit bits contain at least 1 cell that passed the threshold. In this case, the threshold hit bits are enqueued to a hardware FIFO along with the current pHMM position and the current sequence segment index. Then, in concurrently running processes, the threshold hit bits are partitioned into 16 contiguous bit ranges. HAVAC iteratively *bitwise-or* reduces each of these 16 bit ranges to determine if a hit was in any of the ranges. If a bit range contained a hit, the bit range is enqueued to a subsequent hardware FIFO along with the rest of the position metadata and the index of the bit range. This process continues, further dividing the threshold hit bits until it represents a single bit. At this point, the hit's position in the sequence can be identified as the index of that bit plus i'. The hits are then written to the FPGA HBM memory where they can be read by the host to extract the (*i*, *j*) pairs after the HAVAC hardware accelerator finishes.

Because the hit report partitioning runs concurrently to the main systolic array, and the hardware does not know a priori how many hit reports will be written for an invocation of the hardware accelerator, an extra terminator bit t is included along with the threshold hit ranges and positional data and is normally cleared. On the final cycle of the matrix computation (the final pHMM position of the final sequence segment index), the full threshold hit bit range is enqueued to the partitioning system even if it does not contain any hits, along with a set terminator bit. Each subsequent partitioning tier then passes along a threshold bit range with a set terminator on the final (16th) section of the bit range, and then deactivates. Once all hit reports are written to memory and the final partitioning tier deactivates, the number of hits is written to memory and the hardware accelerator completes its operation.

Accelerator synthesis and driver

Synthesis of the HAVAC hardware accelerator design was performed inside the Xilinx Vitis HLS tool. Implementation was performed by the v++ compiler tool. The Alveo U50 card contains 8GiB of HBM, split between 32 256MiB banks. Of the 8GiB of HBM available, 4GiB (16 banks) is used to store sequence data. All sequence characters are represented with 2-bit encoding (ambiguity characters are replaced by random letters, which may result in rare cases of falsely passing the filter; these are quickly filtered out by the downstream driver). HAVAC supports target sequences up to an aggregated length of 16Gbp. 512MiB (2 banks) is allocated for the profile HMM match scores. 3.5GiB (14 banks) is allocated for reporting the 8 byte (i, j) pairs for hits. Therefore, a maximum of 469,762,048 hits can be reported for any query. HAVAC was implemented with 12,288 cell PEs at a clock speed of 144.5 MHz. The number of cell PEs was chosen as a multiple of 4096 to allow the sequence data to better align to the hardware's 4KiB memory boundary, while maximizing the number of PEs that could be successfully be synthesized and implemented (by the HLS tool and v++ compiler). The clock speed was determined automatically by the v++ compiler as the maximum clock speed that could be successfully implemented.

The HAVAC driver software library allows for simple control over the hardware accelerator. Input data is provided to the driver as a fasta file and a HMMER3 model file and is then preprocessed and loaded onto the board's HBM memory banks. The SSV computation can then be performed either synchronously or asynchronously. Once the hardware accelerator has finished, hit data can be read from the device as global sequence and model positions. Using these global positions, the HAVAC driver references the fasta and model files to locate which sequence and model the hits resides in, and the local positions in the sequence or model the hit refers to. These sequence and model positions can then be used by other algorithms in a sequence alignment pipeline as positions of potential homology. Because of the nature of HAVAC being a dedicated co-processor, SSV calculation may be performed concurrently to these other algorithms, further reducing the processing time for large queries that can be broken down into smaller batches.

With enough memory to store 16Gbp of sequence and 13 million positions of pHMM data, HAVAC has far more memory than would be required for any single sequence or pHMM. The HAVAC driver concatenates all sequences in a fasta file and all models in an hmm file before transferring the data to the accelerator for processing. Thus, HAVAC searches the sequences and models in an all-to-all manner. The only limit to the number of sequences or pHMMS that HAVAC can process in a single invocation is the size of the memory allocated to each on the device. Once the hits are reported to the host, a fast SSV validity check is performed with the CPU to eliminate any hits that may have been generated by crossing the boundary between any two sequences or models.

Results

We performed benchmarking experiments on a compute system with two Intel Xeon X5570 CPUs @ 2.96GHz, each with 4 cores, and 114 GB of memory, with test data was stored on an Intel 240GB SSD.[1] Tests were performed by searching human chromosome 22 against query databases generated from subsets of the Rfam family model database. Query datasets were generated by selecting models from Rfam, in order of ascending accession ID, up to a specified number of model positions. For example, for the total model database length threshold around 10,000, Rfam families RF00001 through RF00065 were used, with a sum of model lengths equal to 10,122; meanwhile the set of Rfam families RF00001 through RF00163 produce a subset with a total model length of 20,039.

The HAVAC hardware accelerator was implemented on a Xilinx Alveo U50 Data Center accelerator card. FPGA routing and implementation typically uses only a fraction of the full board capacity—HAVAC's final hardware implementation utilized 48.94% of the device's hardware lookup tables (LUTs), 21.53% hardware register, and 35.94% Block RAMs. HAVAC's runtime was timed as 4 discrete sections; (i) software data allocation and configuring the FPGA with the Xilinx hardware device binary (.xclbin) file, (ii) file I/O for the sequence and pHMM files, data preprocessing, and loading these data onto the device's HBM banks, (iii) runtime of the HAVAC SSV kernel, and (iv) reading the generated hits from the device memory and resolving the global (i, j) pairs to local sequence and model positions. It is common to describe the performance of dynamic programming algorithms in terms of billions of cells updated per second (GCUPS). HAVAC's performance was measured to be 1739 GCUPS using the runtime of the kernel from step (iii). This performance represents a realization of 98% of the theoretical maximum speed achievable with 12,288 cell PEs processing a cell every cycle at 144.5 MHz. In comparison, HMMER3 contains a highly optimized CPU implementation of MSV that utilizes 16-way striped SIMD parallelization reported to reach 12 GCUPS on a single thread running on a 2.66 GHz Intel Gainestown X5550 CPU [27]. The nhmmer SSV matrix calculation on our tests system averaged 7.6 GCUPS when single-threaded, and 18.9 GCUPS with four threads. We tested nhmmer with 8 and 16 threads, but saw performance that was nearly identical to 4 threads; this is consistent with reports that HMMER3 becomes I/O bound at low thread count and does not benefit from additional threads past this point [53, 54]. The HAVAC kernel represents a 227$\times$ matrix calculation speedup over nhmmer with one thread and a 92$\times$ speedup over nhmmer with 4 threads. Table 2 shows the performance in terms of GCUPS for HAVAC and previous publications of Viterbi-family algorithms implemented on FPGAs and GPUs.

[1] This a fairly old CPU. When we tested on a newer Xeon E5-2695v4 14 core system (192 GB RAM, all data on a local spinning disk), nmhmmer was 2–4$\times$ *slower*; and when we tested on an AMD EPYC 7642 48-core system (512 GB RAM, all data on local NVMe disk), nhmmer was 30% slower than the Xeon X5570 with 1 threads, and 25% faster with 8 threads.

Table 2 Fastest reported FPGA and GPU accelerator speeds for Viterbi-family algorithms on biological data from the literature

Implementation	Algorithm	Year	GCUPS (Maximum reported)	Device used
Jacob et al. [55]	Viterbi	2007	10.6 (best case est.)	Virtex-II 6000
Walters et al. [56]	Viterbi	2007	0.7	Spartan-3 XC3S1500
Benkrid et al. [57]	Viterbi	2008	9	Virtex-II Pro 2VP100
Oliver et al. [58]	Viterbi	2008	2.1	Spartan-3 XC3S1500
Md Isa et al. [59]	Viterbi	2012	11.8	Virtex 5 XC5VLX110
Abbas et al. [60]	MSV	2015	81	(two) Stratix-III 260
	Viterbi	2015	3.6	
	MSV + Viterbi	2015	45.7	
Jiang and Ganesan [61]	SSV (GPU)	2016	440	Tesla K40
	MSV (GPU)	2016	277	
	Viterbi (GPU)	2016	14.3	
HAVAC	SSV	2023	1739	

HAVAC's performance was also measured as a complete SSV filter including all steps from initial FPGA configuration to resolution of hit positions. These times were compared to nhmmer's SSV filter using 1, 2, and 4 threads with the subsequent stages of the HMMER pipeline disabled. At small query sizes, the hardware configuration and PCI-e data transfer times dominate run time, resulting in poorer performance than nhmmer. HAVAC and nhmmer with a single thread had similar runtime with a query length of 1000 model positions (representing 6 models), and nhmmer with 4 threads was similar with a query length of 3000 model nodes (representing 16 models). As the size of the DP matrix grows, the hardware kernel's performance becomes the dominating factor – when querying the entire Rfam database (4108 models) against human chromosome 22, HAVAC is 65× faster than nhmmer with 1 thread. Figure 7 compares the runtime performance of HAVAC and the nhmmer SSV filter across various lengths of model databases.

Power utilization of the HAVAC hardware accelerator was approximated using the Xilinx Board Utility (xbutil) tool. Voltage and Amperage values were reported for the PCI-express interface and on-board power rails. These mV and mA values were multiplied to find the watts used, and them summed to total energy use estimate of 29 watts, 31% of the 95 watts of the Thermal Design Power (TDP) of the Intel Xeon X5570 in our test system [62]. This TDP represents the average energy usage when all cores are under full load.

Conclusion

In this work, we have described a new implementation of HMMER's SSV filter algorithm on reconfigurable hardware with excellent runtime characteristics. The limited data cell dependencies of SSV allow design of cell PEs that consume very limited resources. Considering that SSV represents ∼ 70% of nhmmer's runtime, a 31× speedup over the current SSV implementation would result in a ∼ 3.5× improvement on the pipeline overall when given 4 threads.

At 50% LUT and 22% register utilization, HAVAC could likely be reimplemented to increase the number of cell PEs in the design. Attempts were made to increase the

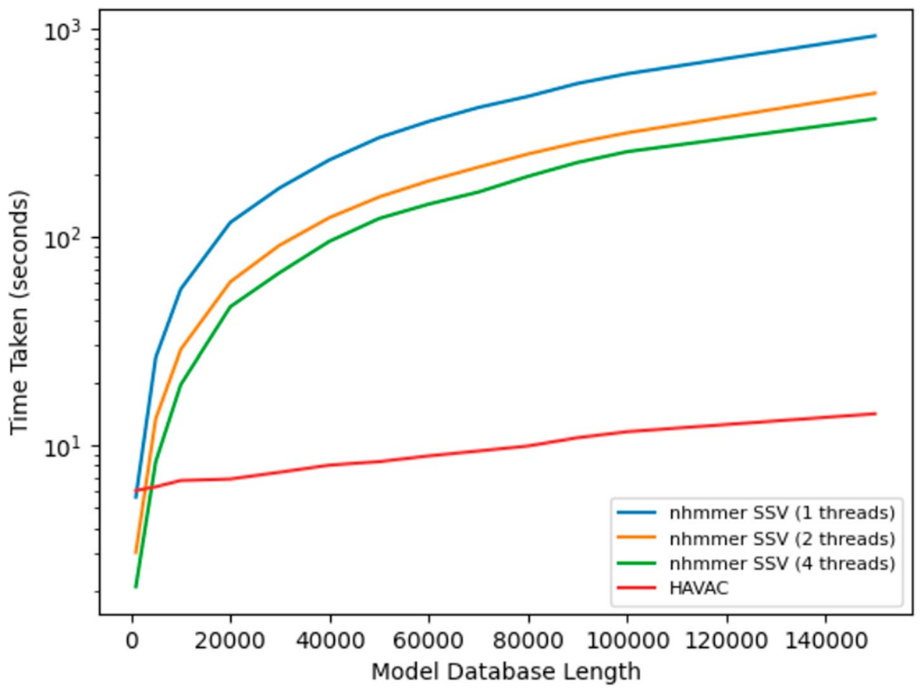

Fig. 7 Runtimes of HAVAC and nhmmer SSV at various numbers of threads

number of cell PEs from 12,288 to 16,384, but the Xilinx v++ implementation stage was unable to generate the design. This is not unreasonable—as utilization grows, FPGA implementation becomes significantly more difficult as signals must be routed around increasingly more congested sections of the design, and may not be able to reach their destination in a reasonable amount of time. Smaller increases to the number of cell PEs are likely possible, although such changes would require modifications to the hit report partitioning system, and may further reduce the maximum implementable clock speed. Importantly, the Alveo U50 represents the lower end of the Xilinx data center accelerator cards, and a mid-range of FPGA accelerators in general, and it is reasonable to expect even faster HAVAC implementations on more powerful hardware.

It is important to remember that the SSV algorithm is a fast approximation of the Viterbi algorithm, but has limited sensitivity by itself. HAVAC efficiently filters to find areas of potential homology between sequences and probabilistic models, but is not an effective tool without a downstream pipeline of more accurate algorithms. As such, HAVAC is only an effective tool in conjunction with the rest of a bioinformatics pipeline. Future work would necessarily involve combining HAVAC with CPU-based downstream Viterbi or Forward algorithms to make a tool that would be applicable to the overall task of genome annotation.

Acknowledgements
We thank George Vacek who piqued our interest in FPGA acceleration of pHMM search while at Convey Computing. We are grateful to the Computer Architecture Group (and especially Farzad Fatollahl-Fard) at Lawerence Berkeley National Labs (LBNL) for providing access to a development FPGA system early in the project. We thank Xilinx for providing a free Alveo U50 FPGA accelerator platform through their Xilinx University Program. Additionally, Patrick McInerney for suggestions for simplifying the equation for profile HMM re-projection. We also gratefully acknowledge the computational resources and expert administration provided by the University of Montana's Griz Shared Computing Cluster (GSCC).

Author contributions

TA and TJW collaborated in project conception and design. TA designed and developed all software, configuration, and experimentation. TA prepared all figures and wrote the initial draft of the manuscript. TJW edited the manuscript, and both authors have approved the final version of the manuscript.

Funding

TA and TJW were supported by NIH NIGMS P20GM103546 and R01GM132600, and by DOE BER DE-SC0021216.

Availability of data and materials

HAVAC code and configuration is available under an open BSD-3-Clause license at https://github.com/TravisWheelerLab/HAVAC. Description and code for benchmarking analysis can be found associated with the software release, in https://github.com/TravisWheelerLab/HAVAC/tree/main/benchmark.

Declarations

Ethics approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Competing interests

The authors declare that they have no competing interest.

Received: 20 December 2023 Accepted: 23 July 2024
Published online: 29 July 2024

References

1. Sahlin K, Baudeau T, Cazaux B, Marchet C. A survey of mapping algorithms in the long-reads era. Genome Biol. 2023;24(1):133.
2. Alser M, Rotman J, Deshpande D, Taraszka K, Shi H, Baykal PI, Yang HT, Xue V, Knyazev S, Singer BD, Balliu B, Koslicki D, Skums P, Zelikovsky A, Alkan C, Mutlu O, Mangul S. Technology dictates algorithms: recent developments in read alignment. Genome Biol. 2021;22(1):249. https://doi.org/10.1186/s13059-021-02443-7.
3. Reinert K, Langmead B, Weese D, Evers DJ. Alignment of next-generation sequencing reads. Annu Rev Genomics Hum Genet. 2015;16(1):133–51. https://doi.org/10.1146/annurev-genom-090413-025358. (**PMID: 25939052**).
4. Trapnell C, Salzberg SL. How to map billions of short reads onto genomes. Nat Biotechnol. 2009;27(5):455–7. https://doi.org/10.1038/nbt0509-455.
5. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. J Mol Biol. 1990;215(3):403–10. https://doi.org/10.1016/S0022-2836(05)80360-2.
6. Henikoff S, Henikoff JG. Amino acid substitution matrices from protein blocks. Proc Natl Acad Sci USA. 1992;89(22):10915–9.
7. Müller T, Spang R, Vingron M. Estimating amino acid substitution models: a comparison of Dayhoff's estimator, the resolvent approach and a maximum likelihood method. Mol Biol Evol. 2002;19(1):8–13.
8. Altschul SF, Madden TL, Schäffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res. 1997;25(17):3389–402. https://doi.org/10.1093/nar/25.17.3389.
9. Henikoff JG, Henikoff S. Using substitution probabilities to improve position-specific scoring matrices. Bioinformatics. 1996;12(2):135–43. https://doi.org/10.1093/bioinformatics/12.2.135.
10. Schäffer AA, Wolf YI, Ponting CP, Koonin EV, Aravind L, Altschul SF. IMPALA: matching a protein sequence against a collection of PSI-BLAST-constructed position-specific score matrices. Bioinformatics. 1999;15(12):1000–11.
11. Eddy SR. Profile hidden Markov models. Bioinformatics (Oxford, England). 1998;14(9):755–63.
12. Sjölander K, Karplus K, Brown M, Hughey R, Krogh A, Mian IS, Haussler D. Dirichlet mixtures: a method for improved detection of weak but significant protein sequence homology. Bioinformatics. 1996;12(4):327–45.
13. Ahola V, Aittokallio T, Uusipaikka E, Vihinen M. Efficient estimation of emission probabilities in profile hidden Markov models. Bioinformatics. 2003;19(18):2359–68.
14. Madera M, Gough J. A comparison of profile hidden Markov model procedures for remote homology detection. Nucleic Acids Res. 2002;30(19):4321–8. https://doi.org/10.1093/nar/gkf544.
15. Wheeler TJ, Eddy SR. nhmmer: DNA homology search with profile HMMs. Bioinformatics. 2013;29(19):2487–9. https://doi.org/10.1093/bioinformatics/btt403.
16. Kiełbasa SM, Wan R, Sato K, Horton P, Frith MC. Adaptive seeds tame genomic sequence comparison. Genome Res. 2011;21(3):487–93.
17. Ma B, Tromp J, Li M. PatternHunter: faster and more sensitive homology search. Bioinformatics. 2002;18(3):440–5.
18. Kent WJ. Blat-the blast-like alignment tool. Genome Res. 2002;12(4):656–64.
19. Söding J. Protein homology detection by HMM-HMM comparison. Bioinformatics. 2005;21(7):951–60.
20. Buchfink B, Xie C, Huson DH. Fast and sensitive protein alignment using diamond. Nat Methods. 2015;12(1):59–60.
21. Steinegger M, Söding J. MMseqs2 enables sensitive protein sequence searching for the analysis of massive data sets. Nat Biotechnol. 2017;35(11):1026–8.

22. Hassaballah M, Omran S, Mahdy YB. A review of SIMD multimedia extensions and their usage in scientific and engineering applications. Comput J. 2008;51(6):630–49.
23. Wozniak A. Using video-oriented instructions to speed up sequence comparison. Bioinformatics. 1997;13(2):145–50.
24. Gálvez S, Agostini F, Caselli J, Hernandez P, Dorado G. BLVector: Fast BLAST-like algorithm for manycore CPU with vectorization. Front Genet. 2021;12:618659. https://doi.org/10.3389/fgene.2021.618659.
25. Rognes T. Faster Smith–Waterman database searches with inter-sequence SIMD parallelisation. BMC Bioinform. 2011;12(1):1–11.
26. Liu D, Steinegger M. Block aligner: an adaptive SIMD-accelerated aligner for sequences and position-specific scoring matrices. Bioinformatics. 2023;39:487.
27. Eddy SR. Accelerated profile hmm searches. PLoS Comput Biol. 2011;7(10):1–16. https://doi.org/10.1371/journal.pcbi.1002195.
28. Steinegger M, Meier M, Mirdita M, Vöhringer H, Haunsberger SJ, Söding J. HH-suite3 for fast remote homology detection and deep protein annotation. BMC Bioinform. 2019;20(1):1–15.
29. Li J-k, Zhang L, Xiao M. The high performance computing applications for bioinformatics research. In: Proceedings of the 6th international conference on bioinformatics and biomedical science. 2017. pp. 70–75.
30. Hasan L, Al-Ars Z. An overview of hardware-based acceleration of biological sequence alignment. In: Computational biology and applied bioinformatics. InTech; 2011. https://doi.org/10.5772/23044 .
31. Eddy SR. A probabilistic model of local sequence alignment that simplifies statistical significance estimation. PLoS Comput Biol. 2008;4(5):1000069.
32. Durbin R, Eddy SR, Krogh A, Mitchison G. Biological sequence analysis: probabilistic models of proteins and nucleic acids. Cambridge University Press; 1998. https://doi.org/10.1017/cbo9780511790492 .
33. Rabiner LR. A tutorial on hidden Markov models and selected applications in speech recognition. Proc IEEE. 1989;77(2):257–86.
34. Frith MC, Hamada M, Horton P. Parameters for accurate genome alignment. BMC Bioinform. 2010;11:1–14.
35. Smith TF, Waterman MS, et al. Identification of common molecular subsequences. J Mol Biol. 1981;147(1):195–7.
36. Eddy SR. Accelerated profile hmm searches. PLoS Comput Biol. 2011;7(10):1002195.
37. Farrar M. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. Bioinformatics. 2006;23(2):156–61. https://doi.org/10.1093/bioinformatics/btl582.
38. Horn DR, Houston M, Hanrahan P. ClawHMMER: a streaming HMMER-search implementatio. In: SC '05: Proceedings of the 2005 ACM/IEEE conference on supercomputing. 2005. pp. 11–11. https://doi.org/10.1109/SC.2005.18
39. Jiang H, Ganesan N. CUDAMPF: a multi-tiered parallel framework for accelerating protein sequence search in HMMER on CUDA-enabled GPU. BMC Bioinform. 2016;17:106. https://doi.org/10.1186/s12859-016-0946-4.
40. Derrien S, Quinton P. Parallelizing HMMER for hardware acceleration on FPGAs. In: 2007 IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP). 2007. pp. 10–17. https://doi.org/10.1109/ASAP.2007.4429951
41. Rucci E, Garcia C, Botella G, De Giusti A, Naiouf M, Prieto-Matias M. OSWALD: OpenCL Smith–Waterman on Alteras FPGA for large protein databases. Int J High Perform Comput Appl. 2016;32:337–50. https://doi.org/10.1177/1094342016654215.
42. Rucci E, Garcia C, Botella G, De Giusti A, Naiouf M, Prieto Matias M. SWIFOLD: Smith–Waterman implementation on FPGA with OpenCL for long DNA sequences. BMC Syst Biol. 2018;12:43–53. https://doi.org/10.1186/s12918-018-0614-6.
43. Li I, Shum W, Truong K. 160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (FPGA). BMC Bioinform. 2007;8:185. https://doi.org/10.1186/1471-2105-8-185.
44. Derrien S, Quinton P. Hardware acceleration of HMMER on FPGAs. J Signal Process Syst. 2010;58(1):53–67. https://doi.org/10.1007/s11265-008-0262-y.
45. Ren S, Sima V-M, Al-Ars Z. FPGA acceleration of the pair-HMMs forward algorithm for DNA sequence analysis. In: 2015 IEEE international conference on bioinformatics and biomedicine (BIBM). 2015. pp. 1465–1470. https://doi.org/10.1109/BIBM.2015.7359892
46. Maddimsetty R, Buhler J, Chamberlain R, Franklin M, Harris B. Accelerator design for protein sequence hmm search. In: Proceedings of the 20th annual international conference on supercomputing, ICS 2006. Proceedings of the international conference on supercomputing. 2006. pp. 288–296. https://doi.org/10.1145/1183401.1183442 . 20th Annual International Conference on Supercomputing, ICS 2006 ; Conference date: 28-06-2006 Through 01-07-2006
47. Oliver T, Yeow LY, Schmidt B. Integrating FPGA acceleration into HMMer. Parallel Comput. 2008;34(11):681–91. https://doi.org/10.1016/j.parco.2008.08.003.
48. Abbas N, Derrien S, Rajopadhye S, Quinton P. Accelerating HMMER on FPGA using parallel prefixes and reductions. In: 2010 International Conference on Field-Programmable Technology. 2010. pp. 37–44. https://doi.org/10.1109/FPT.2010.5681755
49. Nowak F, Bromberger M, Schindewolf M, Karl W. Multi-parallel prefiltering on the convey HC-1 for supporting homology detection. In: Proceedings of the 20th European MPI Users Group Meeting. 2013. pp. 169–174. https://doi.org/10.1145/2488551.2488587
50. Remmert M, Biegert A, Hauser A, Söding J. HHblits: lightning-fast iterative protein sequence searching by HMM-HMM alignment. Nat Methods. 2011;9(2):173–5. https://doi.org/10.1038/nmeth.1818.
51. Lee HS, Jeon JW. Comparison between HLS and HDL image processing in FPGAs. In: 2020 IEEE International Conference on Consumer Electronics—Asia (ICCE-Asia). 2020. pp. 1–2. https://doi.org/10.1109/ICCE-Asia49877.2020.9276803
52. Griffiths-Jones S. Rfam: an RNA family database. Nucleic Acids Res. 2003;31(1):439–41. https://doi.org/10.1093/nar/gkg006.
53. EddyRivasLab Github Issue 161: multithreads running problem. https://github.com/EddyRivasLab/hmmer/issues/161. Accessed: 2023-09-18
54. Arndt W. Modifying HMMER3 to run efficiently on the Cori supercomputer using OpenMP tasking. In: 2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW). IEEE; 2018. pp. 239–246.

55. Jacob AC, Lancaster JM, Buhler JD, Chamberlain RD. Preliminary results in accelerating profile HMM search on FPGAs. In: 2007 IEEE International Parallel and Distributed Processing Symposium. 2007. pp. 1–8. https://doi.org/10.1109/IPDPS.2007.370447
56. Walters JP, Meng X, Chaudhary V, Oliver T, Yeow LY, Schmidt B, Nathan D, Landman J. MPI-HMMER-boost: Distributed FPGA acceleration. J VLSI Signal Process Syst Signal Image Video Technol. 2007;48(3):223–38. https://doi.org/10.1007/s11265-007-0062-9.
57. Benkrid K, Velentzas P, Kasap S. A high performance reconfigurable core for motif searching using profile hmm. In: 2008 NASA/ESA Conference on Adaptive Hardware and Systems. 2008. pp 285–292. https://doi.org/10.1109/AHS.2008.16
58. Oliver T, Yeow LY, Schmidt B. Integrating FPGA acceleration into HMMer. Parallel Comput. 2008;34(11):681–91. https://doi.org/10.1016/j.parco.2008.08.003.
59. Isa MNM, Benkrid K, Clayton T. A novel efficient FPGA architecture for HMMER acceleration. In: 2012 international conference on reconfigurable computing and FPGAs. IEEE. 2012. https://doi.org/10.1109/reconfig.2012.6416723
60. Abbas N, Derrien S, Rajopadhye S, Quinton P, Cornu A, Lavenier D. Combining execution pipelines to improve parallel implementation of HMMER on FPGA. Microprocess Microsyst. 2015;39(7):457–70. https://doi.org/10.1016/j.micpro.2015.06.006.
61. Jiang H, Ganesan N. CUDAMPF: a multi-tiered parallel framework for accelerating protein sequence search in HMMER on CUDA-enabled GPU. BMC Bioinform. 2016. https://doi.org/10.1186/s12859-016-0946-4.
62. Intel. https://www.intel.com/content/www/us/en/products/sku/37111/intel-xeon-processor-x5570-8m-cache-2-93-ghz-6-40-gts-intel-qpi/specifications.html

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.