ApHMM: A Profile Hidden Markov Model Acceleration Framework for Genome Analysis

ABSTRACT

Profile hidden Markov models (pHMMs) are widely used in many bioinformatics applications as they can capture differences between biological sequences (e.g., DNA or protein sequences) to accurately identify their similarities. PHMMs represent sequences with a graph structure such that their states and edges are associated with probabilities to account for changes within sequences, which is then used to calculate a similarity score of a sequence compared to a pHMM graph. Accurately setting the probabilities in pHMMs is essential to correctly identify similarities between sequences. To achieve this, the Baum-Welch algorithm is a commonly-used and highly-accurate method that can change the probabilities of a pHMM to maximize the similarity score of a provided set of input sequences. However, the Baum-Welch algorithm is computationally expensive and existing works accelerate the Baum-Welch algorithm *only* for the widely-adopted traditional design of pHMMs, which cannot support *all* the bioinformatics applications using pHMMs.

We propose *ApHMM*, the *first* hardware-software co-design framework to accelerate the Baum-Welch algorithm for a wide range of pHMM-based applications. ApHMM 1) uses a modified design of traditional pHMMs that can support the pHMM-based applications, 2) reduces the computational and data movement overheads of the Baum-Welch algorithm with software-level optimizations, and 3) provides a hardware design to accelerate the execution of the Baum-Welch algorithm for the modified design of pHMMs. We implement our software optimizations on both CPUs and GPUs, and hardware-software optimizations on a specialized hardware to provide an acceleration framework for all the pHMM-based applications. Our evaluation using a wide range of applications shows that ApHMM outperforms the state-of-the-art CPU implementations of three use cases 1) error correction, 2) protein family search, and 3) multiple sequence alignment by 59.94×, 1.75×, and 1.95×, respectively, while improving their energy efficiency by 64.24×, 1.75×, and 1.96×. When compared to CPU, GPU, and FPGA implementations, ApHMM accelerates the Baum-Welch computation by 15.55×- 260.03×, 1.83×- 5.34×, and 27.97×, respectively.

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) [21]. 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 to visiting a state from another. HMMs are very efficient in decoding the continuous and discrete series of events in many applications [60] such as speech recognition [14, 35, 45, 55, 60], text classification [41, 89], gesture recognition [29, 61, 64, 69, 80], and bioinformatics [10, 26, 39, 83, 93]. The graph structure (i.e., design) of the 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 [20], which is commonly adopted in bioinformatics [94] as well as other fields such as malware detection [53, 75] and pattern matching [72].

It is an essential step in bioinformatics applications to identify similarities between biological sequences (i.e., DNA or protein sequences) while calculating the differences between them, which is important to understand the potential effects and nature of these differences (e.g., natural mutations in evolution). PHMMs enable efficient and accurate identification of similarities between sequences by comparing sequences to a few graphs rather than comparing many sequences to each other, which is computationally very costly and requires special hardware and software optimizations [2, 3, 4, 18, 27, 57, 63, 73, 74, 79, 85]. Figure 1 illustrates a commonly-adopted design of pHMMs. A pHMM models a single or many sequences into a single graph structure with a certain design of states and transitions between them. Each state has a *role* to account for any difference (i.e., insertion (I), substitution or match (M), and deletion (D)) at a designated position between a graph and an *input sequence*. Each state is visited via transitions with a certain probability by using input sequences as observations to calculate the similarity between the input sequence and the pHMM graph. Certain bioinformatics applications employ the use of pHMMs rather than directly comparing sequences as these applications usually compare a sequence to a group of sequences (e.g., protein family), which can be represented by pHMMs to avoid the high cost of many sequence comparisons. The applications that use pHMMs include protein family search [6, 7, 28, 78, 83, 97], multiple sequence alignment (MSA) [5, 19, 22, 23, 56, 93], and error correction [25, 26, 46].

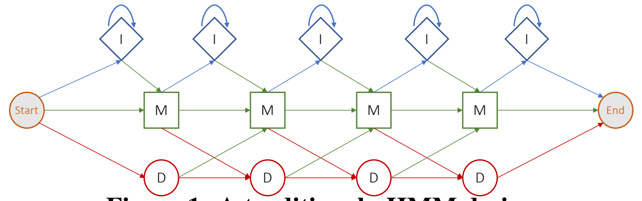

Figure 1: A traditional pHMM design.

To accurately model and compare DNA or protein sequences using pHMMs, assigning accurate probabilities to states and transitions are essential. Although the probabilities associated with states and transitions in pHMMs can be assigned either randomly or based on a simple prior knowledge [26], pHMMs allow updating these probabilities to enable fitting the observed biological sequences to the pHMM graph more accurately with a step called *parameter estimation* or *training*. The training phase 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 [8, 49, 71, 76]. The Baum-Welch algorithm [8] is commonly used to calculate the likelihood maximization [11] as it is highly accurate and scalable to real-size problems (e.g., large protein families) [49]. The next step is *inference*, which aims to identify either 1) similarity of an input observation sequence to a pHMM graph or 2) the *consensus* sequence that gives the *best* similarity score from a pHMM graph.

Despite its advantages, the Baum-Welch algorithm is a computationally expensive method due to the iterative nature of computations, which is exacerbated when the Baum-Welch algorithm is executed many times to maximize the training accuracy [95]. Several works [22, 68, 70, 95] aim to accelerate either the entire or smaller parts of the Baum-Welch algorithm to mitigate the high computational costs while still producing accurate estimations on the pHMM parameters. However, these works mainly adopt the traditional design of pHMMs, which has two main limitations. First, identifying the consensus sequence is an essential task for some bioinformatics applications (e.g., error correction), which is an NP-hard problem when using the traditional pHMM design [40, 54]. Prior works [22, 83] follow simple heuristics to efficiently identify the consensus sequence [40], which may result in incomplete and inaccurate consensus sequences. Second, the design of the traditional pHMMs cause ambiguities for the Baum-Welch algorithm that reduces the inference accuracy [25, 67]. For example, some states have self-loops in a traditional pHMM as shown in Figure 1 to account for multiple insertions in a row. However, using the same state for multiple insertions makes it challenging for precisely determining the correct order of each inserted character as the loops do not provide the order information. The problem with self-loops is further exacerbated when identifying the consensus sequence as *both* the order and number of inserted characters are unknown in the consensus sequence. The traditional pHMM design is not preferable for efficiently and accurately identifying the consensus sequence due to these two limitations. To overcome these main limitations in traditional pHMMs, a recent work [25] proposes a *modified* design of pHMMs to enable more efficiently and accurately generating consensus sequences from pHMMs than traditional pHMMs. Although the modified pHMM design enables new applications by accurately constructing consensus sequences from pHMMs [25, 26, 46], the training step is still computationally costly due to the Baum-Welch algorithm. To our knowledge, there is no prior work that accelerates the Baum-Welch algorithm for the modified design of pHMMs.

Our **goal** is to accelerate the Baum-Welch algorithm while eliminating the main limitations associated with the traditional pHMMs. To this end, we propose ApHMM, the *first* hardware-software co-design to build an acceleration framework for a wide range of bioinformatics applications that use pHMMs. ApHMM is built on four **key ideas**. First, ApHMM adopts the modified pHMM design to resolve the complexity and ambiguity issues in the traditional pHMMs, which enables supporting more pHMM-based applications that traditional pHMM cannot efficiently and accurately support. Second, ApHMM identifies the pipelining opportunities in the Baum-Welch algorithm to minimize the memory footprint of the algorithm. Third, ApHMM exploits the iterative nature of Baum-Welch computations to minimize the data movement overhead by intelligently keeping the computation from most recent iterations in registers and multiple levels of cache. Fourth, ApHMM avoids redundant computations by identifying the most common and identical multiplications from the Baum-Welch algorithm and storing the result of these multiplications in lookup tables (LUTs). These hardware and software optimizations enable ApHMM to accelerate the Baum-Welch algorithm using the modified pHMM design for a wide range of bioinformatics applications.

We evaluate the performance and energy efficiency of ApHMM for executing 1) the Baum-Welch algorithm and 2) several pHMM-based applications and compare 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* hardware-software framework to accelerate and improve pHMMs. We show that our framework can be used for three bioinformatics applications: 1) error correction, 2) protein family search, 3) multiple sequence alignment.
- We provide ApHMM-GPU, the *first* GPU implementation of the Baum-Welch algorithm for pHMMs.
- We provide hardware and software optimizations for improving the performance and reducing the data movement overhead of executing the Baum-Welch algorithm.
- 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.

2 Background

2.1 Profile Hidden Markov Models (pHMMs)

2.1.1 High-level Overview

We explain the *modified* design of profile Hidden Markov Models (pHMMs). Figure 2 shows the general structure of the modified design of pHMMs. To represent a biological sequence while accounting for differences that this biological sequence can have when compared to another sequence, pHMMs have a certain graph structure that associates probabilities for visiting nodes, called *states*, via directed edges, called *transitions*. There are fixed number of states created for each character of the represented sequence. When visited, states emit one of the characters from the defined alphabet of the biological sequence (e.g., A, C, T, and G in DNA sequences). Emitting each character in a state has a certain probability assigned based on the represented sequence. Transitions preserve the correct order of the represented sequence. The probabilities and layout of states and transitions allow making changes to the represented sequence at any position (i.e., insertion, deletion, or substitution). Overall, pHMMs represent a sequence while making it possible to assign probabilities to make certain changes to the represented sequence.

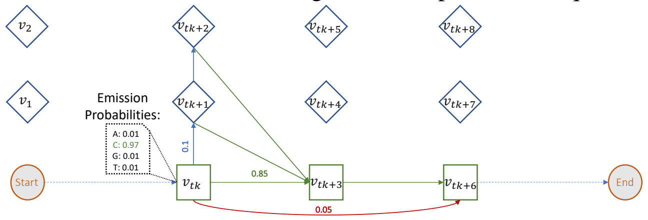

Figure 2: Overall structure of a modified pHMM design.

To represent and compare a biological sequence with another, pHMMs are used in three steps. First, to represent a sequence, pHMM builds the states and transitions by iterating over the each character of the sequence. A pHMM graph includes n-many states for each character and each state has transitions to 1) another state of its own character and 2) states of next characters to preserve correct order of the sequence. Multiple sequences can be represented with a single pHMM graph. To build such a pHMM, the differences between each sequence pair should previously be calculated. To this end, multiple sequence alignment (MSA) is used to identify the relation between each sequence, which is then used to generate the 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 [8] is an 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. 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. 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 [90] are commonly used for inference from pHMMs [13, 28].

2.1.2 Components of pHMMs

To accurately represent a sequence, modified pHMMs use four component: 1) states, 2) transitions, 3) emission probabilities and 4) transition probabilities, as shown in Figure 2. 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}. We represent the *states* and *transitions* as the members of the sets V and A, respectively. First, for each character of S_G at position t, $S_G[t] \in S_G$, pHMMs include k-many consecutive states, $v_{t\times k}, v_{t\times k+1}, \ldots, v_{k\times t+k-1} \in V$. Each of these k states either modifies the character $S_G[t]$ or inserts additional characters. Second, pHMM graphs include transitions from state a v_i to a state v_j, $\alpha_{i,j} \in A$, 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 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. These probabilities allow 1) visiting additional states for insertions and 2) skipping states for deletions in a sequence. We represent the transition probability of a character between states v_i and v_i as $t(\alpha_{i,j})$. 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

States and transitions in modified pHMMs are designed to account for any modification that a sequence can have. These modifications are substitutions, insertions, and deletions. We use the term *match* when a character $S_G[t]$ does *not* need any modification (i.e., insertion, deletion, or substitution). There are n_{S_G} many states in $G(V,A)$, i.e., a state for each character of S_G to handle the matches. We refer to these states as *match states*. The match state $v_{t\times k}$ of the character $S_G[t]$ is usually set to have a higher emission probability for emitting the character $S_G[t]$ than emitting any other character in the same match state, i.e., $e(S_G[t],v_i) >= e(s,v_i)\ \forall s \in \Sigma$. We denote the probability $e(S_G[t],v_i)$ as *match emission probability*.

The *substitution* term refers to a modification where a character is replaced by another character. To account for a substitution of character $S_G[t]$, a pHMM graph emits a character in the match state $v_{t\times k}$ different than $S_G[t]$ such that $S_G[t]$ is substituted with c where $c \in \Sigma,\ \ c \neq S_G[t]$ with the *substitution probability* of $e(c,v_i)$.

An *insertion* event occurs when a sequence of characters, I, needs to be placed after the character $S_G[t]$. In modified pHMMs, there are $l \times n_{S_G}$ $(l = k-1)$ many states to insert l-many characters after *each* character of S_G. We denote such states as *insertion states*. Series of insertion states, $v_{t\times k+1}, \ldots, v_{t\times k+|I|}$, are visited to insert the sequence of characters I after the character $S_G[t]$. An insertion state v_i is visited using a transition from state v_m with a probability of $t(\alpha_{m,i})$ to emit a character c with a probability of $e(c,v_i)$.

We refer to a deletion event when a sequence of characters in S_G needs to be ignored. A pHMM performs the deletion of character $S_G[t]$ by *not* visiting its match state, $v_{t\times k}$. To this end, modified pHMMs have *deletion transitions* from a state $v_{t-1\times k+m}$ to a *match state* $v_{t+1\times k}$ to skip the match state $v_{t\times k}$. To delete multiple characters, many match states are skipped. Overall, modified pHMMs can perform all the modifications for the sequence they represent given the constraints on the maximum number of insertions (l) and deletions (d).

2.2 The Baum-Welch Algorithm

To maximize the similarity score of input observation sequences in a pHMM graph, the Baum-Welch algorithm [8] solves an *expectation maximization* problem [34, 52, 59, 84] 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 $S[1 \dots t-1]$ 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.

$$F_t(i) = \sum_{j \in V} F_{t-1}(j)\alpha_{ji} e(S[t], v_i) \quad i \in V, \quad 1 < t \leq n_S \tag{1}$$

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 $S[t+1 \dots n_S]$ 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.

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

2.2.3 *Parameter Updates*

The Baum-Welch algorithm uses the values that the forward and backward calculations generate 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.

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

$$e^*_X(v_i) = \frac{\sum\limits_{t=1}^{n_S} F_t(i) B_t(i) [S[t] = X]}{\sum\limits_{t=1}^{n_S} F_t(i) B_t(i)} \qquad \forall X \in \Sigma, \forall i \in V \tag{4}$$

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 replace these erroneous parts with more reliable sequences [16, 31, 32, 87, 91, 98] to enable more accurate genome analysis (e.g., read mapping [27,42,51] and genome assembly [15,24,43,44,50,66]). Apollo [26] 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 construct a pHMM graph ($G(V,A)$) for an assembly sequence (S_G) 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 [90] to identify the consensus sequence from the trained pHMM, which translates into the corrected assembly sequence. Apollo uses the modified design of pHMMs to avoid limitations associated with traditional pHMMs when generating the consensus sequences as discussed in Section 1.

2.3.2 *Protein Family Search*

Classifying protein sequences into families is widely used to analyze the potential functions of the proteins of interest [9,37,62,77,86,88]. To achieve this, the goal is to find the family of the protein sequence in existing protein databases. Each protein family in the database is usually represented by a single pHMM 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 [58], where the HMMER [22] software suite is used to build HMMs and assign query sequences to the best fitting Pfam family. The same approach is also used in several other important applications, such as classifying metagenomic sequences [47,48] into potential viral families [81].

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 [38,92]. 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 [17]. The pHMM approaches such as *hmmalign* [22] 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 [96]). 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

Our goal is to identify the overhead of each step of the Baum-Welch algorithm in several use cases. Figure 3 shows the percentage of execution time of all three steps in the Baum-Welch algorithm for three bioinformatics applications: error correction, protein family search, and multiple sequence alignment (MSA). We use gprof [30] on an AMD EPYC 7742 machine to characterize the state-of-the-art CPU implementations of these applications. We make two key observations. First, we find that the Baum-Welch algorithm is the *over-*

whelming performance bottleneck for error correction as the Baum-Welch algorithm takes around 98.57% of the total CPU execution time. Second, our further analysis on protein family search and MSA reveals that the Baum-Welch algorithm constitutes around 50% of the overall execution time of both applications. Thus, there is a pressing need to develop an acceleration framework for the the Baum-Welch algorithm, which is flexible to accelerate *multiple* use cases of the Baum-Welch algorithm in different bioinformatics applications.

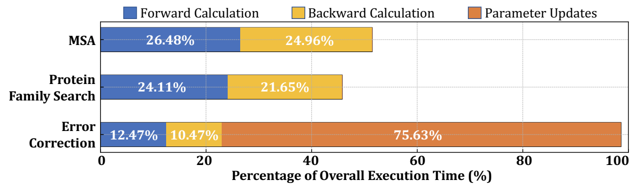

Figure 3: Execution time breakdown of the steps in the Baum-Welch algorithm

We analyze the execution time breakdown for the Baum-Welch algorithm to reveal the steps that take up most of the time for all three applications, as shown in Figure 3. We find that the parameter update step takes the majority of the total execution time for error correction, while Forward and Backward calculation steps are equally expensive for all three applications. We further analyze the available CPU implementation of a pattern matching application that use pHMMs [72]. Our initial analysis shows that almost the entire execution time of this application is spent for the Forward calculation and it takes significantly long times for executing relatively a small dataset compared to the bioinformatics applications. We conclude that a hardware-software co-design should optimize all the steps in the Baum-Welch algorithm for accelerating wide-range of applications.

4 ApHMM Design

In this work, our **goal** is to accelerate and improve the training step of pHMMs. To this end, we propose ApHMM, the *first* hardware-software co-design to build an acceleration framework for a wide range of applications that use the Baum-Welch algorithm to train pHMMs. We explain our acceleration framework, ApHMM. For simplicity, we use the graph notations and equations we introduced in Section 2 when explaining our hardware and software optimizations.

4.1 Microarchitecture Overview

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 based on the modified pHMM design. ApHMM core takes a pHMM graph $G(V,A)$, parameters associated with the pHMM (e.g., probabilities), and an input sequence S (i.e., observation) to compute *all three* steps of the Baum-Welch algorithm: 1) forward, 2) backward, and 3) parameter update computations as discussed in Section 2.2.

Figure 4 presents the microarchitectural details of a single ApHMM core. Figure 4(a) shows the high-level operating units of each ApHMM core where the units are divided into two major sections: 1) *control section* and 2) *compute section*.

Control Section is responsible for issuing both memory requests and proper commands to the compute section to configure for the next set of operations (e.g., forward calculation for the next character of sequence S). As we show in Figure 4(a) (i.e., the green box), the control section has six execution steps: Step (1) issues a read request to L1 memory to obtain each input sequence S and corresponding pHMM graph (i.e., $G(V,A)$). Step (2) issues a read request to L1 memory in order to obtain the corresponding parameters from the previous *timestamp* (e.g., $t-1$ in Equation 1) based on the current execution phase (e.g., forward phase at *timestamp* t, as shown in Equation 1). Step (3) controls the phase when updating emission probabilities, as described in Equation 4 through *Emission Update Pipeline*. Step (4) collects the write requests from various clients and avoids any duplicate requests to the memory to ensure data is synchronized. This step also keeps track of the write data histogram counter for filtering operation, which we explain in Section 4.5.1. Steps (5) and (6) arbitrate among the read and write clients, respectively, and pipeline the read and write requests to the memory. Control section uses these six steps to manage the input and output flow of the compute section efficiently and correctly.

Compute Section is responsible for performing core compute operations of the Baum-Welch algorithm based on the configuration set by the control section (e.g., position of input sequence S). The compute section reads the appropriate data passed by the control section and operates on the data. The compute section contains two major blocks: 1) a block of multiple *Processing Engine and Transition* (PET) groups to calculate forward and backward values, and update the transition probabilities, and 2) a block to update emission probabilities as shown in Figure 4(a) (the blue box). For calculating the forward and backward values, and updating the transition probabilities, ApHMM core uses multiple PET groups and a shared forward *(FWD) write selector* block that uses the PET output to either 1) write to the memory or 2) use it for updating emission probabilities. We show a further breakdown of a PET group in Figure 4(b). Each PET group is comprised of four *PET Engines* where each engine receives the appropriate data to perform the forward (FWD) and backward (BWD) computations for a state v_i and a timestamp t. The PET Engines include one *Processing Engine (PE)* and one *Trans. Pipe* (i.e., transition pipeline), which we explain in Sections 4.2 and 4.4. Each PE can perform the dot product operation on four IEEE754-single precision floating point variables to calculate the forward and backward values as shown in Equations 1 and 2, respectively. The result of these computations are distributed to updated both emission probabilities (Figure 4(c)) and transition probabilities (Figure 4(d)). The transition pipeline performs the operations to update the transition probabilities of a pHMM as shown in Equation 3. For updating the emission probabilities, ApHMM core uses the *Emission Update Pipeline* and the forward and backward values as calculated in PET groups.

4.2 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 achieve this, ApHMM uses the Processing Engines (PEs) in PET groups. In this section, we explain several hardware and software optimizations in PEs to calculate the forward and backward values: 1) broad-

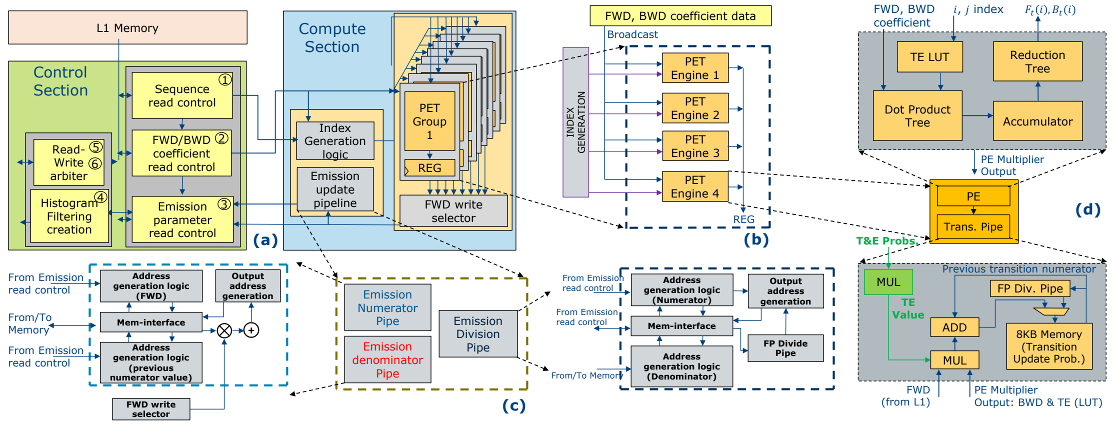

Figure 4: Microarchitecture block diagram for a single ApHMM core. (a) Each ApHMM core is composed of L1 memory and two major sections: the control section (green box) and the compute section (blue box). (b) The Processing Engine and Transition (PET) group organization represents logical placement of multiple PET engines. The PET group contains logic for index generation and broadcasting forward and backward (i.e., FWD/BWD) values to all PET engines. (c) Emission Update Pipeline and components involved in the pipeline are shown. (d) Microarchitecture for the PET engine in which Processing Engine (PE) and complete transition update pipeline (Trans. Pipe) are detailed.

casting the forward and backward values (i.e., coefficients), 2) using lookup tables, and 3) performing the dot product.

4.2.1 *Broadcasting Forward and Backward Values*

ApHMM decouples hardware scaling from bandwidth requirements by broadcasting forward and backward values from previous timestamps during the forward and backward calculation stages of the current timestamp, respectively. Each PE in a PET group is broadcasted with the *same* previously calculated $F_{t-1}(j)$ or $B_{t+1}(j)$ values from the previous timestamp for calculating the $F_t(i)$ or $B_t(i)$ values, respectively. Index generation broadcasts the index values i to calculate $F_t(i)$. This design choice exploits the broadcast opportunities available within the Baum-Welch equations.

4.2.2 *Lookup Tables*

The goal of using lookup tables in PEs is to avoid redundant multiplications of transition and emission probabilities by storing these computations in a lookup table. Since the transition and emission probabilities can be preset (i.e., fixed) values in a pHMM graph $G(V,A)$ before the training step, the resulting multiplication of any combination of these preset transition and emission probabilities can also be considered as fixed values. As Equations 1 and 2 show, multiplications of these fixed transition and emission probabilities are commonly used for every state v_i and timestamp t. To avoid redundant multiplications of these probabilities, our **key** design choice is to implement Transition×Emission (TE) lookup tables (*TE LUTs*) in PEs that capture all possible combinations of fixed probabilities. We store the products of every fixed combinations in TE LUT because these products are common operations in *both* forward and backward calculation. Each PE includes a TE LUT to fetch the correct value to perform the dot product as shown in Figure 4(d).

TE LUTs provide ApHMM with a bandwidth reduction up to 66% per PE while avoiding redundant computations. One TE LUT is utilized per multiplier, which translates to 4 TE LUTs per PET Group. A TE LUT can hold up to 36 LUT entries. Any application can utilize these TE LUTs given that these applications need at most 36 entries of distinct combination of transition and emission probabilities. Otherwise, as shown in Figure 4(d), ApHMM reads emission and transition probabilities from the memory (highlighted in green as *T&E Probs.* in Figure 4(d)) to perform the corresponding multiplication without using TE LUTs.

4.2.3 *Performing the Dot Product*

Our goal is to calculate the forward and backward values of a state v_i per timestamp t. To achieve this, each PE performs two main operations. First, it uses *Dot Product Tree* to perform the multiplication of the forward or backward values from previous timestamps (i.e., $F_{t-1}(j)$ or $B_{t+1}(j)$) and the TE value received from the TE LUT. We use Dot Product Tree to perform and store the multiplication result for index i and different values of j indices to perform multiple multiplications in parallel. Second, PE accumulates these multiple multiplication results from the Dot Product Tree using *Accumulator*, which has four parallel lanes to perform the accumulation in parallel. Third, PE uses *Reduction Tree* to perform the final summation by accumulating the values from four lanes in the Accumulator to calculate $F_t(i)$ or $B_t(i)$. The output of each PE is either 1) stored in a memory for a later use if forward value is calculated or 2) transferred to Transition Pipeline and Emission Update Pipeline without storing the output in the memory if the backward value is calculated. For the latter, ApHMM core is designed to directly consume the backward value and the product of transition and emission probability as fetched from TE LUTs, which is an software optimization that we call partial compute approach as explained next.

4.2.4 *The Partial Compute Approach*

The goal of using partial computes is to decrease the overall memory accesses and memory footprint while increasing the parallelism of the ApHMM along with keeping the utilization of the compute resources. To this end, our **key** design choice enables updating the transition and emission probabilities while the backward calculation is still on progress. To achieve this, backward values are computed in PEs and immediately used to update transition and emission probabilities of the state, v_i. We co-design hardware and software to enable the PET Engines such that they can send the backward value to *both* Emission Update Pipeline (Figure 4(a)) and the Transition Pipeline within the PET Engine (Figure 4(d)), which are responsible for updating emission and transition probabilities, respectively. We assume forward calculation phase is *fully completed* for a sequence S. This **key** design choice enables ApHMM to use backward values directly from PEs instead of writing and retrieving them from memory to avoid redundant memory access.

4.3 Updating Emission Probabilities

Our goal is to update the emission probabilities in $G(V,A)$ of all the states as shown in Equation 4. To achieve this, we use the *Emission Update Pipeline* block as shown in Figure 4(c), which includes three smaller blocks: the 1) Emission Numerator, 2) Emission Denominator, and 3) Emission Division Pipes. The pipeline performs the numerator and denominator computations in parallel as they are independent of each other, which includes a summation of the product $F_t(i)B_t(i)$. To reduce redundant computations, our **key** design choice is to use the $F_t(i)$ and $B_t(i)$ values as calculated in the transition update phase since these values are also used for updating the emission probabilities. Thus, we broadcast these values to the Emission Update Pipeline through *FWD Write Selector* and *Emission parameter read control* as shown in Figure 4(a) to perform the final multiplication for both numerator and denominator of Equation 4.

The ApHMM core writes and reads both 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, ApHMM stores n_Σ many different numerators for each state of the graph, as emission probability may differ per character for each state. To build a framework for many applications, our microarchitecture design is **flexible** such that it allows defining n_Σ as a parameter.

4.4 Updating Transition Probabilities

Our goal is to update the transition probabilities in $G(V,A)$ of all the states as shown in Equation 3. To achieve this, we use the *Transition Pipeline* block as shown in Figure 4(d). The goal of transition pipeline is to efficiently calculate the denominator and numerator in Equation 3 for a state v_i. To achieve this, transition pipeline includes three main components. First, a local 8KB memory stores the result of the numerator of Equation 3 as calculated until the current timestamp t. Second, the multiplication (MUL in yellow box) operation multiplies all the products of $\alpha_{ij}e_{S[t+1]}(v_j)F_t(i)B_{t+1}(j)$ as shown in Equation 3. The value for $B_{t+1}(j)$ is directly fetched from PE (i.e., *PE Multiplier Output*) and $F_t(i)$ is read from memory as we assume the forward values are fully computed before updating transition and emission probabilities. For $\alpha_{ij}e_{S[t+1]}$, the result of the product is either 1) fetched from PE as generated from TE LUTs if these values are fixed or 2) calculated using the custom transition and emission probabilities (*T&E Probs.* and MUL in green box) to generate the *TE value* (i.e., Transition$\times$Emission value in green). Third, the result of the multiplication for the current timestamp of Equation 3 is accumulated with the previous timestamps using the ADD operation. Fourth, when all timestamps are processed, the division operation is signaled to *FP Div. Pipe* and the multiplexer to handle the division operation as shown in Equation 3. The division operation starts reading the numerator from the local 8KB memory and the denominator from L1 memory to update the transition probability.

One **key** design choice in the transition pipeline is to intelligently use local 8KB memories to cache the multiplication result of the numerator of transition update from previous timestamps to reduce the data movement overhead. Trans. Pipe uses the forward and backward values from current timestamp to update the transition probability between state pairs v_i and v_j while fetching the remaining values of the same pair from previous timestamp from the local memory. ApHMM writes the updated transition probability to the same location of the previous timestamp in local memory to *reuse* it in later timestamps as the PET Engine keeps computing the forward and backward values for the same state pairs at different timestamps. The memoization technique allows 1) skipping redundant multiplications and 2) reducing the bandwidth requirement by $2\times$ per transition pipeline.

4.5 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 [12]. For longer sequences, a sequence may be chunked into small pieces while preserving the relative order between sequences. An analysis on a similar software-level optimization reveals that chunking does not degrade the accuracy of the training and inference steps [26].

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. Figure 5 shows the size of different Baum-Welch parameters that needs to be stored in memory based on the sequence (e.g., read) length. It also captures the details for storing the data efficiently across the memory hierarchy. Since 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 forward values in DRAM and fetches them into L2 and L1 memory when they need to be used. 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 5. Our key observation from the data distribution is that the size of Baum-Welch parameters grows as the sequence length increases. Thus, number of sequences that L1 memory can hold reduces with the increase in read length. This does not cause frequent data load from DRAM or the L2 memory as longer sequences occupies the ApHMM core usually for a longer duration, which compensates for the less number of read sequences stored in L1.

4.5.1 Filtering Mechanism

Our goal is to reduce the amount of states we use from previous timestamps when calculating the forward and backward values of the current timestamps, which subsequently reduces the overall forward and backward calculation in the Baum-Welch algorithm. To achieve this, ApHMM filters out the states that result in the smallest forward and backward

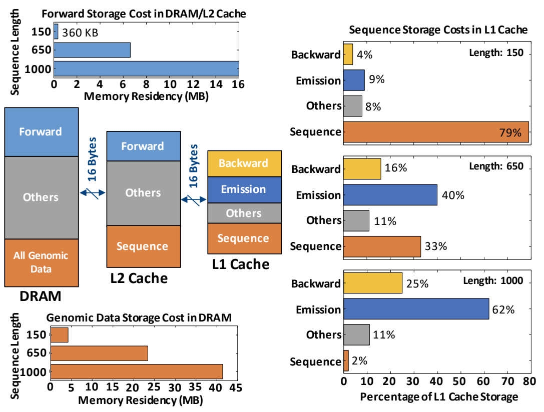
Figure 5: Storage and data distribution across memory hierarchy in ApHMM.

values among other states for the same timestamp. We filter these states out as their contributions in the next timestamps are negligible compared to the states that provide higher forward and backward values. A similar filtering approach is applied in both Hercules [25] and Apollo [26]. Such a filtering aims to prevent the rapid growth of the number of states that needs to be computed at each timestamp without reducing the accuracy of the Baum-Welch algorithm [26]. However, the filtering approach used in Hercules and Apollo performs sorting operation, which is challenging to efficiently implement in a hardware accelerator.

To implement a similar filtering approach, we employ a *histogram-based* filtering mechanism in the ApHMM memory. Our filtering divides the entire range of single precision floating point number into 16 equal parts (e.g., the range between two parts is 4.25E^{+37}) to perform filtering. ApHMM locates the states in the memory based on their forward and backward values. To achieve this, ApHMM compares the forward or backward values from the most recent timestamps to these 16 predefined threshold values. The addresses of the states are assigned such that all the states that are 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. ApHMM allows disabling the filtering mechanism if the application does not require a filter operation to achieve more optimal computations. To build a **flexible** framework for many applications, the microarchitecture is configurable to vary this threshold values for as needed based on the application and the average sequence length.

4.6 System Mapping and Execution Flow

We show a system level scale up version of the ApHMM core in Figure 6. ApHMM uses the L2-DMA table to load the data into the L2 memory and uses 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 4.5. ApHMM enables Trans-DMA to load the transition probabilities from DRAM to the local memory when the TE LUT is not utilized as discussed in Section 4.2.2. In such a scenario, local memory inside the PET engine is loaded with appropriate transition probability data to perform the multiplications without using the TE LUT.

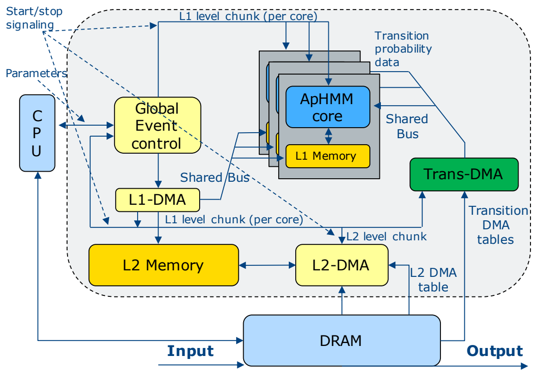
Figure 6: System integration of the ApHMM core.

We present the execution flow of the system with multi-ApHMM core in Figure 7. The operation starts with 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 next set of sequences from L2. Similarly, a counter based signaling is used to indicate L2 to fetch next set of sequences from DRAM. Once all reads are issued ApHMM sends a completion signal and releases the control back to the host. ApHMM delays fetches from L2-L1/DRAM-L2, if there are multiple iterations for the Baum-Welch algorithm until the training parameters converge to a certain value.

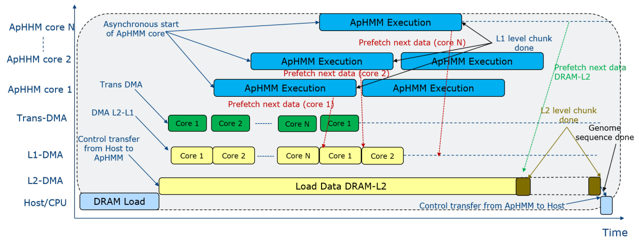
Figure 7: Control and Execution flow between Host and the ApHMM core.

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) based on various hardware configuration and memory bandwidth requirements. 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 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 up to 8 ApHMM cores in our evaluation to show the performance benefits when scaling up ApHMM to multiple cores. 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 Engine	PEs (#): 64, Multipliers per PE (#): 4, Adders per PE (#): 4 Memory per PE: 8, Transition Pipes (#): 64, Emission Pipes (#): 4

We use the CUDA library [65] (version 11.6) to provide a GPU implementation of the software optimizations we describe in Section 4 for executing the Baum-Welch algorithm. Our GPU implementation, **ApHMM-GPU**, implements LUTs (Section 4.2.2) as a shared memory and the Partial Compute Approach (Section 4.2.4) to reflect the software optimizations of ApHMM in GPUs. We integrate our GPU implementation with a pHMM-based error correction tool, Apollo [26] 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 [30] to profile the baseline CPU implementations of the use cases on 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 evaluate ApHMM and compare 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 [26] 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 [26], hmmsearch [22], and hmmalign [22]. 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. We compare the use cases that we accelerate using ApHMM to their corresponding CPU, GPU, and FPGA implementations when available. For the GPU implementations, we use both ApHMM-GPU and HMM_cuda [95]. For the FPGA implementation, we use the FPGA Divide and Conquer (D&C) accelerator proposed for the Baum-Welch algorithm [68]. 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 [51] and miniasm [50] to find *overlapping reads* by mapping reads to each other and 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 of average length 94.2. We use these sequences to search for similar protein families from the entire Pfam database [58] that includes 19,632 pHMMs. To achieve this, the hmmsearch [22] 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 [22] tool performs the Forward and Backward calculations to find similarities between a single pHMM graph and sequences.

5.2 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 its 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 TE LUTs are disabled (Section 4.2.2). Since these ports are shared across each PET Engine, 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 FWD write selector, 4) writing the output. These memory ports are independent from the impact of the number of PEs in a single ApHMM core. Fourth, ApHMM does not require additional memory ports to perform a dot product as a result of the broadcasting feature of ApHMM that we describe in Section 4.2.1. Instead, computing the dot product for the forward and backward calculation 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 overall requirement for the ApHMM core is 8 memory ports with same memory bandwidth per port.

In Figure 8(a), we show the acceleration speedup while scaling ApHMM with number of PEs and bandwidth per memory port, where we keep the number of memory ports fixed to 8. Based on Figure 8(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 8(b). We find that the acceleration on the transition phase starts settling down as the number of PEs grows due to memory port limitation that reduces parallel data read from memory per PE, which eventually results in under-utilization of resources. We conclude that the acceleration trend we observe in Figure 8(a) is mainly due to scaling impact on the forward and backward calculation when the number of PEs is greater than 64 where 8 memory ports starts becoming the bottleneck.

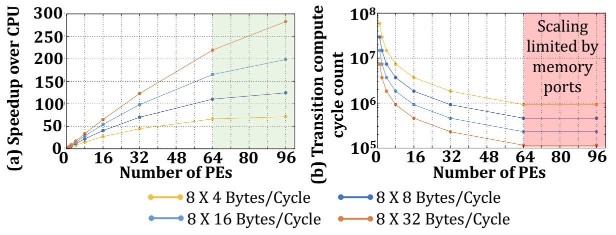

Figure 8: (a) Acceleration scaling with number of processing element. (b) Transition probability compute cycle acceleration with increase in number of PEs.

5.3 Area and Power

Table 2 shows the area breakup of the major modules in ApHMM. For the area overhead, we find that the transition pipeline takes the majority of the total area (77.98%). This is mainly because the transition pipeline consists of several complex units such as multiplexer, division pipeline, local memory. For the power consumption, control logic and PEs contribute to almost the entire power consumption (86%), due to the frequent memory accesses these blocks make. Overall, 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 Logic	0.011	134.4
64 Processing Engines	1.333	304.2
64 Transition Pipelines	5.097	0.8
4 Emission Pipelines	0.094	70.4
Overall	6.536	509.8
128KB L1-Memory	0.632	100

5.4 Accelerating the Baum-Welch Algorithm

Our goal is to show the performance and energy improvements of ApHMM for executing the Baum-Welch algorithm as we show in Figure 9. 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, we observe that 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 the software-hardware optimizations we make in ApHMM, which reduce computations and memory accesses while placing the data more intelligently for a better performance and energy consumption. 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 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 4.2.4. 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 a better performance for Forward calculation than ApHMM. We believe that the GPU implementations are 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 accesses to host for synchronisation 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 speedups and reduces the energy consumption 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.

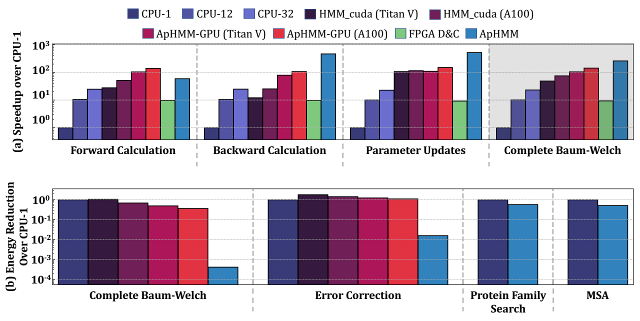

Figure 9: (a) Speedups compared to the single-threaded CPU (CPU-1) implementation. The speedup of each step in the Baum-Welch algorithm is shown separately. (b) Energy reductions compared to the CPU-1 implementation of the Baum-Welch algorithmm and three use cases that use the Baum-Welch algorithm.

5.5 Number of ApHMM cores

We show our methodology for choosing the ideal number of ApHMM-cores for accelerating the applications. Figure 10 shows the speedup of three applications when using single, 2, 4, and 8 ApHMM cores. 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 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 for further reducing the data movement overheads. We use 4-core ApHMM for evaluating the use cases.

5.6 Use Case 1: Error Correction

Figures 11 and 9 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

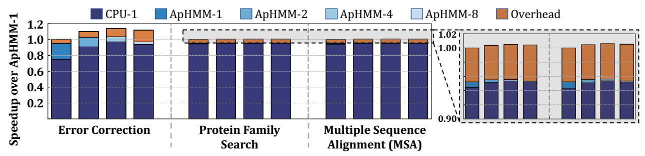

Figure 10: Speedups when using multi-core ApHMM compared to the single core ApHMM (ApHMM-1).

by 64.24× and 71.28×- 115.46× compared to the single-threaded CPU and GPU implementations. These two observations are in line with our observations we make in Section 5.4 as well as our 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 constitute the large portion of the entire use case.

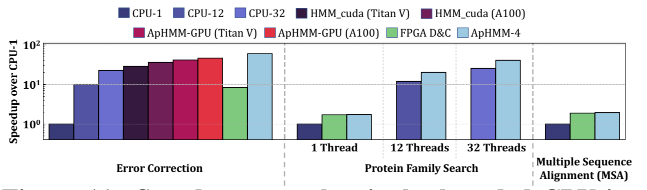

Figure 11: Speedups over the single-threaded CPU implementation of three use cases. In protein family search, we compare ApHMM with each CPU thread separately.

5.7 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 11 and 9, 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 and FPGA implementations. Second, we observe that 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 alphabet size of protein alphabet size (i.e., 20) is much larger than the DNA alphabet size (4), which increases the DRAM access overhead of ApHMM by 12.5%. Due to 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, we observe that 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 Figures 11 (1 Thread, 12 Threads, and 32 Threads). We conclude that ApHMM improves the performance and energy efficiency of the protein family search use case, while there is smaller room for acceleration compared to the error correction use case.

5.8 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 11 and 9, respectively. We make three key observations. First, we observe that ApHMM performs 1.95× and 1.03× better than the CPU and FPGA implementations, while ApHMM is 1.96× 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 3. 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 accelerates the Baum-Welch algorithm based on the modified pHMM design. In this section, we explain previous attempts to accelerate *pHMMs*. A group of previous work [22,33,36,68,82,95] focuses on specific algorithms and designs of HMMs. Ibrahim et al. [36] propose a FPGA-based accelerator for pHMMs to accelerate the inference step (i.e., the Viterbi algorithm). Haung et al. [33] and Soiman et al. [82] accelerate the Forward calculation based on HMM designs different than pHMMs for FPGAs and supercomputers, respectively. HMM_cuda [95] use 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 none of these works focus on accelerating the Baum-Welch algorithm on pHMMs.

HMMER3 [22] proposes to accelerate pHMM using SIMD operations for executing Baum-Welch algorithm based on the the traditional pHMM design. FPGA D&C [68] accelerates the Baum-Welch using FPGAs based on the traditional design of pHMMs. ApHMM differs from these works as ApHMM the Baum-Welch algorithm based on the modified pHMM design, which allows ApHMM to support a wide range of applications that traditional pHMMs cannot support.

7 Conclusion

We propose ApHMM, the the *first* hardware-software co-design framework that accelerates a wide range of bioinformatics applications that use 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 implement the Baum-Welch algorithm based on the modified pHMM design. 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., the inference step) based on the modified version of pHMMs. We also suggest a special focus on processing-in-memory as a future work to minimize the data movement overhead that ApHMM is bottlenecked.

References

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

[2] M. Alser, Z. Bingöl, D. Senol Cali, J. Kim, S. Ghose, C. Alkan, and O. Mutlu, "Accelerating Genome Analysis: A Primer on an Ongoing Journey," *IEEE Micro*, vol. 40, no. 5, pp. 65–75, Oct. 2020.

[3] M. Alser, J. Rotman, D. Deshpande, K. Taraszka, H. Shi, P. I. Baykal, H. T. Yang, V. Xue, S. Knyazev, B. D. Singer, B. Balliu, D. Koslicki, P. Skums, A. Zelikovsky, C. Alkan, O. Mutlu, and S. Mangul, "Technology dictates algorithms: recent developments in read alignment," *Genome Biology*, vol. 22, no. 1, p. 249, Aug. 2021.

[4] M. Alser, T. Shahroodi, J. Gómez-Luna, C. Alkan, and O. Mutlu, "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.

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

[6] P. Baldi, Y. Chauvin, T. Hunkapiller, and M. A. McClure, "Hidden Markov models of biological primary sequence information." *Proceedings of the National Academy of Sciences*, vol. 91, no. 3, pp. 1059–1063, 1994.

[7] A. Bateman, E. Birney, L. Cerruti, R. Durbin, L. Etwiller, S. R. Eddy, S. Griffiths-Jones, K. L. Howe, M. Marshall, and E. L. Sonnhammer, "The Pfam protein families database," *Nucleic Acids Research*, vol. 30, no. 1, pp. 276–280, 2002.

[8] 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.

[9] M. L. Bileschi, D. Belanger, D. H. Bryant, T. Sanderson, B. Carter, D. Sculley, A. Bateman, M. A. DePristo, and L. J. Colwell, "Using deep learning to annotate the protein universe," *Nature Biotechnology*, Feb. 2022.

[10] P. Boufounos, S. El-Difrawy, and D. Ehrlich, "Basecalling using hidden Markov models," *Genomics, Signal Processing, and Statistics*, vol. 341, no. 1, pp. 23–36, Jan. 2004.

[11] Y. Boussemart, J. Las Fargeas, M. L. Cummings, and N. Roy, "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.

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

[13] C. Kern, A. J. González, L. Liao, and K. Vijay-Shanker, "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, Sep. 2013.

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

[15] H. Cheng, G. T. Concepcion, X. Feng, H. Zhang, and H. Li, "Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm," *Nature Methods*, vol. 18, no. 2, pp. 170–175, Feb. 2021.

[16] C.-S. Chin, D. H. Alexander, P. Marks, A. A. Klammer, J. Drake, C. Heiner, A. Clum, A. Copeland, J. Huddleston, E. E. Eichler, S. W. Turner, and J. Korlach, "Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data," *Nature Methods*, vol. 10, no. 6, pp. 563–569, Jun. 2013.

[17] B. Chowdhury and G. Garai, "A review on multiple sequence alignment from the perspective of genetic algorithm," *Genomics*, vol. 109, no. 5-6, pp. 419–431, 2017.

[18] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Norion, A. Scibisz, S. Subramoneyon, C. Alkan, S. Ghose, and O. Mutlu, "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)*, Oct. 2020, pp. 951–966.

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

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

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

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

[23] 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.

[24] B. Ekim, B. Berger, and R. Chikhi, "Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer," *Cell Systems*, vol. 12, no. 10, pp. 958–968.e6, Oct. 2021.

[25] C. Firtina, Z. Bar-Joseph, C. Alkan, and A. E. Cicek, "Hercules: a profile HMM-based hybrid error correction algorithm for long reads," *Nucleic Acids Research*, vol. 46, no. 21, pp. e125–e125, 2018.

[26] C. Firtina, J. S. Kim, M. Alser, D. Senol Cali, A. E. Cicek, C. Alkan, and O. Mutlu, "Apollo: a sequencing-technology-independent, scalable and accurate assembly polishing algorithm," *Bioinformatics*, vol. 36, no. 12, pp. 3669–3679, 2020.

[27] C. Firtina, J. Park, J. S. Kim, M. Alser, D. S. Cali, T. Shahroodi, N. M. Ghiasi, G. Singh, K. Kanellopoulos, C. Alkan, and O. Mutlu, "BLEND: A Fast, Memory-Efficient, and Accurate Mechanism to Find Fuzzy Seed Matches," 2021. [Online]. Available: https://arxiv.org/abs/2112.08687

[28] T. Friedrich, B. Pils, T. Dandekar, J. Schultz, and T. Müller, "Modelling interaction sites in protein domains with interaction profile hidden Markov models," *Bioinformatics*, vol. 22, no. 23, pp. 2851–2857, Dec. 2006.

[29] G. Malysa, D. Wang, L. Netsch, and M. Ali, "Hidden Markov model-based gesture recognition with FMCW radar," in *2016 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, Dec. 2016, pp. 1017–1021.

[30] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A Call Graph Execution Profiler," *SIGPLAN Not.*, vol. 39, no. 4, pp. 49–57, Apr. 2004.

[31] J. Hu, J. Fan, Z. Sun, and S. Liu, "NextPolish: a fast and efficient genome polishing tool for long-read assembly," *Bioinformatics*, vol. 36, no. 7, pp. 2253–2255, Apr. 2020.

[32] N. Huang, F. Nie, P. Ni, F. Luo, X. Gao, and J. Wang, "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.

[33] S. Huang, G. J. Manikandan, A. Ramachandran, K. Rupnow, W.-m. W. Hwu, and D. Chen, "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*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 275–284.

[34] 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.

[35] 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*, Mar. 2010, pp. 168–172.

[36] A. Ibrahim, H. Elsimary, A. Aljumah, and F. Gebali, "Reconfigurable Hardware Accelerator for Profile Hidden Markov Models," *Arabian Journal for Science and Engineering*, vol. 41, no. 8, pp. 3267–3277, 2016.

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

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

[39] K. Liang, X. Wang, and D. Anastassiou, "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, Sep. 2007.

[40] R. Y. Kahsay, G. Wang, G. Gao, L. Liao, and R. Dunbrack, "Quasi-consensus-based comparison of profile hidden Markov models for protein sequences," *Bioinformatics*, vol. 21, no. 10, pp. 2287–2293, May 2005.

[41] M. Kang, J. Ahn, and K. Lee, "Opinion mining using ensemble text hidden Markov models for text classification," *Expert Systems with Applications*, vol. 94, pp. 218–227, Mar. 2018.

[42] J. S. Kim, C. Firtina, M. B. Cavlak, D. S. Cali, M. Alser, N. Hajinazar, C. Alkan, and O. Mutlu, "AirLift: A Fast and Comprehensive Technique for Remapping Alignments between Reference Genomes," 2019. [Online]. Available: https://arxiv.org/abs/1912.08735

[43] M. Kolmogorov, J. Yuan, Y. Lin, and P. A. Pevzner, "Assembly of long, error-prone reads using repeat graphs," *Nature Biotechnology*, vol. 37, no. 5, pp. 540–546, May 2019.

[44] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, "Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation," *Genome Research*, vol. 27, no. 5, pp. 722–736, May 2017.

[45] L. Li, Y. Zhao, D. Jiang, Y. Zhang, F. Wang, I. Gonzalez, E. Valentin, and H. Sahli, "Hybrid Deep Neural Network–Hidden Markov Model (DNN-HMM) Based Speech Emotion Recognition," in *2013 Humaine Association Conference on Affective Computing and Intelligent Interaction*, Sep. 2013, pp. 312–317.

[46] H. Lanyue, C. Jianhua, W. Rongshu, L. Zhiwen, and H. Bin, "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.

[47] N. LaPierre, M. Alser, E. Eskin, D. Koslicki, and S. Mangul, "Metalign: efficient alignment-based metagenomic profiling via containment min hash," *Genome biology*, vol. 21, no. 1, pp. 1–15, 2020.

[48] N. LaPierre, S. Mangul, M. Alser, I. Mandric, N. C. Wu, D. Koslicki, and E. Eskin, "MiCoP: microbial community profiling method for detecting viral and fungal organisms in metagenomic samples," *BMC genomics*, vol. 20, no. 5, pp. 1–10, 2019.

[49] S. J. Lewis, A. Raval, and J. E. Angus, "Bayesian Monte Carlo estimation for profile hidden Markov models," *Mathematical and Computer Modelling*, vol. 47, no. 11, pp. 1198–1216, 2008.

[50] 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. [Online]. Available: https://doi.org/10.1093/bioinformatics/btw152

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

[52] 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.

[53] X. Liu, Z. Zhuo, X. Du, X. Zhang, Q. Zhu, and M. Guizani, "Adversarial attacks against profile HMM website fingerprinting detection model," *Cognitive Systems Research*, vol. 54, pp. 83–89, May 2019.

[54] 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.

[55] M. Hamidi, H. Satori, O. Zealouk, K. Satori, and N. Laaidi, "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)*, Oct. 2018, pp. 16–21.

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

[57] N. Mansouri Ghiasi, J. Park, H. Mustafa, J. Kim, A. Olgun, A. Gollwitzer, D. Senol Cali, C. Firtina, H. Mao, N. Almadhoun Alserr, R. Ausavarungnirun, N. Vijaykumar, M. Alser, and O. Mutlu, "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.

[58] J. Mistry, S. Chuguransky, L. Williams, M. Qureshi, G. A. Salazar, E. L. Sonnhammer, S. C. Tosatto, L. Paladin, S. Raj, L. J. Richardson *et al.*, "Pfam: The protein families database in 2021," *Nucleic Acids Research*, vol. 49, no. D1, pp. D412–D419, 2021.

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

[60] B. Mor, S. Garhwal, and A. Kumar, "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.

[61] B. S. Moreira, A. Perkusich, and S. O. D. Luiz, "An Acoustic Sensing Gesture Recognition System Design Based on a Hidden Markov Model," *Sensors*, vol. 20, no. 17, 2020.

[62] N. J. Mulder and R. Apweiler, "Tools and resources for identifying protein families, domains and motifs," *Genome biology*, vol. 3, no. 1, pp. 1–8, 2001.

[63] A. Nag, C. N. Ramachandra, R. Balasubramonian, R. Stutsman, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon, "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.

[64] N. Nguyen-Duc-Thanh, S. Lee, and D. Kim, "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.

[65] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "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, Mar. 2008.

[66] S. Nurk, B. P. Walenz, A. Rhie, M. R. Vollger, G. A. Logsdon, R. Grothe, K. H. Miga, E. E. Eichler, A. M. Phillippy, and S. Koren, "HiCanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads," *Genome Research*, vol. 30, no. 9, pp. 1291–1305, Sep. 2020.

[67] S. Pattabiraman and T. Warnow, "Profile Hidden Markov Models Are Not Identifiable," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 18, no. 1, pp. 162–172, 2021.

[68] M. Pietras and P. Klesk, "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. 6, pp. 935–946, 2017.

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

[70] S. Ren, V. Sima, and Z. Al-Ars, "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.

[71] V. Rezaei, H. Pezeshk, and H. Pérez-Sa'nchez, "Generalized Baum-Welch Algorithm Based on the Similarity between Sequences," *PLOS ONE*, vol. 8, no. 12, p. e80565, Dec. 2013.

[72] A. B. Riddell, "Reliable Editions from Unreliable Components: Estimating Ebooks from Print Editions Using Profile Hidden Markov Models," 2022. [Online]. Available: https://arxiv.org/abs/2204.01638

[73] S. Angizi, J. Sun, W. Zhang, and D. Fan, "PIM-Aligner: A Processing-in-MRAM Platform for Biological Sequence Alignment," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2020, pp. 1265–1270.

[74] S. D. Goenka, Y. Turakhia, B. Paten, and M. Horowitz, "SegAlign: A Scalable GPU-Based Whole Genome Aligner," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2020, pp. 1–13.

[75] 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.

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

[77] S. Seo, M. Oh, Y. Park, and S. Kim, "DeepFam: deep learning based alignment-free method for protein family modeling and prediction," *Bioinformatics*, vol. 34, no. 13, pp. i254–i262, Jul. 2018.

[78] N. G. Sgourakis, P. G. Bagos, P. K. Papasaikas, and S. J. Hamodrakas, "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.

[79] G. Singh, M. Alser, D. Senol Cali, D. Diamantopoulos, L. Gómez-Luna, H. Corporaal, and O. Mutlu, "FPGA-Based Near-Memory Acceleration of Modern Data-Intensive Applications," *IEEE Micro*, vol. 41, no. 4, pp. 39–48, Aug. 2021.

[80] K. Sinha, R. Kumari, A. Priya, and P. Paul, "A Computer Vision-Based Gesture Recognition Using Hidden Markov Model," in *Innovations in Soft Computing and Information Technology*, J. Chattopadhyay, R. Singh, and V. Bhattacherjee, Eds. Singapore: Springer Singapore, 2019, pp. 55–67.

[81] P. Skewes-Cox, T. J. Sharpton, K. S. Pollard, and J. L. DeRisi, "Profile hidden Markov models for the detection of viruses within metagenomic sequence data," *PloS one*, vol. 9, no. 8, p. e105067, 2014.

[82] S. Soiman, I. Rusu, and S. Pentiuc, "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.

[83] M. Steinegger, M. Meier, M. Mirdita, H. Vöhringer, S. J. Haunsberger, and J. Söding, "HH-suite3 for fast remote homology detection and deep protein annotation," *BMC Bioinformatics*, vol. 20, no. 1, p. 473, 2019.

[84] 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.

[85] Y. Turakhia, G. Bejerano, and W. J. Dally, "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.

[86] 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.

[87] R. Vaser, I. Sović, N. Nagarajan, and M. Šikić, "Fast and accurate de novo genome assembly from long uncorrected reads," *Genome Research*, vol. 27, no. 5, pp. 737–746, May 2017.

[88] R. Vicedomini, J. Bouly, E. Laine, A. Falciatore, and A. Carbone, "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.

[89] A. S. Vieira, E. L. Iglesias, and L. Borrajo, "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, M. P. Rocha, F. Fdez-Riverola, and J. F. De Paz Santana, Eds. Cham: Springer International Publishing, 2014, pp. 225–234.

[90] 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.

[91] B. J. Walker, T. Abeel, T. Shea, M. Priest, A. Abouelliel, S. Sakthikumar, C. A. Cuomo, Q. Zeng, J. Wortman, S. K. Young, and A. M. Earl, "Pilon: An Integrated Tool for Comprehensive Microbial Variant Detection and Genome Assembly Improvement," *PLOS ONE*, vol. 9, no. 11, p. e112963, Nov. 2014.

[92] L. Wang and T. Jiang, "On the complexity of multiple sequence alignment," *Journal of computational biology*, vol. 1, no. 4, pp. 337–348, 1994.

[93] T. J. Wheeler, J. Clements, S. R. Eddy, R. Hubley, T. A. Jones, J. Jurka, A. F. A. Smit, and R. D. Finn, "Dfam: a database of repetitive DNA based on profile hidden Markov models," *Nucleic Acids Research*, vol. 41, no. D1, pp. D70–D82, 2012.

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

[95] L. Yu, Y. Ukidave, and D. Kaeli, "GPU-Accelerated HMM for Speech Recognition," in *2014 43rd International Conference on Parallel Processing Workshops*, 2014, pp. 395–402.

[96] Q. Zhan, N. Wang, S. Jin, R. Tan, Q. Jiang, and Y. Wang, "ProbPFP: a multiple sequence alignment algorithm combining partition function and hidden markov model with particle swarm optimization," in *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2018, pp. 1290–1295.

[97] 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.

[98] 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.