ApHMM: Accelerating Profile Hidden Markov Models for Fast and Energy-Efficient Genome Analysis

Can Firtina[1]   Kamlesh Pillai[2]   Gurpreet S. Kalsi[2]   Bharathwaj Suresh[2]   Damla Senol Cali[3]
Jeremie S. Kim[1]   Taha Shahroodi[4]   Meryem Banu Cavlak[1]   Joel Lindegger[1]   Mohammed Alser[1]
Juan Gómez Luna[1]   Sreenivas Subramoney[2]   Onur Mutlu[1]

[1]*ETH Zurich*   [2]*Intel Labs*   [3]*Bionano Genomics*   [4]*TU Delft*

Profile hidden Markov models (pHMMs) are widely used in many bioinformatics applications to accurately identify similarities between biological sequences (e.g., DNA or protein sequences). PHMMs represent sequences with a graph structure such that their states and edges account for any modification (i.e., insertions, deletions, and substitutions) by assigning probabilities to these modifications. These probabilities are then used to calculate the similarity score of a sequence compared to a pHMM graph. The Baum-Welch algorithm is a commonly-used and highly-accurate method that sets and uses these probabilities of a pHMM graph to maximize and calculate similarity scores. Accurately setting and using the probabilities in pHMMs is essential to identify similarities between sequences correctly. However, the Baum-Welch algorithm is computationally expensive, and existing works provide either software- or hardware-only solutions for a fixed pHMM design. When we analyze the state-of-the-art works, we find that there is a pressing need for a flexible, high-performant, and energy-efficient hardware-software co-design to efficiently and effectively solve all the major inefficiencies in the Baum-Welch algorithm for pHMMs.

We propose ApHMM, *the* first *flexible acceleration framework that can significantly reduce computational and energy overheads of the Baum-Welch algorithm for pHMMs. ApHMM leverages hardware-software co-design to solve the major inefficiencies in the Baum-Welch algorithm by 1) designing a flexible hardware to support different pHMMs designs, 2) exploiting the predictable data dependency pattern in an on-chip memory with memoization techniques, 3) quickly eliminating negligible computations with a hardware-based filter, and 4) minimizing the redundant computations. We implement our hardware-software optimizations on a specialized hardware to provide the* first *flexible Baum-Welch accelerator for pHMMs. We also provide the* first *GPU implementation of the Baum-Welch algorithm for pHMMs that includes our software optimizations. Our evaluation shows that ApHMM provides significant speedups of* $15.55\times$ - $260.03\times$, $1.83\times$ - $5.34\times$, *and* $27.97\times$ *when compared to CPU, GPU, and FPGA implementations of the Baum-Welch algorithm, respectively. ApHMM outperforms the state-of-the-art CPU implementations of three important bioinformatics applications, 1) error correction, 2) protein family search, and 3) multiple sequence alignment, by* $1.29\times$ - $59.94\times$, $1.03\times$ - $1.75\times$, *and* $1.03\times$ - $1.95\times$, *respectively, while improving their energy efficiency by* $64.24\times$ - $115.46\times$, $1.75\times$, $1.96\times$.

1. Introduction

Hidden Markov Models (HMMs) are useful for calculating the probability of a sequence of previously unknown (hidden) events (e.g., the weather condition) given observed events (e.g., clothing choice of a person) [28]. To calculate the probability, HMMs use a graph structure where a sequence of nodes (i.e., states) are visited based on the series of observations with a certain probability associated with visiting a state from another. HMMs are very efficient in decoding the continuous and discrete series of events in many applications [79] such as speech recognition [41, 63, 75, 79, 83, 91, 131], text classification [3, 51, 86, 122, 135], gesture recognition [19, 23, 40, 73, 80, 87, 108, 110], and bioinformatics [16, 27, 44, 64, 85, 115, 129, 132, 137]. The graph structures (i.e., designs) of HMMs are typically tailored for each application, which defines the *roles* and probabilities of the states and edges connecting these states, called *transitions*. One important special design of HMMs is known as the *profile Hidden Markov Model* (pHMM) design [27], which is commonly adopted in bioinformatics [11, 13, 26, 29, 30, 33, 34, 36, 58, 71, 107, 113, 128, 133, 138], malware detection [4, 10, 67, 95, 97, 102] and pattern matching [25, 52, 66, 68, 100, 101].

Identifying differences between biological sequences (e.g., DNA sequences) is an essential step in bioinformatics applications to understand the effects of these differences (e.g., genetic variations and their relations to certain diseases). PHMMs enable efficient and accurate identification of differences by comparing sequences to a few graphs that represent a group of sequences rather than comparing many sequences to each other, which is computationally very costly and requires special hardware and software optimizations [5–9, 35, 37, 54–56, 74, 84, 104, 105, 109, 118]. Figure 1 illustrates a *traditional* design of pHMMs. A pHMM represents a single or many sequences with a graph structure using states and transitions. There are three types of states for each character of a sequence that a pHMM graph represents: insertion (`I`), match or mismatch (`M`), and deletion (`D`) states. Each state accounts for a certain difference or a match between a graph and an *input sequence* at a particular position. For example, the `I` states recognize insertions in an input sequence that are missing from the pHMM graph at a position. Many bioinformatics applications use pHMM graphs rather than directly comparing sequences to avoid the high cost of many sequence comparisons. The applications that use pHMMs include protein family search [11, 13, 32, 36, 72, 113, 114, 138], multiple sequence alignment (MSA) [10, 26, 29–31, 71, 82, 92, 107, 113, 128], and error correction [33, 34, 58].

To accurately model and compare DNA or protein sequences using pHMMs, assigning accurate probabilities to states and transitions is essential. PHMMs allow updating these probabilities to fit the observed biological sequences to the pHMM graph accurately. Probabilities are adjusted during the *training*
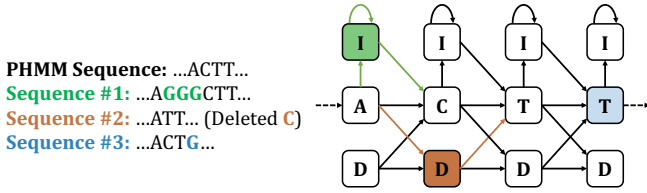
Figure 1: Portion of an example pHMM design that represents a DNA sequence (`PHMM Sequence`). Differences between `PHMM Sequence` and Sequences #1, #2, and #3 are highlighted by colors. Highlighted transitions and states identify each corresponding difference.

step. The training step aims to maximize the probability of observing the input biological sequences in a given pHMM, also known as *likelihood maximization*. There are several algorithms that perform such a maximization in pHMMs [14, 59, 99, 103]. The Baum-Welch algorithm [14] is commonly used to calculate the likelihood maximization [17] as it is highly accurate and scalable to real-size problems (e.g., large protein families) [59]. The next step is *inference*, which aims to identify either 1) the similarity of an input observation sequence to a pHMM graph or 2) the sequence with the highest similarity to the pHMM graph, which is known as the *consensus sequence* of the pHMM graph and used for error correction in biological sequences. Parts of the Baum-Welch algorithm can be used for calculating the similarity of an input sequence in the inference step.

Despite its advantages, the Baum-Welch algorithm is a computationally expensive method [50, 69] due to the nature of its dynamic programming approach. Several works [29, 93, 98, 112, 134] aim to accelerate either the entire or smaller parts of the Baum-Welch algorithm for HMMs or pHMMs to mitigate the high computational costs. While these works can improve the performance for executing the Baum-Welch algorithm, they either 1) provide software- or hardware-only solutions for a fixed pHMM-design or 2) are completely oblivious to the pHMM design.

To identify the inefficiencies in using pHMMs with the Baum-Welch algorithm, we analyze the state-of-the-art implementations of three pHMM-based bioinformatics applications: 1) error correction, 2) protein family search, and 3) multiple sequence alignment (Section 3). We make six key observations. 1) The Baum-Welch algorithm is the main performance bottleneck in the pHMM applications as it constitutes at least around 50% of the total execution time of these applications. 2) SIMD-based approaches cannot fully vectorize the floating-point operations. 3) Significant portion of floating-point operations is redundant in the training step due to a lack of a mechanism for reusing the same products. 4) Existing strategies for filtering out the negligible states from the computation are costly despite their advantages. 5) The spatial locality inherent in pHMMs cannot be exploited in generic HMM-based accelerators and applications as these accelerators and applications are oblivious to the design of HMMs. 6) The Baum-Welch algorithm is the bottleneck even for the non-genomic application we evaluate. Unfortunately, software- or hardware-only solutions cannot solve these inefficiencies. These observa-

tions show that there is a pressing need for a flexible, high-performant, and energy-efficient hardware-software co-design to efficiently and effectively solve these inefficiencies in the Baum-Welch algorithm for pHMMs.

Our **goal** is to accelerate the Baum-Welch algorithm while eliminating the inefficiencies when executing the Baum-Welch algorithm for pHMMs. To this end, we propose ApHMM, the *first* flexible hardware-software co-designed acceleration framework that can significantly reduce the computational and energy overheads of the Baum-Welch algorithm for pHMMs. ApHMM is built on four **key mechanisms**. First, ApHMM is highly flexible that can use different pHMM designs with the ability to change certain parameter choices to enable the adoption of ApHMM for many pHMM-based applications. This enables 1) additional support for pHMM-based error correction [33, 34, 58] that traditional pHMM design cannot efficiently and accurately support [33]. Second, ApHMM exploits the spatial locality that pHMMs provide with the Baum-Welch algorithm by efficiently utilizing on-chip memories with memoizing techniques. Third, ApHMM efficiently eliminates negligible computations with a hardware-based filter design. Fourth, ApHMM avoids redundant floating-point operations by 1) providing a mechanism for efficiently reusing the most common products of multiplications in lookup tables (LUTs) and 2) identifying pipelining and broadcasting opportunities where certain computations are moved between multiple steps in the Baum-Welch algorithm without extra storage or computational overheads. Among these mechanisms, the fourth mechanism includes our software optimizations, while on-chip memory and hardware-based filter require a special and efficient hardware design.

To evaluate ApHMM, we 1) design a flexible hardware-software co-designed acceleration framework in an accelerator and 2) implement the software optimizations for GPUs. We evaluate the performance and energy efficiency of ApHMM for executing 1) the Baum-Welch algorithm and 2) several pHMM-based applications and compare ApHMM to the corresponding CPU, GPU, and FPGA baselines. First, our extensive evaluations show that ApHMM provides significant 1) speedup for executing the Baum-Welch algorithm by 15.55×- 260.03× (CPU), 1.83×- 5.34× (GPU), and 27.97× (FPGA) and 2) energy efficiency by 2474.09× (CPU) and 896.70×-2622.94× (GPU). Second, ApHMM improves the overall runtime of the pHMM-based applications, error correction, protein family search, and MSA, by 1.29×- 59.94×, 1.03×- 1.75×, and 1.03×- 1.95× and reduces their overall energy consumption by 64.24×- 115.46×, 1.75×, 1.96× over their state-of-the-art CPU, GPU, and FPGA implementations, respectively. We make the following **key contributions**:

- We introduce ApHMM, the *first* flexible hardware-software co-designed framework to accelerate pHMMs. We show that our framework can be used at least for three bioinformatics applications: 1) error correction, 2) protein family search, and 3) multiple sequence alignment.

- We provide ApHMM-GPU, the *first* GPU implementation of the Baum-Welch algorithm for pHMMs, which includes our software optimizations.
- We identify key inefficiencies in the state-of-the-art pHMM applications and provide key mechanisms with efficient hardware and software optimizations for significantly reducing the computational and energy overhead of the Baum-Welch algorithm for pHMMs.
- We show that ApHMM provides significant speedups and energy reductions for executing the Baum-Welch algorithm compared to the CPU, GPU, and FPGA implementations, while ApHMM-GPU performs better than the state-of-the-art GPU implementation.
- We provide the source code of our software optimizations, ApHMM-GPU, as implemented in an error correction application. The source code is available at `https://github.com/CMU-SAFARI/ApHMM-GPU`.

2. Background

2.1. Profile Hidden Markov Models (pHMMs)

2.1.1. High-level Overview. We explain the design of profile Hidden Markov Models (pHMMs). Figure 1 shows the *traditional* structure of pHMMs. To represent a biological sequence and account for differences between the represented sequence and other sequences, pHMMs have a constrained graph structure. Visiting nodes, called *states*, via directed edges, called *transitions*, are associated with certain probabilities to identify differences. To assign a probability for any modification at any sequence position, states are created for each character of the represented sequence. When visited, states may *emit* one of the characters from the defined alphabet of the biological sequence (e.g., A, C, T, and G in DNA sequences) with a certain probability. Transitions preserve the correct order of the represented sequence while allowing insertions and deletions to the represented sequence.

To represent and compare biological sequences, pHMMs are used in three steps. First, to represent a sequence, pHMM builds the states and transitions by iterating over each character of the sequence. Multiple sequences can also be represented with a single pHMM graph. A typical pHMM graph includes insertion, match/mismatch, and deletion states for each character of the represented sequence. Connections between states have predefined patterns, as illustrated in Figure 1. Match states have connections to only match and deletion states of the next character and insertion state of the same character. Deletion states connect to match and deletion states of the next character. Insertion states connect to themselves with a loop and the match state of the next character. The flow from previous to next characters ensures the correct order of the represented sequence in a pHMM graph.

Second, the training step maximizes the similarity score of sequences that are similar to the sequence that the pHMM graph represents. To this end, the training step uses additional input sequences as observation to modify the probabilities of the pHMM. The Baum-Welch algorithm [14] is a highly accurate training algorithm for pHMMs.

Third, the inference step aims to either 1) calculate the similarity score of an input sequence to the sequence represented by a pHMM or 2) identify the consensus sequence that generates the best similarity score from a pHMM graph. 1) Calculating the similarity score is useful for applications such as protein family search and MSA. This is because pHMM graphs can avoid making redundant comparisons between sequences by comparing a sequence to a single pHMM graph that represents multiple sequences. Parts of the Baum-Welch algorithm (i.e., the Forward and Backward calculations) can be used in this step for calculating the scores [29]. 2) The goal of generating the consensus sequence is to identify the modifications that need to be applied to the represented sequence. These modifications enable error correction tools to identify and correct the errors in DNA sequences. Decoding algorithms such as the Viterbi decoding [123] are commonly used for inference from pHMMs [36, 53].

2.1.2. Components of pHMMs. We formally define the pHMM graph structure and its components. We assume that pHMM is a graph, $G(V, A)$, the sequence that the pHMM represents is S_G, and the length of the sequence is n_{S_G}. To accurately represent a sequence, pHMMs use four components: 1) states, 2) transitions, 3) emission, and 4) transition probabilities. We represent the *states* and *transitions* as the members of the sets V and A, respectively. First, for each character of sequence S_G at position t, $S_G[t] \in S_G$, pHMMs include 3 consecutive states, v_{3t}, v_{3t+1}, and $v_{3t+2} \in V$: 1) match, 2) insertion, and 3) deletion states. Each of these states modifies the character $S_G[t]$, inserts additional characters after $S_G[t]$, or deletes $S_G[t]$. Second, pHMM graphs include transitions from state v_i to state v_j, $\alpha_{ij} \in A$, such that the condition $i \leq j$ always holds true to preserve the correct order of characters in S_G. Third, to define how probable to observe a certain character when a state is visited, emission probabilities are assigned for each character in a state. These emission probabilities can account for matches and substitutions in match states when comparing a sequence to a pHMM graph. We represent the emission probability of character c in state v_i as $e_c(v_i)$. Fourth, to identify the series of states to visit, probabilities are assigned to transitions. We represent the transition probability of a character between states v_i and v_j as α_{ij}. These four main components build up the entire pHMM graph to represent a sequence and calculate the similarity scores when compared to other sequences.

2.1.3. Identifying the Modifications. Figure 1 shows three types of modifications that pHMMs can identify, 1) insertions, 2) deletions, and 3) substitutions when comparing the sequence a pHMM represents (i.e., `PHMM Sequence` in Figure 1) to other sequences. First, insertion states can identify the characters that are missing from the pHMM sequence at a certain position. For example, Sequence #1 in Figure 1 includes three additional `G` characters after `A`. To identify such insertions, the highlighted insertion state `I` can be taken three times after visiting the state with label `A`. Second, deletion states can identify the characters that are deleted from the sequences we compare with the pHMM sequence. Sequence #2 in Figure 1

provides significant similarity to the pHMM sequence only with a single character missing. To identify the missing character, the highlighted deletion state is visited as it corresponds to deleting the second character in the pHMM sequence, C. Third, match states can identify the characters in sequences different than the character at the same position of a pHMM sequence, which we call substitutions. The states in Figure 1 with DNA letters are match states and show the characters they represent in the corresponding pHMM sequence. The last character of Sequence #3 is different than the last character of the pHMM sequence in Figure 1. Such a substitution is identified by visiting the highlighted match state of the last character of the pHMM sequence.

2.2. The Baum-Welch Algorithm

To maximize or calculate the similarity score of input observation sequences in a pHMM graph, the Baum-Welch algorithm [14] solves an *expectation-maximization* problem [45, 65, 78, 117], where the *expectation* step calculates the statistical values based on an input sequence to train the probabilities of pHMMs. To this end, the algorithm performs the expectation-maximization based on an observation sequence S for the pHMM graph $G(V, A)$ in three steps: 1) forward calculation, 2) backward calculation, and 3) parameter updates.

2.2.1. Forward Calculation.
The goal of performing the forward calculation is to compute the probability of observing sequence S when we compare S and S_G from their first characters to the last characters. Equation 1 shows the calculation of the forward value $F_t(i)$ of state v_i for character $S[t]$. The forward value, $F_t(i)$, represents the likelihood of emitting the character $S[t]$ in state v_i given that *all* previous characters $S[1 \ldots t-1]$ are emitted by following an *unknown* path *forward* that leads to state v_i. $F_t(i)$ is calculated for all states $v_i \in V$ and for all characters of S. Although t represents the position of the character of S, we use the *timestamp* term for t for the remainder of this paper. To represent transition and emission probabilities, we use the α_{ji} and $e_{S[t]}(v_i)$ notations as we define in Section 2.1.2.

$$F_t(i) = \sum_{j \in V} F_{t-1}(j)\alpha_{ji}e_{S[t]}(v_i) \quad i \in V, \;\; 1 < t \le n_S \qquad (1)$$

2.2.2. Backward Calculation.
The goal of the backward calculation is to compute the probability of observing sequence S when we compare S and S_G from their last characters to the first characters. Equation 2 shows the calculation of the backward value $B_t(i)$ of state v_i for character $S[t]$. The backward value, $B_t(i)$, represents the likelihood of emitting $S[t]$ in state v_i given that *all* further characters $S[t+1 \ldots n_S]$ are emitted by following an *unknown* path *backwards* (i.e., taking transitions in reverse order). $B_t(i)$ is calculated for all states $v_i \in V$ and for all characters of S.

$$B_t(i) = \sum_{j \in V} B_{t+1}(j)\alpha_{ij}e_{S[t+1]}(v_j) \quad i \in V, \; 1 \le t < n_S \qquad (2)$$

2.2.3. Parameter Updates.
The Baum-Welch algorithm uses the values that the forward and backward calculations gen-erate for the observation sequence S to *update* the emission and transition probabilities in $G(V, A)$. The parameter update procedure maximizes the similarity score of S in $G(V, A)$. This procedure updates the parameters as shown in Equations 3 and 4. The special $[S[t] = X]$ notation in Equation 4 is a conditional variable such that the variable returns 1 if the character X matches with the character $[S[t]]$, and returns 0 otherwise.

$$\alpha_{ij}^* = \frac{\sum_{t=1}^{n_S-1} \alpha_{ij}e_{S[t+1]}(v_j)F_t(i)B_{t+1}(j)}{\sum_{t=1}^{n_S-1} \sum_{x \in V} \alpha_{ix}e_{S[t+1]}(v_x)F_t(i)B_{t+1}(x)} \quad \forall \alpha_{ij} \in A \quad (3)$$

$$e_X^*(v_i) = \frac{\sum_{t=1}^{n_S} F_t(i)B_t(i)[S[t] = X]}{\sum_{t=1}^{n_S} F_t(i)B_t(i)} \quad \forall X \in \Sigma, \forall i \in V \quad (4)$$

2.3. Use Cases for Profile HMMs

2.3.1. Error Correction.
The goal of error correction is to locate the erroneous parts in DNA or genome sequences to re-place these parts with more reliable sequences [21, 42, 43, 120, 124, 139] to enable more accurate genome analysis (e.g., read mapping and genome assembly). Apollo [34] is a recent error correction algorithm that takes an assembly sequence and a set of reads as input to correct the errors in an assembly. Apollo constructs a pHMM graph for an assembly sequence to correct the errors in two steps: 1) training and 2) inference. First, to correct erroneous parts in an assembly, Apollo uses reads as observations to train the pHMM graph with the Baum-Welch algorithm. Second, Apollo uses the Viterbi algorithm [123] to identify the consensus sequence from the trained pHMM, which translates into the corrected assembly sequence. Apollo uses a slightly modified design of pHMMs to avoid certain lim-itations associated with traditional pHMMs when generating the consensus sequences [50, 69]. The modified design avoids loops in the insertion states and uses transitions to account for deletions instead of deletion states. These modifications allow the pHMM-based error correction applications [33, 34, 58] to construct more accurate consensus sequences from pHMMs.

2.3.2. Protein Family Search.
Classifying protein sequences into families is widely used to analyze the potential functions of the proteins of interest [15, 47, 81, 106, 119, 121]. Protein family search finds the family of the protein sequence in existing protein databases. A pHMM usually represents one protein family in the database to avoid searching for many individual sequences. The protein sequence can then be assigned to a protein family based on the similarity score of the protein when compared to a pHMM in a database. This approach is used to search protein sequences in the Pfam database [77], where the HMMER [29] software suite is used to build HMMs and assign query sequences to the best fitting Pfam family. Similar to the Pfam database, HMMER's protein family search tool is integrated into the European Bioinformatics Institute (EBI) website as a web tool. The same approach is also used in

several other important applications, such as classifying many genomic sequences into potential viral families [111].

2.3.3. Multiple Sequence Alignment.
Multiple sequence alignment (MSA) detects the differences between several biological sequences. Dynamic programming algorithms can optimally find differences between genomic sequences, but the complexity of these algorithms increases drastically with the number of sequences [49, 125]. To mitigate these computational problems, heuristics algorithms are used to obtain an approximate yet computationally efficient solution for multiple alignment of genomic sequences. PHMM-based approaches provide an efficient solution for MSA [22]. The pHMM approaches, such as *hmmalign* [29], assign likelihoods to all possible combinations of differences between sequences to calculate the pairwise similarity scores using forward and backward calculations or other optimization methods (e.g., particle swarm optimization [136]). PHMM-based MSA approaches are mainly useful to avoid making redundant comparisons as a sequence can be compared to a pHMM graph, similar to protein family search.

3. Motivation and Goal

3.1. Sources of Inefficiencies

To identify and understand the performance bottlenecks of state-of-the-art pHMM-based applications, we thoroughly analyze existing tools for the three use cases of pHMM: 1) Apollo [34] for error correction, 2) hmmsearch [29] for protein family search, and 3) hmmalign [29] for multiple sequence alignment (MSA). We make six key observations based on our profiling with Intel VTune [2] and gprof [38].

Observation 1: The Baum-Welch Algorithm is the Bottleneck. Figure 2 shows the percentage of the execution time of all three steps in the Baum-Welch algorithm for the three bioinformatics applications. We find that the Baum-Welch algorithm is overall the *performance bottleneck* for all three applications as the algorithm constitutes from 45.76% up to 98.57% of the total CPU execution time. Our profiling shows that these applications are mainly compute-bound. Forward and Backward calculations are the common steps in all three applications, whereas Parameter Updates step is executed only for error correction. This is because protein family search and MSA use the Forward and Backward calculations mainly for scoring between a sequence and a pHMM graph as a part of inference. We do *not* include the cost of training for these applications as it is executed either once or only a few times, such that the cost of training becomes insignificant compared to the frequently executed inference. However, the nature of error correction requires frequently performing both training and inference for every input sequence such that the cost of training is not negligible for this application. As a result, accelerating the entire Baum-Welch algorithm is key for accelerating the end-to-end performance of the applications.

Observation 2: SIMD-based tools provide suboptimal vectorization. The Baum-Welch algorithm requires frequent floating-point multiplications and additions. To resolve performance issues, hmmalign and hmmsearch use SIMD instruc-
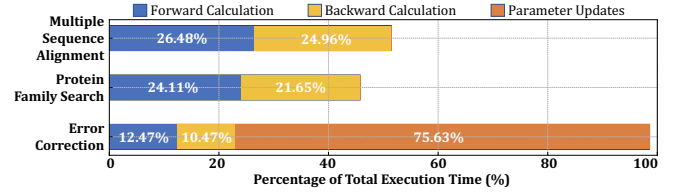

Figure 2: Percentage of the total execution time for the three steps of the Baum-Welch algorithm

tions. We observe that these tools have poor SIMD utilization due to poor port utilization and poor vector capacity usage (below 50%). These issues show that optimizations for floating-point operations provide limited computational benefits when executing the Baum-Welch algorithm.

Observation 3: Significant portion of floating-point operations is redundant. We observe that the same multiplications are repeatedly executed in the training step because certain floating-point values associated with transition and emission probabilities are mainly constant during training in error correction. Our profiling analysis with VTune shows that these redundant computations constitute around 22.7% of the overall execution time when using the Baum-Welch algorithm for training in error correction.

Observation 4: Filtering the states is costly despite its advantages. The Baum-Welch algorithm requires performing many operations for a large number of states. These operations are repeated in many iterations, and the number of states can grow in each iteration. There are several approaches to keep the state space (i.e., number of states) near-constant to improve the performance or the space efficiency of the Baum-Welch algorithm [33, 34, 39, 57, 76, 116, 127]. A simple approach is to pick the `best-n` states that provide the highest scores at each iteration while the rest of the states are ignored in the next iteration, known as filtering [33]. Figure 3 shows the relation between the filter size (i.e., the number of states picked as `best-n` states), runtime, and accuracy. Although the filtering approach is useful for reducing the runtime without significantly degrading the overall accuracy of the Baum-Welch algorithm, such an approach requires extra computations (e.g., sorting) to pick the `best-n` states. We find that such a filtering approach incurs non-negligible performance costs by constituting around 8.5% of the overall execution time in the training step.
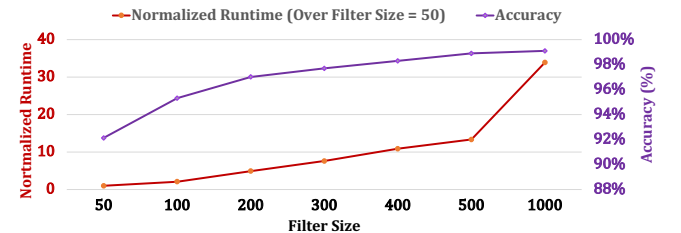

Figure 3: Effect of the filter size on the runtime and the accuracy of the Baum-Welch algorithm

Observation 5: HMM accelerators are suboptimal for accelerating pHMMs. Generic HMMs do not require any constraints on the connection between states (i.e., transitions)

and the number of states. PHMMs are a special case for HMMs where transitions are predefined, and the number of states is determined based on the sequence that a pHMM graph represents. These design choices in HMMs and pHMMs affect the data dependency pattern when executing the Baum-Welch Algorithm. Figure 4 shows an example of the data dependency patterns in pHMMs and HMMs when executing the Baum-Welch algorithm. We observe that although HMMs and pHMMs provide similar temporal localities (e.g., only the values from the previous iteration are used), pHMMs provide better spatial localities with their constrained design. This observation suggests that HMM-based accelerators cannot fully exploit the spatial localities that pHMMs provide as they are oblivious to the design of pHMMs.
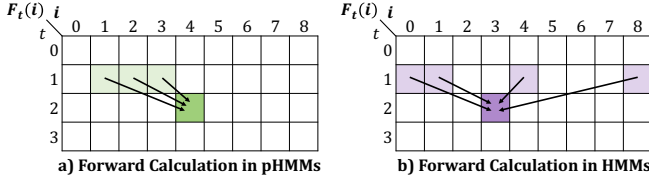

Figure 4: Data dependency in pHMMs and HMMs

Observation 6: Non-genomics pHMM-based applications suffer from the computational overhead of the Baum-Welch algorithm. Among many non-genomics pHMM-based implementations [4, 10, 25, 52, 66–68, 95, 97, 100–102], we analyze the available CPU implementation of a recent pattern-matching application that uses pHMMs [100]. Our initial analysis shows that almost the entire execution time (98%) of this application is spent on the Forward calculation, and it takes significantly longer times to execute a relatively small dataset compared to the bioinformatics applications.

Many applications use either the entire or parts of the Baum-Welch algorithm for training the probabilities of HMMs and pHMMs [4, 10, 13, 16, 20, 25, 29, 33, 34, 36, 52, 58, 97, 100, 102, 107, 113, 138]. However, the Baum-Welch algorithm can result in significant performance overheads on these applications due to computational inefficiencies. Solving the inefficiencies in the Baum-Welch algorithm are mainly important for services that frequently use these applications, such as the EBI website using HMMER for searching protein sequences in protein databases [70]. Based on the latest report in 2018, there have been more than 28 million HMMER queries on the EBI website within two years (2016-2017) [94]. On average, these queries execute parts of the Baum-Welch algorithm more than 38,000 times daily. Such frequent usage leads to significant waste in compute cycles and energy due to the inefficiencies in the Baum-Welch algorithm.

3.2. Goal

Based on our observations, we find that we need to have a specialized, flexible, high-performant, and energy-efficient design to ❶ support different pHMM designs with specialized compute units for each step in the Baum-Welch algorithm, ❷ eliminate redundant operations by enabling efficient reuse of the common multiplication products ❸ exploit spatio-temporal locality in an on-chip memory, and ❹ per-

form efficient filtering. Such a design has the potential to significantly reduce the computational and energy overhead of the applications that use the Baum-Welch algorithm in pHMMs. Unfortunately, software- or hardware-only solutions cannot solve these inefficiencies easily. There is a pressing need to develop a hardware-software co-designed and flexible acceleration framework for several pHMM-based applications that use the Baum-Welch algorithm.

In this work, our **goal** is to computational and energy overhead of the pHMMs-based applications that use the Baum-Welch algorithm with a flexible, high-performance, energy-efficient hardware-software co-designed acceleration framework. To this end, we propose ApHMM, the *first* highly flexible, high-performant, and energy-efficient accelerator that can support different pHMM designs to accelerate wide-range pHMM-based applications.

4. ApHMM Design

4.1. Microarchitecture Overview

ApHMM provides a **flexible**, high-performant, and energy-efficient hardware-software co-designed acceleration framework for calculating each step in the Baum-Welch algorithm. Figure 5 shows the main flow of ApHMM when executing the Baum-Welch algorithm for pHMMs. To exploit the massive parallelism that DNA and protein sequences provide, ApHMM processes many sequences in parallel using multiple copies of hardware units called *ApHMM Cores*. Each ApHMM Core aims to accelerate the Baum-Welch algorithm for pHMMs. An ApHMM Core contains two main blocks: 1) Control Block and 2) Compute Block. Control block provides efficient on- and off-chip synchronization and communication with CPU, DRAM, and on-chip L2 and L1 memory. Compute block efficiently and effectively performs each step in the Baum-Welch algorithm: 1) Forward calculation, 2) Backward calculation, and 3) Parameter Updates with respect to their corresponding equations in Section 2.2.
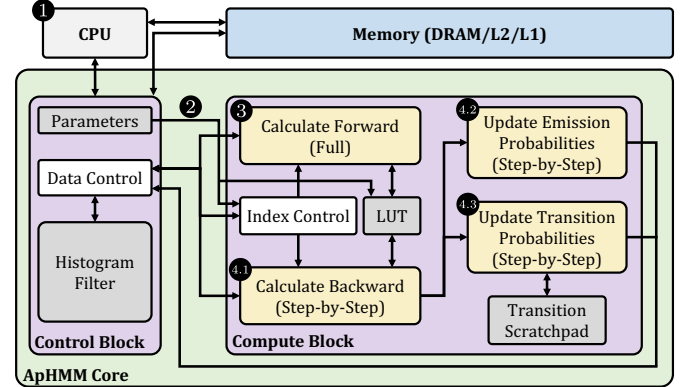

Figure 5: Overview of ApHMM

ApHMM starts when the CPU loads necessary data to memory and sends the parameters to ApHMM ❶. ApHMM uses the parameters to decide on the pHMM design (i.e., either traditional pHMM design or modified design for error correction) and steps to execute in the Baum-Welch algorithm. The parameters related to design are sent to Compute Block so that each

Compute Block can efficiently make proper state connections ❷. For each character in the input sequence that we aim to calculate the similarity score, Compute Block performs 1) Forward, 2) Backward, 3) and Parameter Updates steps. ApHMM enables disabling the calculation of Backward and Parameter Updates steps if they are not needed for an application. ApHMM iterates over the entire input sequence to fully perform the Forward calculation with respect to Equation 1 ❸. ApHMM then re-iterates each character on the input sequence character-by-character to perform the Backward calculations for each timestamp t with respect to Equation 2 (i.e., step-by-step) ❹.❶. ApHMM updates emission ❹.❷ and transition probabilities ❹.❸ as the Backward values are calculated in each timestamp.

4.2. Control Block

Control Section is responsible for managing the input and output flow of the compute section efficiently and correctly by issuing both memory requests and proper commands to Compute Block to configure for the next set of operations (e.g., the forward calculation for the next character of sequence S). Figure 5 shows three main units in Control Block.

Parameters. Control Block contains the parameters of pHMM and the Baum-Welch algorithm. These parameters define 1) pHMM design (i.e., either the traditional design or modified design for error correction) and 2) steps to execute in the Baum-Welch algorithm as ApHMM allows disabling the calculation of Backward or Parameter Updates steps.

Data Control. To ensure the correct, efficient, and synchronized data flow, ApHMM uses Data Control to 1) arbitrate among the read and write clients and 2) pipeline the read and write requests to the memory and other units in the accelerator (e.g., Histogram Filter). Data control is the main memory management unit for issuing a read request to L1 memory to obtain 1) each input sequence S, 2) corresponding pHMM graph (i.e., $G(V, A)$), 3) corresponding parameters and coefficients from the previous *timestamp* (e.g., Forward coefficients from timestamp $t - 1$ as shown in Equation 1). Data Control collects and controls the write requests from various clients to ensure data is synchronized.

Histogram Filter. The filtering approach is useful for eliminating negligible states from Forward and Backward calculations without significantly degrading the accuracy (Section 3). The **challenge** for truthfully implementing a simple filtering mechanism is to perform sorting in hardware, which is challenging to implement efficiently. Our **key idea** is to replace the sorting mechanism with a histogram-based filter to enable placing the values into different bins based on their values. This provides quick and approximate identification of non-negligible states (i.e., states with best values until the filter is full) based on the bins they are located. To enable such a binning mechanism, we employ a **flexible** *histogram-based* filtering mechanism in the ApHMM on-chip memory.

Figure 6 shows the overall structure of our Histogram Filter. Our filtering places the states into bins that correspond to a memory block based on their Forward or Backward values

from the current timestamp of the execution. The Histogram Filter divides the entire range of single-precision floating-point numbers into 16 equal parts (e.g., the range between two parts is $4.25E^{+37}$), where each bin corresponds to a range of predefined threshold values. We empirically chose to use 16 blocks to ensure our filtering mechanism achieves the same minimum accuracy when the filter size is 500 in Figure 3. The addresses of the states are assigned such that all the states in between the same two threshold values fall into the same memory space block. Such an addressing mechanism enables ApHMM to efficiently discard states that fall under the chosen threshold value in the next timestamp as their addresses are already known without requiring sorting. To build a **flexible** framework for many applications, the microarchitecture is configurable to vary these threshold values based on the application and the average sequence length.
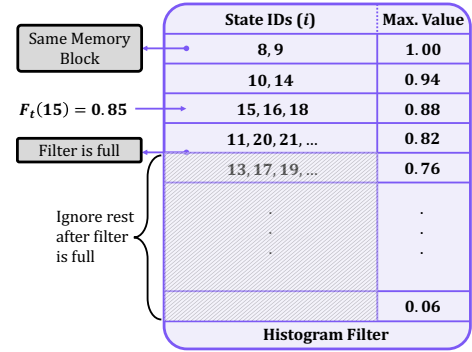

	State IDs (i)	Max. Value
Same Memory Block →	8, 9	1.00
	10, 14	0.94
$F_t(15) = 0.85$ →	15, 16, 18	0.88
Filter is full →	11, 20, 21, ...	0.82
	13, 17, 19, ...	0.76
Ignore rest after filter is full	. . .	. . .
		0.06
	Histogram Filter	

Figure 6: Overall structure of a Histogram Filter

ApHMM allows disabling the filtering mechanism if the application does not require a filter operation to achieve more optimal computations. Figure 7 shows the trade-off when using ApHMM with and without filter with sequences of different lengths. We observe that enabling the filtering mechanism provides significantly higher performance.
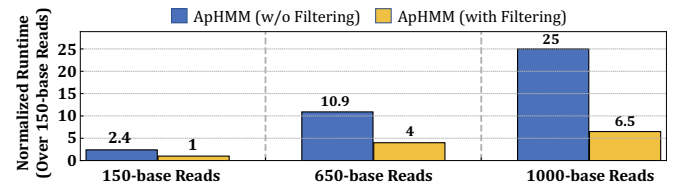

Figure 7: Effect of the Histogram Filter approach in ApHMM for different sequence lengths

4.3. Compute Block

Figure 8 shows the overall structure of a Compute Block. Compute Block is responsible for performing core compute operations of each step in the Baum-Welch algorithm (Figure 5) based on the configuration set by the Control Block via Index Control ❶. A Compute Block contains two major units: 1) a unit for calculating Forward (Equation 1) and Backward (Equation 2) values ❷ and updating transition probabilities (Equation 3) ❸.❶, and 2) a unit for updating the emission probabilities (Equation 4) ❸.❷. Each unit performs the corresponding calculations in the Baum-Welch algorithm.
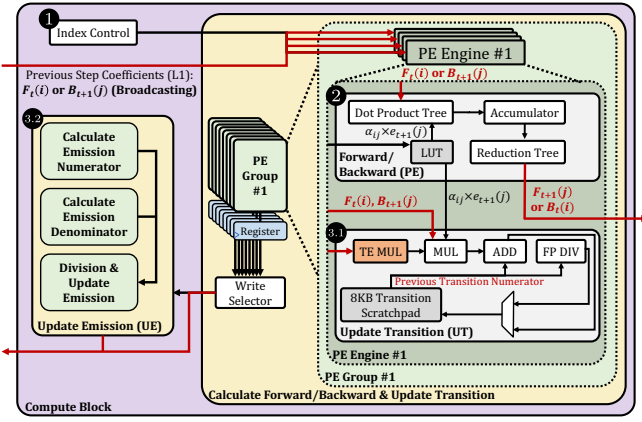
Figure 8: Overview of a Compute Block. Red arrows show on- and off-chip memory requests.

4.3.1. Forward and Backward Calculations. Our goal is to calculate the forward and backward values for all states in a pHMM graph $G(V, A)$, as shown in Equations 1 and 2, respectively. To calculate the Forward or Backward value of a state i at a timestamp t, ApHMM uses Processing Engines (PEs). Since pHMMs may require processing hundreds to thousands of states to process at a time, ApHMM includes many PEs and groups them PE Groups. Each PE is responsible for calculating the Forward and Backward values of a state v_i per timestamp t. Our **key challenge** is to balance the utilization of the compute units with available memory bandwidth. We address and discuss this trade-off between the number of PEs and memory bandwidth in Section 4.6. To achieve efficient calculation of the Forward and Backward values, PE performs two main operations.

First, PE uses the parallel four lanes in *Dot Product Tree* and *Accumulator* to perform multiple multiply and accumulation operations in parallel, where the final summation is calculated in the Reduction Tree. This design enables efficient multiplication and summation of values from previous timestamps (i.e., $F_{t-1}(j)$ or $B_{t+1}(j)$). Second, to avoid redundant multiplications of transition and emission probabilities, **the key idea** in PEs is to efficiently enable the reuse of the products of these common multiplications. To achieve this, our key mechanism stores these common products in lookup tables (LUTs) in each PE while enabling efficient retrievals of the common products. We store these products as these values can be preset (i.e., fixed) before the training step starts and frequently used during training while causing high computational overheads.

Our **key challenge** is to design minimal but effective LUTs to avoid area and energy overheads associated with LUTs without compromising the computational efficiency LUTs provide. To this end, we analyze error correction, protein family search, and multiple sequence alignment implementations. We observe that 1) redundant multiplications are frequent only during training and 2) the alphabet size of the biological sequence significantly determines the number of common products (i.e., 4 in DNA and 20 in proteins). Since error correction is mainly bottlenecked during the training step, we focus on the DNA alphabet and the pHMM design that error correction uses. We

identify that each state uses 1) at most 4 different emission probabilities (i.e., DNA letters) and 2) on average 7 different transitions. This results in 28 different combinations of emission and transition probabilities. To enable slightly better flexibility, we assume 9 different transitions and include 36 entries in LUTs.

The **key benefit** is LUTs provide ApHMM with a bandwidth reduction of up to 66% per PE while avoiding redundant computations. ApHMM is **flexible** such that it enables disabling the use of LUTs and instead performing the actual multiplication of transition and emission probabilities with the shared `TE MUL` unit in Figure 8.

4.3.2. Updating the Transition Probabilities. Our goal is to update the transition probabilities of all the states, as shown in Equation 3. To achieve this, we design the *Update Transition (UT)* compute unit and tightly couple it with PEs, as shown in Figure 8. Each UT efficiently calculates the denominator and numerator in Equation 3 for a state v_i. UTs include two key mechanisms.

First, to enable efficient broadcasting of common values between Backward calculation and Parameter Updates steps, ApHMM connects PEs with UTs for updating transitions. Each PE in a PE Group is broadcasted with the *same* previously calculated $F_t(i)$ or $B_{t+1}(j)$ values from the previous timestamp for calculating the $F_{t+1}(j)$ or $B_t(i)$ values, respectively. Incoming red arrows in Figure 8 show the flow of these values in PEs and UTs. This **key design choice** exploits the broadcast opportunities available within the common multiplications in the Baum-Welch equations. ApHMM Cores are designed to directly consume the broadcasted Backward values in multiple steps of the Baum-Welch algorithm in parallel to reduce the bandwidth and storage requirements. We exploit the broadcasting opportunities because we observe that Backward values do not need to be fully computed, and they can be consumed as they are broadcasted in the current timestamp. We update Emission and Transition probabilities step-by-step as Backward values are calculated, which is a hardware-software optimization that we call partial compute approach. **The key benefits** of our broadcasting and partial compute approach are 1) decoupling hardware scaling from bandwidth requirements and 2) reducing the bandwidth requirement by $4\times$ (i.e., 32 bits/cycle instead of 128 bits/cycle).

Second, to exploit the spatio-temporal locality in pHMMs, we utilize on-chip memory in UTs with memoization techniques that allow us to store the recent transition calculations. We observe from Equation 3 that transition update is calculated using the values of states connected to each other. Since the connections are predefined and provide spatial locality (Figure 4), our **key idea** is to memoize the calculation of all the numerators from the same i to different states by storing these numerators in the same memory space. This enables us to process the same state i in different timestamps within the same PE Engine to reduce the data movement overhead within ApHMM. To this end, we use an 8KB on-chip memory (Transition Scratchpad) to store and reuse the result of the numerator

of Equation 3. Since we store the numerators that contribute to all the transitions of a state *i* within the same memory space, we perform the final division in Equation 3 by using the values in the Transition Scratchpad. We use an 8KB memory as this enables us to store 256 different numerators from any state *i* to any other state *j*. We observe that pHMMs have 3-12 distinct transitions per state. Thus, 8KB storage enables us to operate on at least 20 different states within the same PE. **The memoization technique allows** 1) skipping redundant data movement and 2) reducing the bandwidth requirement by 2× per UT.

4.3.3. Updating the Emission Probabilities. Our goal is to update the emission probabilities of all the states, as shown in Equation 4. To achieve this, we use the *Update Emission (UE)* unit, as shown in Figure 8, which includes three smaller units: 1) Calculate Emission Numerator, 2) Calculate Emission Denominator, and 3) Division & Update Emission. UE performs the numerator and denominator computations in parallel as they are independent of each other, which includes a summation of the products $F_t(i)B_t(i)$. These $F_t(i)$ and $B_t(i)$ values are used to update *both* the transition and emission probabilities, as shown in Equation 3. To reduce redundant computations, our **key design** choice is to use the $F_t(i)$ and $B_t(i)$ values as broadcasted in the transition update step since these values are also used for updating the emission probabilities. Thus, we broadcast these values to UEs through *Write Selectors*, as shown in Figure 8.

The ApHMM Core writes and reads both the numerator and denominator values to L1 memory to update the emission probabilities. The results of the division operations and the posterior emission probabilities (i.e., $e_X^*(v_i)$ in Equation 4) are written back to L1 memory after processing each read sequence *S*. If we assume that the number of characters in an alphabet Σ is n_Σ (e.g., $n_\Sigma = 4$ for DNA letters), ApHMM stores n_Σ many different numerators for each state of the graph as emission probability may differ per character for each state. Our microarchitecture design is **flexible** such that it allows defining n_Σ as a parameter.

4.4. Data distribution and L1 Memory Layout

To implement genomic sequence execution in a limited cache environment, the sequences are divided into *chunks* of sequence lengths ranging from 150 to 1,000 characters to represent both sequencing reads and almost all protein sequences, as these protein sequences are mostly smaller than 1,000 characters [18]. For longer sequences, a sequence may be chunked into small pieces while preserving the relative order between sequences. An analysis of a similar software-level optimization reveals that chunking does not degrade the accuracy of the training and inference steps [34].

ApHMM partitions the L1 memory into four major sections: 1) chunked sequences that can be fed directly to the ApHMM Core, 2) a pHMM graph, 3) parameters to calculate the Baum-Welch algorithm, and 4) other temporary results generated by the ApHMM Core. Chunking the memories into blocks is not hard coded, and each section can use more space if needed.

ApHMM identifies the sections in memory blocks using additional 2 bits that label these four sections. Figure 9 shows the size of different Baum-Welch parameters that must be stored in memory based on the sequence length. It also captures the details for storing the data efficiently across the memory hierarchy. Since the entire genomic data set is traditionally large, it is typically stored in DRAM, and only smaller subsets of the entire data are fetched to the L2 and L1 memory. Similarly, ApHMM stores the entire forward values in DRAM and fetches them into L2 and L1 memory when needed. ApHMM uses the L1 memory of 128KB to support a larger spectrum of sequence lengths ranging between 150-1000 characters. We show the data distribution in L1 memory in Figure 9. Our key observation from the data distribution is that the size of Baum-Welch parameters grows as the sequence length increases. Thus, the number of sequences that L1 memory can hold reduces with increased read length. This does not cause frequent data load from DRAM or the L2 memory as longer sequences occupy the ApHMM Core usually for a longer duration, which compensates for the less number of read sequences stored in L1.
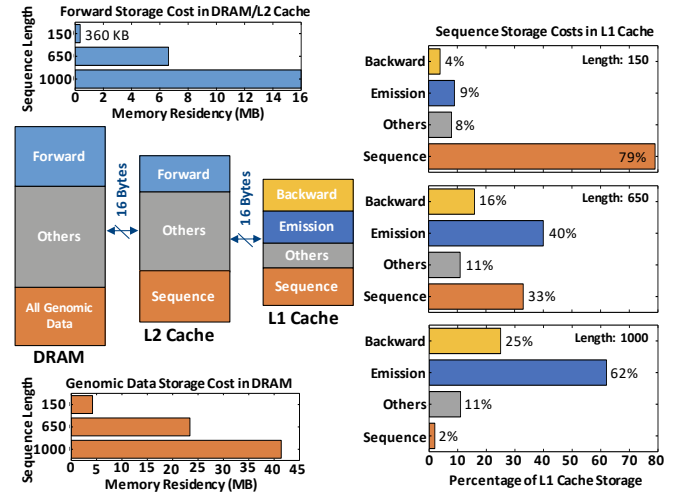

Figure 9: Data distribution across memory hierarchy.

4.5. System Mapping and Execution Flow

We show a system-level scale-up version of the ApHMM Core in Figure 10. ApHMM uses the L2-DMA table to load the data into the L2 memory and the L1-DMA table to write the corresponding data into the L1 memory per ApHMM Core according to the data distribution, as described in Section 4.4. ApHMM enables Probs-DMA to load the transition probabilities from DRAM to the local memory when the LUTs are not utilized, as discussed in Section 4.3.1. In such a scenario, local memory inside the PE Engine is loaded with appropriate transition probability data to perform the multiplications without using LUTs.

We present the execution flow of the system with multi-ApHMM Core in Figure 11. The operation starts with the host loading the data into DRAM and issuing DMA across various memory hierarchies through a global event control. Each ApHMM Core can start asynchronously, and near the completion of all reads from L1, hardware sets a flag for fetching
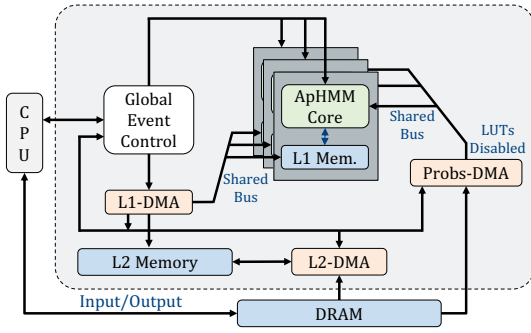
Figure 10: System integration of the ApHMM Core.

the next set of sequences from L2. Similarly, a counter-based signaling tells L2 to fetch the next set of sequences from DRAM. Once all reads are issued, ApHMM sends a completion signal and releases the control back to the host.
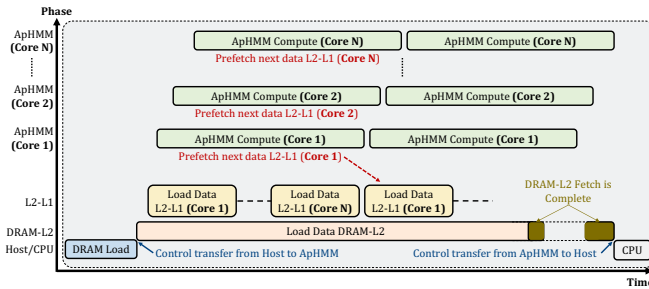

Figure 11: Control and execution flow of ApHMM Cores.

4.6. Hardware Configuration Choice

Our goal is to identify the ideal number of memory ports and processing elements (PE) for better scaling ApHMM with many cores. We identify the number of memory ports and their dependency on the hardware scaling in four steps. First, ApHMM requires one input memory port for reading the input sequence to update the probabilities in a pHMM graph. Second, updating the transition probabilities requires 3 memory ports: 1) reading the forward value from L1, 2) reading the transition and 3) emission probabilities if using the LUTs is disabled (Section 4.3.1). Since these ports are shared across each PE Engine, the number of PEs and memory bandwidth per port determines the utilization of these memory ports. Third, ApHMM requires 4 memory ports to update the emission probabilities for 1) calculating the numerator and 2) denominator in Equation 4, 3) reading the forward from Write Selectors, and 4) writing the output. These memory ports are *independent* of the impact of the number of PEs in a single ApHMM Core. Fourth, ApHMM does not require additional memory ports for each step in the Baum-Welch algorithm as a result of the broadcasting feature of ApHMM (Section 4.3.2). Instead, computing these steps depends on the 1) memory bandwidth per port, which determines the number of multiplications and accumulations in parallel in a PE, and 2) number of processing engines (PEs). We conclude that the overall requirement for the ApHMM Core is 8 memory ports with the same bandwidth per port.

In Figure 12(a), we show the acceleration speedup while scaling ApHMM with the number of PEs and bandwidth per

memory port, where we keep the number of memory ports fixed to 8. Based on Figure 12(a), we observe that a linear trend of increase in acceleration is possible until the number of PEs reaches 64, where the rate of acceleration starts reducing. We explore the reason for such a trend in Figure 12(b). We find that the acceleration on the transition step starts settling down as the number of PEs grows due to memory port limitation that reduces parallel data read from memory per PE, eventually resulting in the underutilization of resources. We conclude that the acceleration trend we observe in Figure 12(a) is mainly due to the scaling impact on the forward and backward calculation when the number of PEs is greater than 64 where 8 memory ports start becoming the bottleneck.

Choosing the memory bandwidth affects the number of PE Groups and PE Engines while keeping the number of PEs constant. Although our hardware can scale for higher bandwidth, we choose 16 Bytes/cycle, which results in 4 PE Engines (128 bit/32 FP32), and 16 PE Groups (64PEs/4 = 16). This configuration enables us to choose a smaller Transition Scratchpad with the increased parallelism across PE Engines without significantly compromising performance.
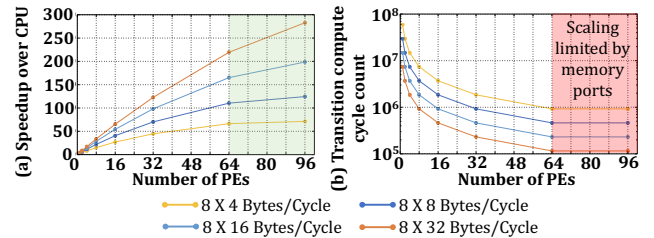

Figure 12: (a) Acceleration scaling with number of processing element. (b) Transition probability compute cycle acceleration with the increased number of PEs.

4.6.1. Number of ApHMM Cores. We show our methodology for choosing the ideal number of ApHMM-cores for accelerating the applications. Figure 13 shows the speedup of three bioinformatics applications when using single, 2, 4, and 8 ApHMM Cores. We incorporate the estimated off- and on-chip data movement overhead in our analysis. We observe that using 4 ApHMM-cores provides the best speedup overall. This is because the applications provide smaller rooms for acceleration, and the data movement overhead starts becoming the bottleneck as we increase the number of cores. This observation suggests that there is still room for improving the performance of ApHMM by placing ApHMM inside or near the memory (e.g., high-bandwidth memories) to eliminate these data movement overheads. We use 4-core ApHMM to achieve the best performance.
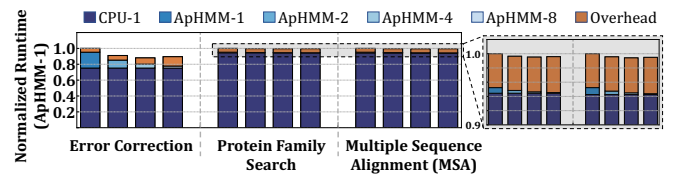

Figure 13: Normalized runtimes of multi-core ApHMM compared to the single-core ApHMM (ApHMM-1).

5. Evaluation

We evaluate our acceleration framework, ApHMM, for three use cases: 1) error correction, 2) protein family search and 3) multiple sequence alignment (MSA). We compare our results with the CPU, GPU, and FPGA implementations of the use cases.

5.1. Evaluation Methodology

We use the configurations shown in Table 1 to implement the ApHMM design described in Section 4 in SystemVerilog. We carry out synthesis using Synopsys Design Compiler [1] in a typical 28nm process technology node at 1GHz clock frequency with tightly integrated on-chip memory (1GHz) to extract the logic area and power numbers. We develop an analytical model to extract performance and area numbers for a scale-up configuration of ApHMM. We use 4 ApHMM cores in our evaluation (Section 4.6.1). We account for 5% extra cycles to compensate for arbitrating the across memory ports. These extra cycles estimate the cycles for synchronously loading data from DRAM to L2 memory of a single ApHMM core and asynchronous accesses to DRAM when more data needs to be from DRAM for a core (e.g., Forward calculation may not fit the L2 memory).

Table 1: Microarchitecture Configuration

Memory	Memory BW (Bytes/cycle): 16, Memory Ports (#): 8
	L1 Cache Size: 128KB
Processing	PEs (#): 64, Multipliers per PE (#): 4, Adders per PE (#): 4
Engine	Memory per PE: 8, Update Transitions (#): 64, Update Emissions (#): 4

We use the CUDA library [88] (version 11.6) to provide a GPU implementation of the software optimizations described in Section 4 for executing the Baum-Welch algorithm. Our GPU implementation, **ApHMM-GPU**, uses the pHMM design designed for error correction, implements LUTs (Section 4.3.1) as a shared memory, and uses buffers to arbitrate between current and previous Forward/Backward calculations to reflect the software optimizations of ApHMM in GPUs. We integrate our GPU implementation with a pHMM-based error correction tool, Apollo [34], to evaluate the GPU implementation. Our GPU implementation is the *first* GPU implementation of the Baum-Welch algorithm for profile Hidden Markov models.

We use gprof [38] to profile the baseline CPU implementations of the use cases on the AMD EPYC 7742 processor (2.26GHz, 7nm process) with single- and multi-threaded settings. We use the CUDA library and *nvidia-smi* to capture the runtime and power usage of ApHMM-GPU on NVIDIA A100 and NVIDIA Titan V GPUs, respectively.

We compare ApHMM with the CPU, GPU, and FPGA implementations of the Baum-Welch algorithm and use cases in terms of execution time and energy consumption. To evaluate the Baum-Welch algorithm, we execute the algorithm in Apollo [34] and calculate the average execution time and energy consumption of a single execution of the Baum-Welch algorithm. To evaluate the end-to-end execution time and energy consumption of error correction, protein family search, and multiple sequence alignment, we use Apollo [34], hmmsearch [29], and hmmalign [29]. We replace their implementation of the Baum-Welch algorithm with ApHMM when collecting the results of the end-to-end executions of the use cases accelerated using ApHMM. When available, we compare the use cases that we accelerate using ApHMM to their corresponding CPU, GPU, and FPGA implementations. For the GPU implementations, we use both ApHMM-GPU and HMM_cuda [134]. For the FPGA implementation, we use the FPGA Divide and Conquer (D&C) accelerator proposed for the Baum-Welch algorithm [93]. When evaluating the FPGA accelerator, we ignore the data movement overhead and estimate the acceleration based on the speedup as provided by the earlier work.

5.1.1. Data Set. To evaluate the error correction use case, we prepare the input data that Apollo requires: 1) assembly and 2) read mapping to the assembly. To construct the assembly and map reads to the assembly, we use reads from a real sample that includes overall 163,482 reads of Escherichia coli (E.coli) genome sequenced using PacBio sequencing technology. The accession code of this sample is SAMN06173305. Out of 163,482 reads, we randomly select 10,000 sequencing reads for our evaluation. We use minimap2 [62] and miniasm [61] to 1) find *overlapping reads* and 2) construct the assembly from these overlapping reads, respectively. To find the read mappings to the assembly, we use minimap2 to map the same reads to the assembly that we generate using these reads. We provide these inputs to Apollo for correcting errors in the assembly we construct.

To evaluate the protein family search, we use the protein sequences from a commonly studied protein family, Mitochondrial carrier (PF00153), which includes 214,393 sequences with an average length of 94.2. We use these sequences to search for similar protein families from the entire Pfam database [77] that includes 19,632 pHMMs. To achieve this, the hmmsearch [29] tool performs the Forward and Backward calculations to find similarities between pHMMs and sequences.

To evaluate multiple sequence alignment, we use 1,140,478 protein sequences from protein families Mitochondrial carrier (PF00153), Zinc finger (PF00096), bacterial binding protein-dependent transport systems (PF00528), and ATP-binding cassette transporter (PF00005). We align these sequences to the pHMM graph of the Mitochondrial carrier protein family. To achieve this, the hmmalign [29] tool performs the Forward and Backward calculations to find similarities between a single pHMM graph and sequences.

5.2. Area and Power

Table 2 shows the area breakup of the major modules in ApHMM. For the area overhead, we find that the Update Transition (UT) units take up most of the total area (77.98%). This is mainly because UTs consist of several complex units, such as a multiplexer, division pipeline, and local memory. For the power consumption, Control Block and PEs contribute to almost the entire power consumption (86%) due to the frequent memory accesses these blocks make. Overall, the ApHMM Core incurs an area overhead of 6.5mm^2 in 28nm with a power cost of 0.509W.

Table 2: Area and Power breakdown of ApHMM

Module Name	Area (mm^2)	Power (mW)
Control Block	0.011	134.4
64 Processing Engines (PEs)	1.333	304.2
64 Update Transitions (UTs)	5.097	0.8
4 Update Emissions (UEs)	0.094	70.4
Overall	6.536	509.8
128KB L1-Memory	0.632	100

5.3. Accelerating the Baum-Welch Algorithm

Figure 14 shows the performance and energy improvements of ApHMM for executing the Baum-Welch algorithm. Based on these results, we make six *key* observations. First, we observe that ApHMM is 15.55×- 260.03×, 1.83×-5.34×, and 27.97× faster than the CPU, GPU, and FPGA implementations of the Baum-Welch algorithm, respectively. Second, ApHMM reduces the energy consumption for calculating the Baum-Welch algorithm by 2474.09× and 896.70×-2622.94× compared to the single-threaded CPU implementation and GPU implementations, respectively. These speedups and reduction in energy consumption show the combined benefits of our software-hardware optimizations. Third, the parameter update step is the most time-consuming step for the CPU and the GPU implementations, while ApHMM takes the most time in the forward calculation step. The reason for such a trend shift is that ApHMM reads and writes to L2 Cache and DRAM more frequently during the forward calculation than the other steps, as ApHMM requires the forward calculation step to be fully completed and stored in the memory before moving to the next steps as we explain in Section 4.3.2. Fourth, we observe that ApHMM-GPU performs better than HMM_cuda by 2.02× on average. HMM_cuda executes the Baum-Welch algorithm on any type of hidden Markov model without a special focus on pHMMs. As we develop our optimizations based on pHMMs, ApHMM-GPU can take advantage of these optimizations for more efficient execution. Fifth, both ApHMM-GPU and HMM_cuda provide better performance for the Forward calculation than ApHMM. We believe that the GPU implementations are a better candidate for applications that execute only the Forward calculations as ApHMM targets providing the best performance for the complete Baum-Welch algorithm. Sixth, the GPU implementations provide a limited speedup over the multi-threaded CPU implementations mainly because of frequent access to the host for synchronization and sorting (e.g., the filtering mechanism). These required accesses from GPU to host can be minimized with a specialized hardware design, as we propose in ApHMM for performing the filtering mechanism. We conclude that ApHMM provides substantial improvements, especially when we combine speedups and energy reductions for executing the complete Baum-Welch algorithm compared to the CPU and GPU implementations, which makes it a better candidate to accelerate the applications that use the Baum-Welch algorithm than the CPU, GPU, and FPGA implementations.
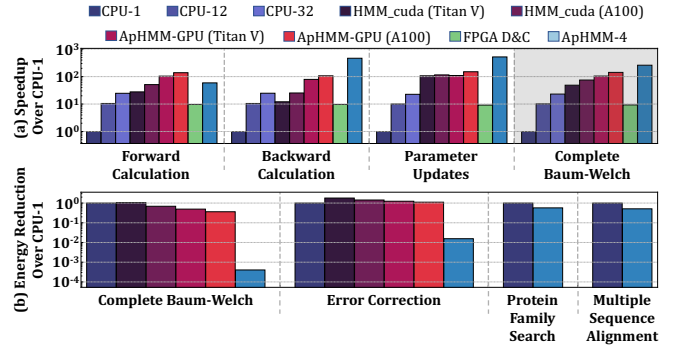

Figure 14: (a) Normalized speedups of each step in the Baum-Welch algorithm over single-threaded CPU (CPU-1). (b) Energy reductions compared to the CPU-1 implementation of the Baum-Welch algorithm and three pHMM-based applications.

5.4. Use Case 1: Error Correction

Figures 15 and 14 show the end-to-end execution time and energy reduction results for error correction, respectively. We make four key observations. First, we observe that ApHMM is 2.66×- 59.94×, 1.29×- 2.09×, and 7.21× faster than the CPU, GPU, and FPGA implementations of Apollo, respectively. Second, ApHMM reduces the energy consumption by 64.24× and 71.28×- 115.46× compared to the single-threaded CPU and GPU implementations. These two observations are in line with the observations we make in Section 5.3 as well as the motivation results we describe in Section 3: Apollo is mainly bounded by the Baum-Welch algorithm, and ApHMM accelerates the Baum-Welch algorithm significantly, providing significant performance improvements and energy reductions for error correction. We conclude that ApHMM *significantly* improves the energy efficiency and performance of the error correction mainly because the Baum-Welch algorithm constitutes a large portion of the entire use case.
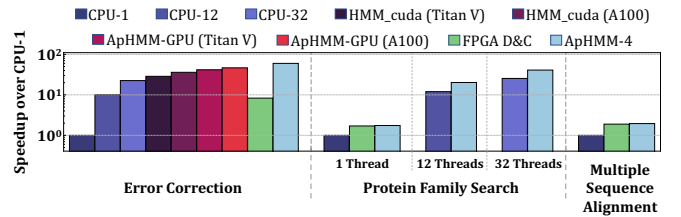

Figure 15: Speedups over the single-threaded CPU implementations. In protein family search, we compare ApHMM with each CPU thread separately.

5.5. Use Case 2: Protein Family Search

Our goal is to evaluate the performance and energy consumption of ApHMM for the protein family search use case, as shown in Figures 15 and 14, respectively. We make three key observations. First, we observe that ApHMM provides speedup by 1.61×- 1.75×, and 1.03× compared to the CPU and FPGA implementations. Second, ApHMM is 1.75× more energy efficient than the single-threaded CPU implementation. The speedup ratio that ApHMM provides is lower in protein family search than error correction because 1) ApHMM accelerates a smaller portion of protein family search (45.76%) than error correction (98.57%), and 2) the protein alphabet size

(20) is much larger than the DNA alphabet size (4), which increases the DRAM access overhead of ApHMM by 12.5%. Due to the smaller portion that ApHMM accelerates and increased memory accesses, it is expected that ApHMM provides lower performance improvements and energy reductions compared to the error correction use case. Third, ApHMM can provide better speedup compared to the multi-threaded CPU as a large portion of the parts that ApHMM does not accelerate can still be executed in parallel using the same amount of threads, as shown in Figure 15. We conclude that ApHMM improves the performance and energy efficiency for protein family search, while there is a smaller room for acceleration compared to error correction.

5.6. Use Case 3: Multiple Sequence Alignment

Our goal is to evaluate the ApHMM's end-to-end performance and energy consumption for multiple sequence alignment (MSA), as shown in Figures 15 and 14, respectively. We make three key observations. First, we observe that ApHMM performs $1.95\times$ and $1.03\times$ better than the CPU and FPGA implementations, while ApHMM is $1.96\times$ more energy efficient than the CPU implementation of MSA. We note that the hmmalign tool does not provide the multi-threaded CPU implementation for MSA. ApHMM provides better speedup for MSA than protein family search because MSA performs more forward and backward calculations (51.44%) than the protein search use case (45.76%), as shown in Figure 2. Third, ApHMM provides slightly better performance than the existing FPGA accelerator (FPGA D&C) in all applications, even though we ignore the data movement overhead of FPGA D&C, which suggests that ApHMM may perform much better than FPGA D&C in real systems. We conclude that ApHMM improves the performance and energy efficiency of the MSA use case better than protein family search.

6. Related Work

To our knowledge, this is the first work that provides a flexible and hardware-software co-designed acceleration framework to efficiently and effectively execute the complete Baum-Welch algorithm for pHMMs. In this section, we explain previous attempts to accelerate *HMMs*. Previous works [12, 24, 29, 44, 46, 48, 60, 89, 90, 93, 96, 98, 112, 126, 129, 130, 134] mainly focus on specific algorithms and designs of HMMs to accelerate the HMM-based applications. Several works [24, 46, 48, 89, 90, 96] propose FPGA- or GPU-based accelerators for pHMMs to accelerate a different algorithm used in the inference step for pHMMs. A group of previous works [44, 60, 98, 112] accelerates the Forward calculation based on the HMM designs different than pHMMs for FPGAs and supercomputers. HMM_cuda [134] uses GPUs to accelerate the Baum-Welch algorithm for any HMM design. ApHMM differs from all of these works as it accelerates the entire Baum-Welch algorithm on pHMMs for more optimized performance, while these works are oblivious to the pHMM design when accelerating the Baum-Welch algorithm.

A related design choice to pHMMs is Pair HMMs. Pair HMMs are useful for identifying differences between DNA and protein sequences. To identify differences, Pair HMMs use states to represent a certain scoring function (e.g., affine gap penalty) or variation type (i.e., insertion, deletion, mismatch, or match) by typically using only one state for each score or difference. This makes Pair HMMs a good candidate for generalizing pairwise sequence comparisons as they can compare pairs of sequences while being oblivious to any sequence. Unlike pHMMs, Pair HMMs are not built to represent sequences. Thus, Pair HMMs cannot 1) compare a sequence to a group of sequences and 2) perform error correction. Pair HMMs mainly target variant calling and sequence alignment problems in bioinformatics. There is a large body of work that accelerates Pair HMMs [12, 44, 60, 98, 126, 129, 130]. ApHMM differs from these works as its hardware-software co-design is optimized for pHMMs.

7. Conclusion

We propose ApHMM, the *first* hardware-software co-design framework that accelerates the execution of the entire Baum-Welch algorithm for pHMMs. ApHMM particularly accelerates the Baum-Welch algorithm as it is a common computational bottleneck for important bioinformatics applications. ApHMM proposes several hardware-software optimizations to efficiently and effectively execute the Baum-Welch algorithm for pHMMs. The hardware-software co-design of ApHMM provides significant performance improvements and energy reductions compared to CPU, GPU, and FPGAs, as ApHMM minimizes redundant computations and data movement overhead for executing the Baum-Welch algorithm. We hope that ApHMM enables further future work by accelerating the remaining steps used with pHMMs (e.g., Viterbi decoding) based on the optimizations we provide in ApHMM.

Acknowledgments

References

[1] "Tool from Synopsys, Design Compiler (Version L-2016.03-SP2)." https://www.synopsys.com

[2] "Intel Vtune Profiler," 2022. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[3] I. Ahmad *et al.*, "Open-vocabulary recognition of machine-printed Arabic text using hidden Markov models," *Pattern Recognition*, vol. 51, pp. 97–111, Mar. 2016.

[4] M. Ali *et al.*, "Profile Hidden Markov Model Malware Detection and API Call Obfuscation," in *Proceedings of the 8th International Conference on Information Systems Security and Privacy - Volume 1: ForSE,*. SciTePress, 2022, pp. 688–695, backup Publisher: INSTICC.

[5] M. Alser *et al.*, "Accelerating Genome Analysis: A Primer on an Ongoing Journey," *IEEE Micro*, vol. 40, no. 5, pp. 65–75, Oct. 2020.

[6] M. Alser *et al.*, "Going from molecules to genomic variations to scientific discovery: Intelligent algorithms and architectures for intelligent genome analysis," *arXiv*, May 2022.

[7] M. Alser *et al.*, "Technology dictates algorithms: recent developments in read alignment," *Genome Biology*, vol. 22, no. 1, p. 249, Aug. 2021.

[8] M. Alser *et al.*, "SneakySnake: a fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs," *Bioinformatics*, vol. 36, no. 22-23, pp. 5282–5290, Dec. 2020.

[9] S. Angizi *et al.*, "PIM-Aligner: A Processing-in-MRAM Platform for Biological Sequence Alignment," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 1265–1270.

[10] S. Attaluri *et al.*, "Profile hidden Markov models and metamorphic virus detection," *Journal in Computer Virology*, vol. 5, no. 2, pp. 151–169, 2009.

[11] P. Baldi *et al.*, "Hidden Markov models of biological primary sequence information." *Proceedings of the National Academy of Sciences*, vol. 91, no. 3, pp. 1059–1063, 1994.

[12] S. S. Banerjee *et al.*, "On accelerating pair-HMM computations in programmable hardware," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8.

[13] A. Bateman *et al.*, "The PFAM protein families database," *Nucleic Acids Research*, vol. 30, no. 1, pp. 276–280, 2002.

[14] L. E. Baum, "An inequality and associated maximization technique in statistical estimation of probabilistic functions of a Markov process," *Inequalities*, vol. 3, pp. 1–8, 1972.

[15] M. L. Bileschi *et al.*, "Using deep learning to annotate the protein universe," *Nature Biotechnology*, Feb. 2022.

[16] P. Boufounos *et al.*, "Basecalling using hidden Markov models," *Genomics, Signal Processing, and Statistics*, vol. 341, no. 1, pp. 23–36, Jan. 2004.

[17] Y. Boussemart *et al.*, "Comparing Learning Techniques for Hidden Markov Models of Human Supervisory Control Behavior," in *AIAA Infotech@Aerospace Conference*. Reston, Virigina: American Institute of Aeronautics and Astronautics, 2009.

[18] L. Brocchieri and S. Karlin, "Protein length in eukaryotic and prokaryotic proteomes," *Nucleic Acids Research*, vol. 33, no. 10, pp. 3390–3400, 2005.

[19] A. D. Calin, "Gesture Recognition on Kinect Time Series Data Using Dynamic Time Warping and Hidden Markov Models," in *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2016, pp. 264–271.

[20] P. Chen *et al.*, "Detecting critical state before phase transition of complex biological systems by hidden Markov model," *Bioinformatics*, vol. 32, no. 14, pp. 2143–2150, Jul. 2016.

[21] C.-S. Chin *et al.*, "Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data," *Nature Methods*, vol. 10, no. 6, pp. 563–569, Jun. 2013.

[22] B. Chowdhury and G. Garai, "A review on multiple sequence alignment from the perspective of genetic algorithm," *Genomics*, vol. 109, no. 5, pp. 419–431, Oct. 2017.

[23] N. Deo *et al.*, "In-vehicle Hand Gesture Recognition using Hidden Markov models," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, 2016, pp. 2179–2184.

[24] S. Derrien and P. Quinton, "Hardware Acceleration of HMMER on FPGAs," *Journal of Signal Processing Systems*, vol. 58, no. 1, p. 53, Oct. 2008.

[25] W. Ding *et al.*, "Skeleton-Based Human Action Recognition with Profile Hidden Markov Models," in *Computer Vision*, H. Zha *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 12–21.

[26] R. Durbin *et al.*, *Biological Sequence Analysis*, 1998.

[27] S. R. Eddy, "Profile hidden Markov models," *Bioinformatics*, vol. 14, no. 9, pp. 755–763, 1998.

[28] S. R. Eddy, "What is a hidden Markov model?" *Nature Biotechnology*, vol. 22, no. 10, pp. 1315–1316, Oct. 2004.

[29] S. R. Eddy, "Accelerated Profile HMM Searches," *PLoS Computational Biology*, vol. 7, no. 10, p. e1002195, 2011.

[30] R. C. Edgar and K. Sjolander, "COACH: profile-profile alignment of protein families using hidden Markov models," *Bioinformatics*, vol. 20, no. 8, pp. 1309–1318, 2004.

[31] R. C. Edgar and K. Sjölander, "SATCHMO: sequence alignment and tree construction using hidden Markov models," *Bioinformatics*, vol. 19, no. 11, pp. 1404–1411, Jul. 2003.

[32] R. D. Finn *et al.*, "The Pfam protein families database," *Nucleic Acids Research*, vol. 38, no. suppl_1, pp. D211–D222, Jan. 2010.

[33] C. Firtina *et al.*, "Hercules: a profile HMM-based hybrid error correction algorithm for long reads," *Nucleic Acids Research*, vol. 46, no. 21, pp. e125–e125, 2018.

[34] C. Firtina *et al.*, "Apollo: a sequencing-technology-independent, scalable and accurate assembly polishing algorithm," *Bioinformatics*, vol. 36, no. 12, pp. 3669–3679, 2020.

[35] C. Firtina *et al.*, "BLEND: A Fast, Memory-Efficient, and Accurate Mechanism to Find Fuzzy Seed Matches," *arXiv*, p. arXiv:2112.08687, 2021.

[36] T. Friedrich *et al.*, "Modelling interaction sites in protein domains with interaction profile hidden Markov models," *Bioinformatics*, vol. 22, no. 23, pp. 2851–2857, Dec. 2006.

[37] S. D. Goenka *et al.*, "SegAlign: A Scalable GPU-Based Whole Genome Aligner," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–13.

[38] S. L. Graham *et al.*, "Gprof: A Call Graph Execution Profiler," *SIGPLAN Not.*, vol. 39, no. 4, pp. 49–57, Apr. 2004.

[39] J. Grice *et al.*, "Reduced space sequence alignment," *Bioinformatics*, vol. 13, no. 1, pp. 45–53, Feb. 1997.

[40] M. Haid *et al.*, "Inertial-Based Gesture Recognition for Artificial Intelligent Cockpit Control using Hidden Markov Models," in *2019 IEEE International Conference on Consumer Electronics (ICCE)*, 2019, pp. 1–4.

[41] M. Hamidi *et al.*, "Interactive Voice Response Server Voice Network Administration Using Hidden Markov Model Speech Recognition System," in *2018 Second World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*, 2018, pp. 16–21.

[42] J. Hu *et al.*, "NextPolish: a fast and efficient genome polishing tool for long-read assembly," *Bioinformatics*, vol. 36, no. 7, pp. 2253–2255, Apr. 2020.

[43] N. Huang *et al.*, "NeuralPolish: a novel Nanopore polishing method based on alignment matrix construction and orthogonal Bi-GRU Networks," *Bioinformatics*, vol. 37, no. 19, pp. 3120–3127, Oct. 2021.

[44] S. Huang *et al.*, "Hardware Acceleration of the Pair-HMM Algorithm for DNA Variant Calling," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 275–284.

[45] A. Hubin, "An Adaptive Simulated Annealing EM Algorithm for Inference on Non-Homogeneous Hidden Markov Models," in *Proceedings of the International Conference on Artificial Intelligence, Information Processing and Cloud Computing*, ser. AIIPCC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[46] A. Ibrahim *et al.*, "Reconfigurable Hardware Accelerator for Profile Hidden Markov Models," *Arabian Journal for Science and Engineering*, vol. 41, no. 8, pp. 3267–3277, 2016.

[47] M. Jeffryes and A. Bateman, "Rapid identification of novel protein families using similarity searches," *F1000Research*, vol. 7, pp. ISCB Comm J–1975, Dec. 2018.

[48] H. Jiang *et al.*, "CUDAMPF++: A Proactive Resource Exhaustion Scheme for Accelerating Homologous Sequence Search on CUDA-Enabled GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2206–2222, 2018.

[49] W. Just, "Computational Complexity of Multiple Sequence Alignment with SP-Score," *Journal of Computational Biology*, vol. 8, no. 6, pp. 615–623, 2001.

[50] R. Y. Kahsay *et al.*, "Quasi-consensus-based comparison of profile hidden Markov models for protein sequences," *Bioinformatics*, vol. 21, no. 10, pp. 2287–2293, 2005.

[51] M. Kang *et al.*, "Opinion mining using ensemble text hidden Markov models for text classification," *Expert Systems with Applications*, vol. 94, pp. 218–227, Mar. 2018.

14

[52] I. Kazantzidis *et al.*, "Profile Hidden Markov Models for Foreground Object Modelling," in *2018 25th IEEE International Conference on Image Processing (ICIP)*, 2018, pp. 1628–1632.

[53] C. Kern *et al.*, "Predicting Interacting Residues Using Long-Distance Information and Novel Decoding in Hidden Markov Models," *IEEE Transactions on NanoBioscience*, vol. 12, no. 3, pp. 158–164, 2013.

[54] J. S. Kim *et al.*, "FastRemap: A Tool for Quickly Remapping Reads between Genome Assemblies," *arXiv*, Jan. 2022.

[55] J. S. Kim *et al.*, "AirLift: A Fast and Comprehensive Technique for Remapping Alignments between Reference Genomes," *bioRxiv*, p. 2021.02.16.431517, Jan. 2021.

[56] J. S. Kim *et al.*, "GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies," *BMC Genomics*, vol. 19, no. 2, p. 89, May 2018.

[57] B. Kirkpatrick and K. Kirkpatrick, "Optimal state-space reduction for pedigree hidden markov models," *arXiv*, p. arXiv:1202.2468, 2012.

[58] H. Lanyue *et al.*, "A Long read hybrid error correction algorithm based on segmented pHMM," in *2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, 2020, pp. 1501–1504.

[59] S. J. Lewis *et al.*, "Bayesian Monte Carlo estimation for profile hidden Markov models," *Mathematical and Computer Modelling*, vol. 47, no. 11, pp. 1198–1216, 2008.

[60] E. Li *et al.*, "Improved GPU Implementations of the Pair-HMM Forward Algorithm for DNA Sequence Alignment," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, 2021, pp. 299–306.

[61] H. Li, "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences," *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, Jul. 2016.

[62] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, Sep. 2018.

[63] L. Li *et al.*, "Hybrid Deep Neural Network–Hidden Markov Model (DNN-HMM) Based Speech Emotion Recognition," in *2013 Humaine Association Conference on Affective Computing and Intelligent Interaction*, 2013, pp. 312–317.

[64] K.-c. Liang *et al.*, "Bayesian Basecalling for DNA Sequence Analysis Using Hidden Markov Models," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 4, no. 3, pp. 430–440, 2007.

[65] D. V. Lindberg and D. Grana, "Petro-Elastic Log-Facies Classification Using the Expectation–Maximization Algorithm and Hidden Markov Models," *Mathematical Geosciences*, vol. 47, no. 6, pp. 719–752, Aug. 2015.

[66] F. Liu *et al.*, "Characterizing activity sequences using profile Hidden Markov Models," *Expert Systems with Applications*, vol. 42, no. 13, pp. 5705–5722, Aug. 2015.

[67] X. Liu *et al.*, "Adversarial attacks against profile HMM website fingerprinting detection model," *Cognitive Systems Research*, vol. 54, pp. 83–89, May 2019.

[68] Y. Liu *et al.*, "Who is the expert? Analyzing gaze data to predict expertise level in collaborative applications," in *2009 IEEE International Conference on Multimedia and Expo*, 2009, pp. 898–901.

[69] R. B. Lyngsø and C. N. S. Pedersen, "The consensus string problem and the complexity of comparing hidden Markov models," *Journal of Computer and System Sciences*, vol. 65, no. 3, pp. 545–569, 2002.

[70] F. Madeira *et al.*, "The EMBL-EBI search and sequence analysis tools APIs in 2019," *Nucleic Acids Research*, vol. 47, no. W1, pp. W636–W641, Jul. 2019.

[71] M. Madera, "Profile Comparer: a program for scoring and aligning profile hidden Markov models," *Bioinformatics*, vol. 24, no. 22, pp. 2630–2631, 2008.

[72] M. Madera and J. Gough, "A comparison of profile hidden Markov model procedures for remote homology detection," *Nucleic Acids Research*, vol. 30, no. 19, pp. 4321–4328, Oct. 2002.

[73] G. Malysa *et al.*, "Hidden Markov model-based gesture recognition with FMCW radar," in *2016 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, 2016, pp. 1017–1021.

[74] N. Mansouri Ghiasi *et al.*, "GenStore: A High-Performance in-Storage Processing System for Genome Sequence Analysis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 635–654.

[75] S. Mao *et al.*, "Revisiting Hidden Markov Models for Speech Emotion Recognition," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019, pp. 6715–6719.

[76] I. Miklós and I. M. Meyer, "A linear memory algorithm for Baum-Welch training," *BMC Bioinformatics*, vol. 6, no. 1, p. 231, Sep. 2005.

[77] J. Mistry *et al.*, "Pfam: The protein families database in 2021," *Nucleic Acids Research*, vol. 49, no. D1, pp. D412–D419, Jan. 2021.

[78] T. K. Moon, "The expectation-maximization algorithm," *IEEE Signal Processing Magazine*, vol. 13, no. 6, pp. 47–60, 1996.

[79] B. Mor *et al.*, "A Systematic Review of Hidden Markov Models and Their Applications," *Archives of Computational Methods in Engineering*, vol. 28, no. 3, pp. 1429–1448, May 2021.

[80] B. S. Moreira *et al.*, "An Acoustic Sensing Gesture Recognition System Design Based on a Hidden Markov Model," *Sensors*, vol. 20, no. 17, 2020.

[81] N. J. Mulder and R. Apweiler, "Tools and resources for identifying protein families, domains and motifs," *Genome Biology*, vol. 3, no. 1, p. reviews2001.1, Dec. 2001.

[82] S. Mulia *et al.*, "Profile HMM based Multiple Sequence Alignment for DNA Sequences," *INTERNATIONAL CONFERENCE ON MODELLING OPTIMIZATION AND COMPUTING*, vol. 38, pp. 1783–1787, Jan. 2012.

[83] M. K. Mustafa *et al.*, "A comparative review of dynamic neural networks and hidden Markov model methods for mobile on-device speech recognition," *Neural Computing and Applications*, vol. 31, no. 2, pp. 891–899, Feb. 2019.

[84] A. Nag *et al.*, "GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, pp. 334–346.

[85] V. Narasimhan *et al.*, "BCFtools/RoH: a hidden Markov model approach for detecting autozygosity from next-generation sequencing data," *Bioinformatics*, vol. 32, no. 11, pp. 1749–1751, Jun. 2016.

[86] Z. Nasim and S. Ghani, "Sentiment Analysis on Urdu Tweets Using Markov Chains," *SN Computer Science*, vol. 1, no. 5, p. 269, Aug. 2020.

[87] N. Nguyen-Duc-Thanh *et al.*, "Two-Stage Hidden Markov Model in Gesture Recognition for Human Robot Interaction," *International Journal of Advanced Robotic Systems*, vol. 9, no. 2, p. 39, Aug. 2012.

[88] J. Nickolls *et al.*, "Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?" *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[89] T. Oliver *et al.*, "High Performance Database Searching with HMMer on FPGAs," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–7.

[90] T. Oliver *et al.*, "Integrating FPGA acceleration into HMMer," *High-Performance Computational Biology*, vol. 34, no. 11, pp. 681–691, Nov. 2008.

[91] I. Patel and Y. S. Rao, "Speech Recognition Using Hidden Markov Model with MFCC-Subband Technique," in *2010 International Conference on Recent Trends in Information, Telecommunication and Computing*, 2010, pp. 168–172.

[92] J. Pei and N. V. Grishin, "PROMALS: towards accurate multiple sequence alignments of distantly related proteins," *Bioinformatics*, vol. 23, no. 7, pp. 802–808, Apr. 2007.

[93] M. Pietras and P. Klęsk, "FPGA implementation of logarithmic versions of Baum-Welch and Viterbi algorithms for reduced precision hidden Markov models," *Bulletin of the Polish Academy of Sciences: Technical Sciences*, vol. 65, no. No 6 (Special Section on Civil Engineering – Ongoing Technical Research. Part II), pp. 935–947, 2017.

[94] S. C. Potter *et al.*, "HMMER web server: 2018 update," *Nucleic Acids Research*, vol. 46, no. W1, pp. W200–W204, Jul. 2018.

[95] R. Pranamulia *et al.*, "Profile hidden Markov model for malware classification — usage of system call sequence for malware classification," in *2017 International Conference on Data and Software Engineering (ICoDSE)*, 2017, pp. 1–5.

[96] S. Quirem *et al.*, "CUDA acceleration of P7Viterbi algorithm in HMMER 3.0," in *30th IEEE International Performance Computing and Communications Conference*, 2011, pp. 1–2.

[97] S. Ravi *et al.*, "Behavior-based Malware analysis using profile hidden Markov models," in *2013 International Conference on Security and Cryptography (SECRYPT)*, 2013, pp. 1–12.

[98] S. Ren *et al.*, "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.

[99] V. Rezaei *et al.*, "Generalized Baum-Welch Algorithm Based on the Similarity between Sequences," *PLOS ONE*, vol. 8, no. 12, p. e80565, 2013.

[100] A. B. Riddell, "Reliable Editions from Unreliable Components: Estimating Ebooks from Print Editions Using Profile Hidden Markov Models," *arXiv*, p. arXiv:2204.01638, 2022.

[101] I. Saadi *et al.*, "A Framework to Identify Housing Location Patterns Using Profile Hidden Markov Models," *Advanced Science Letters*, vol. 22, no. 9, pp. 2117–2121, Sep. 2016.

[102] S. K. Sasidharan and C. Thomas, "ProDroid — An Android malware detection framework based on profile hidden Markov model," *Pervasive and Mobile Computing*, vol. 72, p. 101336, Apr. 2021.

[103] S. L. Scott, "Bayesian Methods for Hidden Markov Models," *Journal of the American Statistical Association*, vol. 97, no. 457, pp. 337–351, 2002.

[104] D. Senol Cali *et al.*, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 951–966.

[105] D. Senol Cali *et al.*, "SeGraM: A Universal Hardware Accelerator for Genomic Sequence-to-Graph and Sequence-to-Sequence Mapping," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 638–655.

[106] S. Seo *et al.*, "DeepFam: deep learning based alignment-free method for protein family modeling and prediction," *Bioinformatics*, vol. 34, no. 13, pp. i254–i262, Jul. 2018.

[107] N. G. Sgourakis *et al.*, "A method for the prediction of GPCRs coupling specificity to G-proteins using refined profile Hidden Markov Models," *BMC Bioinformatics*, vol. 6, no. 1, p. 104, 2005.

[108] R. Shrivastava, "A hidden Markov model based dynamic hand gesture recognition system using OpenCV," in *2013 3rd IEEE International Advance Computing Conference (IACC)*, 2013, pp. 947–950.

[109] G. Singh *et al.*, "FPGA-Based Near-Memory Acceleration of Modern Data-Intensive Applications," *IEEE Micro*, vol. 41, no. 4, pp. 39–48, Aug. 2021.

[110] K. Sinha *et al.*, "A Computer Vision-Based Gesture Recognition Using Hidden Markov Model," in *Innovations in Soft Computing and Information Technology*, J. Chattopadhyay *et al.*, Eds. Singapore: Springer Singapore, 2019, pp. 55–67.

[111] P. Skewes-Cox *et al.*, "Profile Hidden Markov Models for the Detection of Viruses within Metagenomic Sequence Data," *PLOS ONE*, vol. 9, no. 8, p. e105067, Aug. 2014.

[112] S.-I. Soiman *et al.*, "A parallel accelerated approach of HMM Forward Algorithm for IBM Roadrunner clusters," in *2014 International Conference on Development and Application Systems (DAS)*, 2014, pp. 184–188.

[113] M. Steinegger *et al.*, "HH-suite3 for fast remote homology detection and deep protein annotation," *BMC Bioinformatics*, vol. 20, no. 1, p. 473, 2019.

[114] J. Söding *et al.*, "The HHpred interactive server for protein homology detection and structure prediction," *Nucleic Acids Research*, vol. 33, no. suppl_2, pp. W244–W248, Jul. 2005.

[115] I. A. Tamposis *et al.*, "Semi-supervised learning of Hidden Markov Models for biological sequence analysis," *Bioinformatics*, vol. 35, no. 13, pp. 2208–2215, Jul. 2019.

[116] C. Tarnas and R. Hughey, "Reduced space hidden Markov model training," *Bioinformatics*, vol. 14, no. 5, pp. 401–406, Jun. 1998.

[117] A. Tavanaei and A. S. Maida, "Training a Hidden Markov Model with a Bayesian Spiking Neural Network," *Journal of Signal Processing Systems*, vol. 90, no. 2, pp. 211–220, Feb. 2018.

[118] Y. Turakhia *et al.*, "Darwin: A Genomics Co-Processor Provides up to 15,000X Acceleration on Long Read Assembly," *SIGPLAN Not.*, vol. 53, no. 2, pp. 199–213, Mar. 2018.

[119] P. Turjanski and D. U. Ferreiro, "On the Natural Structure of Amino Acid Patterns in Families of Protein Sequences," *The Journal of Physical Chemistry B*, vol. 122, no. 49, pp. 11 295–11 301, Dec. 2018.

[120] R. Vaser *et al.*, "Fast and accurate de novo genome assembly from long uncorrected reads," *Genome Research*, vol. 27, no. 5, pp. 737–746, 2017.

[121] R. Vicedomini *et al.*, "Multiple profile models extract features from protein sequence data and resolve functional diversity of very different protein families," *Molecular Biology and Evolution*, p. msac070, Mar. 2022.

[122] A. S. Vieira *et al.*, "T-HMM: A Novel Biomedical Text Classifier Based on Hidden Markov Models," in *8th International Conference on Practical Applications of Computational Biology & Bioinformatics (PACBB 2014)*, J. Saez-Rodriguez *et al.*, Eds. Cham: Springer International Publishing, 2014, pp. 225–234.

[123] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[124] B. J. Walker *et al.*, "Pilon: An Integrated Tool for Comprehensive Microbial Variant Detection and Genome Assembly Improvement," *PLoS ONE*, vol. 9, no. 11, p. e112963, 2014.

[125] L. Wang and T. Jiang, "On the Complexity of Multiple Sequence Alignment," *Journal of Computational Biology*, vol. 1, no. 4, pp. 337–348, Jan. 1994.

[126] R. Wertenbroek and Y. Thoma, "Acceleration of the Pair-HMM forward algorithm on FPGA with cloud integration for GATK," in *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2019, pp. 534–541.

[127] R. Wheeler and R. Hughey, "Optimizing reduced-space sequence analysis," *Bioinformatics*, vol. 16, no. 12, pp. 1082–1090, Dec. 2000.

[128] T. J. Wheeler *et al.*, "Dfam: a database of repetitive DNA based on profile hidden Markov models," *Nucleic Acids Research*, vol. 41, no. D1, pp. D70–D82, 2012.

[129] X. Wu *et al.*, "17.3 GCUPS Pruning-Based Pair-Hidden-Markov-Model Accelerator for Next-Generation DNA Sequencing," in *2020 IEEE Symposium on VLSI Circuits*, 2020, pp. 1–2.

[130] X. Wu *et al.*, "A High-Throughput Pruning-Based Pair-Hidden-Markov-Model Hardware Accelerator for Next-Generation DNA Sequencing," *IEEE Solid-State Circuits Letters*, vol. 4, pp. 31–35, 2021.

[131] C. Xue, "A Novel English Speech Recognition Approach Based on Hidden Markov Model," in *2018 International Conference on Virtual Reality and Intelligent Systems (ICVRIS)*, 2018, pp. 1–4.

[132] X. Yin *et al.*, "ARGs-OAP v2.0 with an expanded SARG database and Hidden Markov Models for enhancement characterization and quantification of antibiotic resistance genes in environmental metagenomes," *Bioinformatics*, vol. 34, no. 13, pp. 2263–2270, Jul. 2018.

[133] B.-J. Yoon, "Hidden Markov Models and their Applications in Biological Sequence Analysis," *Current Genomics*, vol. 10, no. 6, pp. 402–415, 2009.

[134] L. Yu *et al.*, "GPU-Accelerated HMM for Speech Recognition," in *2014 43rd International Conference on Parallel Processing Workshops*. IEEE, 2014, pp. 395–402.

[135] H. Zeinali *et al.*, "Text-dependent speaker verification based on i-vectors, Neural Networks and Hidden Markov Models," *Computer Speech & Language*, vol. 46, pp. 53–71, Nov. 2017.

[136] Q. Zhan *et al.*, "ProbPFP: a multiple sequence alignment algorithm combining hidden Markov model optimized by particle swarm optimization with partition function," *BMC Bioinformatics*, vol. 20, no. 18, p. 573, Nov. 2019.

[137] L. Zhang *et al.*, "FISH: fast and accurate diploid genotype imputation via segmental hidden Markov model," *Bioinformatics*, vol. 30, no. 13, pp. 1876–1883, Jul. 2014.

[138] Z. Zhang and W. I. Wood, "A profile hidden Markov model for signal peptides generated by HMMER," *Bioinformatics*, vol. 19, no. 2, pp. 307–308, 2003.

[139] A. V. Zimin and S. L. Salzberg, "The genome polishing tool POLCA makes fast and accurate corrections in genome assemblies," *PLOS Computational Biology*, vol. 16, no. 6, p. e1007981, Jun. 2020.