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

Originally published in:
ACM Transactions on Architecture and Code Optimization 21(1), https://doi.org/10.1145/3632950

Funding acknowledgement:
213084 - Near-Data-Processing Architectures and Algorithms for Metagenomic Analysis (SNF)

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

CAN FIRTINA, ETH Zurich, Switzerland
KAMLESH PILLAI, GURPREET S. KALSI, and BHARATHWAJ SURESH, Intel Labs, USA
DAMLA SENOL CALI, Carnegie Mellon University, USA
JEREMIE S. KIM, ETH Zurich, Switzerland
TAHA SHAHROODI, TU Delft, Netherlands
MERYEM BANU CAVLAK, JOËL LINDEGGER, MOHAMMED ALSER, and JUAN GÓMEZ LUNA, ETH Zurich, Switzerland
SREENIVAS SUBRAMONEY, Intel Labs, USA
ONUR MUTLU, ETH Zurich, Switzerland

Profile hidden Markov models (pHMMs) are widely employed in various bioinformatics applications to identify similarities between biological sequences, such as DNA or protein sequences. In pHMMs, sequences are represented as graph structures, where states and edges capture modifications (i.e., insertions, deletions, and substitutions) by assigning probabilities to them. These probabilities are subsequently used to compute the similarity score between a sequence and a pHMM graph. The Baum-Welch algorithm, a prevalent and highly accurate method, utilizes these probabilities to optimize and compute similarity scores. Accurate computation of these probabilities is essential for the correct identification of sequence similarities. However, the Baum-Welch algorithm is computationally intensive, and existing solutions offer either software-only or hardware-only approaches with fixed pHMM designs. When we analyze state-of-the-art works, we identify an urgent need for a flexible, high-performance, and energy-efficient hardware-software co-design to address the major inefficiencies in the Baum-Welch algorithm for pHMMs.

We introduce *ApHMM*, the *first* flexible acceleration framework designed to significantly reduce both computational and energy overheads associated with the Baum-Welch algorithm for pHMMs. ApHMM employs hardware-software co-design to tackle the major inefficiencies in the Baum-Welch algorithm by (1) designing flexible hardware to accommodate various pHMM designs, (2) exploiting predictable data dependency patterns through on-chip memory with memoization techniques, (3) rapidly filtering out unnecessary computations using a hardware-based filter, and (4) minimizing redundant computations.

We acknowledge the generous gifts and support provided by our industrial partners: Intel, Google, Huawei, Microsoft, VMware, and the Semiconductor Research Corporation. This work is also partially supported by the European Union's Horizon programme for research and innovation (Grant No. 101047160-BioPIM) and the Swiss National Science Foundation (SNSF) (Grant No. 200021_213084).

Authors' addresses: C. Firtina, J. S. Kim, M. Banu Cavlak, J. Lindegger, M. Alser, J. Gómez Luna, and O. Mutlu, ETH Zurich, Switzerland; e-mails: {can.firtina, jeremie.kim}@safari.ethz.ch, bcavlak@ethz.ch, {joel.lindegger, mohammed.alser, juan.gomez, onur.mutlu}@safari.ethz.ch; K. Pillai, G. S. Kalsi, B. Suresh, and S. Subramoney, Intel Labs, India; e-mails: {kamlesh.r.pillai, gurpreet.s.kalsi, bharathwaj.suresh, sreenivas.subramoney}@intel.com; D. Senol Cali, Carnegie Mellon University, USA; e-mail: dsenol@andrew.cmu.edu; T. Shahroodi, TU Delft, Netherlands; e-mail: t.shahroodi@tudelft.nl.

ACM 1544-3566/2024/02-ART19
https://doi.org/10.1145/3632950

ApHMM achieves substantial speedups of 15.55×–260.03×, 1.83×–5.34×, and 27.97× when compared to CPU, GPU, and FPGA implementations of the Baum-Welch algorithm, respectively. ApHMM outperforms state-of-the-art CPU implementations in three key bioinformatics applications: (1) error correction, (2) protein family search, and (3) multiple sequence alignment, by 1.29×–59.94×, 1.03×–1.75×, and 1.03×–1.95×, respectively, while improving their energy efficiency by 64.24×–115.46×, 1.75×, and 1.96×.

CCS Concepts: • **Computer systems organization → Special purpose systems**; • **Applied computing → Computational biology**; **Genomics**; **Bioinformatics**; • **Theory of computation → Markov decision processes;**

Additional Key Words and Phrases: Bioinformatics, genomics, profile hidden markov models, the Baum-Welch Algorithm

ACM Reference Format:
Can Firtina, Kamlesh Pillai, Gurpreet S. Kalsi, Bharathwaj Suresh, Damla Senol Cali, Jeremie S. Kim, Taha Shahroodi, Meryem Banu Cavlak, Joël Lindegger, Mohammed Alser, Juan Gómez Luna, Sreenivas Subramoney, and Onur Mutlu. 2024. ApHMM: Accelerating Profile Hidden Markov Models for Fast and Energy-efficient Genome Analysis. *ACM Trans. Arch. Code Optim.* 21, 1, Article 19 (February 2024), 29 pages. https://doi.org/10.1145/3632950

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) [1]. 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 when visiting a state from another. HMMs are very efficient in decoding the continuous and discrete series of events in many applications [2] such as speech recognition [2–8], text classification [9–13], gesture recognition [14–21], and bioinformatics [22–45]. 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 [44], which is commonly adopted in bioinformatics [23–25, 27, 32–35, 37–39, 41–43, 45], malware detection [46–51], and pattern matching [52–57].

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) [58–80]. 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 [58, 67, 68, 70]. Figure 1 illustrates a *traditional* pHMMs design. A pHMM represents a single (or many) sequence(s) 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 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 [25, 37, 41, 42, 45, 81–83], **multiple sequence alignment (MSA)** [25, 32, 33, 35, 38, 39, 43, 51, 84–86], and error correction [23, 24, 27].

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 accurately fit the observed biological sequences to the pHMM graph. Probabilities are adjusted

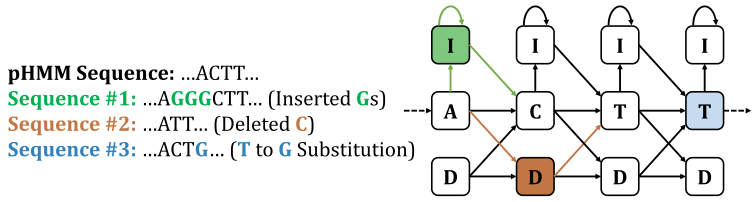

Fig. 1. Portion of an example pHMM design that represents a DNA sequence (pHMM Sequence). Differences between pHMM Sequence and Sequences #1, #2, and #3 are highlighted with their corresponding colors. Highlighted transitions and states identify each corresponding difference. The states with DNA characters correspond to match or mismatch (M) states, while I and D states correspond to insertion and deletion, respectively.

during the *training* 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 maximization in pHMMs [87–90]. The Baum-Welch algorithm [89] is commonly used to calculate likelihood maximization [91] as it is highly accurate and scalable to real-size problems (e.g., large protein families) [88]. 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 [92, 93] due to the nature of its dynamic programming approach. Several works [33, 94–97] 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 of executing the Baum-Welch algorithm, they either (1) provide software-only 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 causes significant computational overhead 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) A significant fraction of floating-point operations is redundant in the training step due to the lack of a mechanism for reusing the same operations. (4) Existing strategies for filtering out the unnecessary (i.e., 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 main source of computational overhead even for the non-genomic application we evaluate (Section 3). Our observations demonstrate 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 and can use different pHMM designs to change certain parameter choices to enable the adoption of ApHMM for many pHMM-based applications. This enables, first, additional support for pHMM-based error correction [23, 24, 27]

that traditional pHMM design cannot efficiently and accurately support [27]. Second, ApHMM exploits the spatial locality that pHMMs provide with the Baum-Welch algorithm by efficiently utilizing on-chip memories with memoization techniques. Third, ApHMM efficiently eliminates unnecessary 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 via **lookup tables (LUTs)** and (2) identifying pipelining and broadcasting opportunities where results from certain operations are used in multiple steps in the Baum-Welch algorithm without additional storage or computational overheads.

To evaluate ApHMM, we (1) design a flexible hardware-software co-designed accelerator and (2) implement our software optimizations on 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 three 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×, and 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 for at least 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 new mechanisms with efficient hardware and software optimizations to significantly reduce the computational and energy overheads 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)

High-level Overview. Figure 1 shows the traditional structure of pHMMs. pHMMs represent a sequence or a group of sequences using a certain graph structure with a fixed number of nodes for every character of represented sequences. Visiting nodes, called *states*, via directed edges, called *transitions*, are associated with probabilities to identify differences at any position between the represented sequences and other sequences. States 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 sequences and allow making modifications to these sequences. We explain the detailed structure of pHMMs in Supplemental Section 1.

2.2 Baum-Welch Algorithm

The probabilities associated with transitions and states are essential for identifying similarities between sequences. The Baum-Welch algorithm provides a set of equations to update and use these probabilities accurately. To calculate the similarity score of input observation sequences in a pHMM graph, the Baum-Welch algorithm [89] solves an expectation-maximization problem [98–101], 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.

Forward Calculation. The goal of the forward calculation is to compute the probability of observing sequence S when we compare it with the sequence S_G that the pHMM graph $G(V, A)$ represents. 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 $S[t]$ at position t of S 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 article. To represent transition and emission probabilities, we use the α_{ji} and $e_{S[t]}(v_i)$ notations as we define in Supplemental Section **??**:

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

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:

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

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 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:

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

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

2.3 Use Cases of Profile HMMs

Error Correction. The goal of error correction is to locate the erroneous parts in DNA or genome sequences and replace these parts with more reliable sequences [102–107] to enable more accurate genome analysis (e.g., read mapping and genome assembly). Apollo [24] 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

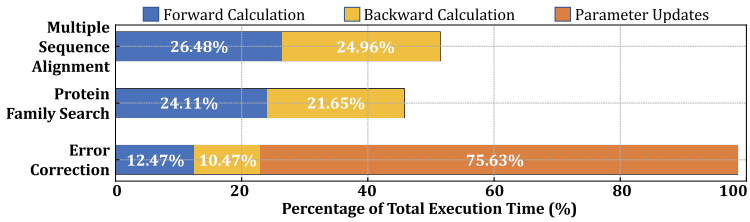

Fig. 2. Percentage of the total execution time for the three steps of the Baum-Welch algorithm

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 [108] 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 limitations associated with traditional pHMMs when generating the consensus sequences [92, 93]. 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 [23, 24, 27] to construct more accurate consensus sequences from pHMMs.

Protein Family Search. Classifying protein sequences into families is widely used to analyze the potential functions of the proteins of interest [109–114]. The 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 [115], where the HMMER [33] 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 [116].

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 [117, 118]. To mitigate these computational problems, heuristic algorithms are used to obtain an approximate yet computationally efficient solution for multiple alignments of genomic sequences. pHMM-based approaches provide an efficient solution for MSA [119]. The pHMM approaches, such as *hmmalign* [33], 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 [120]). pHMM-based MSA approaches are mainly useful to avoid making redundant comparisons as a sequence can be compared to a pHMM graph, similar to the protein family search.

3 MOTIVATION AND GOAL

3.1 Sources of Inefficiencies

To identify and understand the performance overheads of state-of-the-art pHMM-based applications, we thoroughly analyze existing tools for the three use cases of pHMM: (1) Apollo [24] for error correction, (2) hmmsearch [33] for protein family search, and (3) hmmalign [33] for MSA. We make six key observations based on our profiling with Intel VTune [121] and gprof [122].

Observation 1: The Baum-Welch Algorithm causes significant computational overhead. 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 causes significant performance overhead 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 the 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 part of inference. We do *not* include the cost of training for these applications as it is executed only once or a few times, such that the cost of training is 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 substantial for this application. As a result, accelerating the entire Baum-Welch algorithm is key for improving the end-to-end performance of the applications.

Observation 2: SIMD-based tools on CPU and GPUs provide suboptimal vectorization. The Baum-Welch algorithm involves frequent floating-point multiplications and additions. To resolve performance issues, several CPU-based tools (e.g., hmmalign) use SIMD instructions. However, these tools exhibit poor SIMD utilization due to inadequate port utilization and low vector capacity usage (below 50%). This suggests that CPU-based optimizations for floating-point operations, such as SIMD instructions, provide limited computational benefits for the Baum-Welch algorithm. We further investigate if the SIMD utilization in GPUs exhibits similar low utilization. To observe this, we profile our GPU work, ApHMM-GPU, to execute the two main kernels in the application: Forward and Backward calculations. We observe that the Forward calculation suffers from low SIMD utilization (i.e., percentage of active threads per warp) of around 50%, while the SIMD utilization of Backward calculation is usually close to 100%. The GPU implementation iterates over all the states that have a connection to the state that the thread is working on. However, the number of states to iterate can substantially be different per thread during Forward calculation as insertion and match states have a largely different number of *incoming states*, which is not the case in Backward calculation. This imbalance causes high warp divergence during Forward calculation, reducing the SIMD utilization.

Observation 3: A 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 with 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 [24, 27, 123–127]. A simple approach is to pick the `best-n` states that provide the highest scores in each iteration while the rest of the states are ignored in the next iteration, known as filtering [27]. Figure 3 shows the relation between the filter size (i.e., the number of states picked as `best-n` states), the overall runtime of the Baum-Welch algorithm, and its corresponding 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 substantial performance costs by constituting around 8.5% of the overall execution time in the training step.

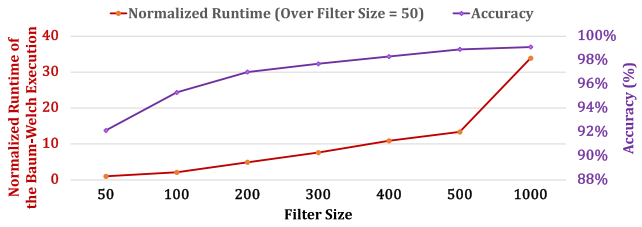

Fig. 3. Effect of the filter size on the runtime and the accuracy of the Baum-Welch algorithm

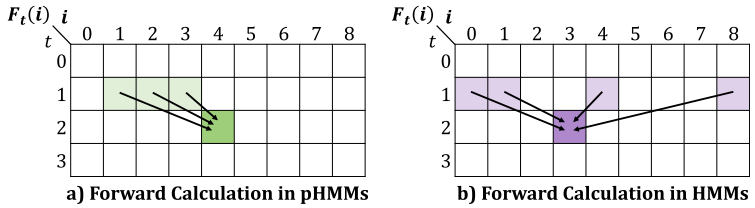

Fig. 4. Data dependency in pHMMs and HMMs

Observation 5: HMM accelerators are suboptimal for accelerating pHMMs. Generic HMMs do not require 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 patterns 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 locality (e.g., only the values from the previous iteration are used), pHMMs provide better spatial locality with their constrained design. This observation suggests that HMM-based accelerators cannot fully exploit the spatial locality that pHMMs provide as they are oblivious to the design of pHMMs.

Observation 6: Non-genomics pHMM-based applications also suffer from the computational overhead of the Baum-Welch algorithm. Among many non-genomics pHMM-based implementations [46–57], we analyze the available CPU implementation of a recent pattern-matching application that uses pHMMs [52]. 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 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 [23–25, 27, 33, 37, 38, 40–42, 46, 47, 50–53, 55, 128]. However, due to computational inefficiencies, the Baum-Welch algorithm can result in significant performance overheads on these applications. Solving the inefficiencies in the Baum-Welch algorithm is mainly important for services that frequently use these applications, such as the EBI website using HMMER for searching protein sequences in protein databases [129]. 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) [130]. 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.

While the Baum-Welch algorithm is computationally intensive and can consume a significant portion of the runtime and energy in various applications, these applications are often run multiple times as part of routine analyses or when new data becomes available. For error

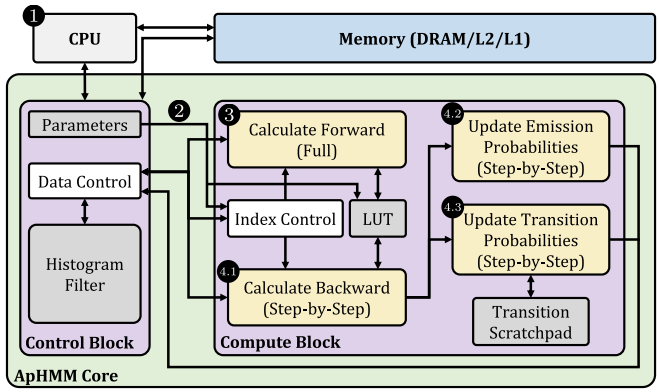

Fig. 5. Overview of ApHMM

correction, the assembly of a particular genome can be reconstructed and corrected multiple times if additional sequencing data for the genome becomes available or if the tools used in the assembly construction pipeline are updated or replaced. For the protein family search and the multiple sequence alignment, protein sequencing data is frequently used multiple times due to regular updates in databases like the Pfam database [115, 131]. These updates can generate new insights [132], such as more accurate reannotation of genes in assemblies [133]. This frequent use of sequenced data can make using the Baum-Welch algorithm a time and energy-consuming process in the overall sequencing data analysis pipeline. Improving the efficiency of the Baum-Welch algorithm can significantly reduce both the compute cycles and energy consumption, especially in use cases where sequencing data is used multiple times.

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 spatiotemporal locality in on-chip memory, and ❹ perform 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 reduce computational and energy overheads of 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-performance, 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 L2/L1 cache. 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.

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) ❹.1. ApHMM updates emission ❹.2 and transition probabilities ❹.3 as the Backward values are calculated in each timestamp.

4.2 Control Block

Control Block 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 the sequence S). Figure 5 shows three main units in Control Block: (1) `Parameters`, (2) `Data Control`, and (3) `Histogram Filter`.

`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 cache 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 beneficial for eliminating unnecessary (i.e., negligible) states from Forward and Backward calculations without significantly compromising accuracy (Section 3). The **challenge** in implementing a straightforward filtering mechanism lies in performing sorting in hardware, which is difficult to achieve efficiently. Our **key idea** is to replace the sorting mechanism with a histogram-based filter, allowing values to be placed into different bins based on their Forward or Backward values. This offers quick and approximate identification of necessary states (i.e., states with the best values until the filter is full) based on their bin locations. To enable such a binning mechanism, we employ a **flexible** *histogram-based* filtering mechanism in the ApHMM on-chip memory.

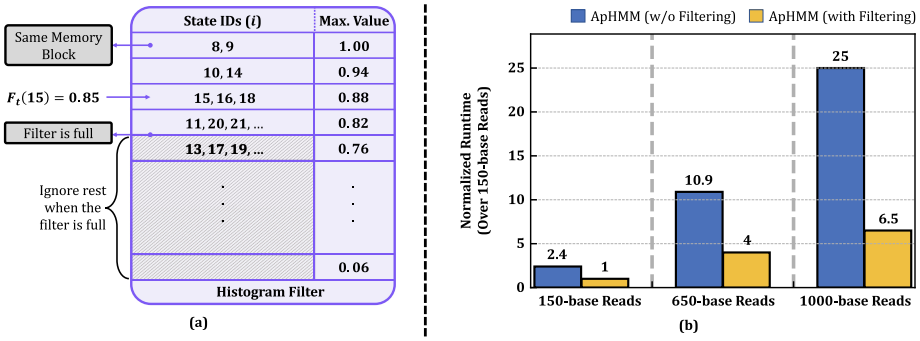

Fig. 6. (a) Overall structure of a `Histogram Filter`. (b) Effect of the `Histogram Filter` approach in ApHMM for different sequence lengths.

Figure 6(a) shows the overall structure of our `Histogram Filter`. The filter categorizes states into bins based on their Forward or Backward values at the current execution timestamp in three steps. First, `Histogram Filter` divides the $[0, 1]$ range into n bins, with each bin corresponding to a specific range of Forward or Backward values. The range for each bin is $1/n$. We empirically chose 16 bins, ensuring a range of $1/16 = 0.0625$, to maintain the same accuracy when the filter size is 500 (Figure 3). For simplicity, we use 0.06 as the range value in Figure 6(a), with the maximum value in each bin's range displayed under *Max. Value*.

Second, the `Histogram Filter` assigns addresses to states such that all states with Forward or Backward values within the same range fall into the same memory block. This addressing mechanism employs a *base and offset* strategy, where the base represents the start of the memory block for a specific range of values, and the offset is the pointer to the next free memory region within the memory block. This base and offset strategy allows ApHMM to discard unnecessary states efficiently, as their addresses are known without sorting.

Third, to identify the addresses of unnecessary states, the `Histogram Filter` accumulates the count of states in each bin, starting with the bin with the largest *Max. Value* (i.e., 1.00). When the overall state count exceeds the filter size (e.g., 500), the remaining bins are assumed to contain only unnecessary states. The `Histogram Filter` can find all the necessary states that a filtering technique with a sorting mechanism finds, albeit with the cost of including states beyond the predetermined filter size, as the accumulated state count in the last bin can exceed the filter count. While it is possible to perform additional computations in the last bin to prevent exceeding the filter size, we leave such optimization for future work.

To build a **flexible** framework for various applications, The microarchitecture is configurable to vary the number of bins (n) based on the application and the average sequence length. We recommend conducting an empirical analysis before determining this range for a particular application, as it may vary significantly.

ApHMM offers the option to disable the filtering mechanism if the application does not necessitate a filter operation for more optimal computations. Figure 6(b) shows the normalized runtime of ApHMM with and without using a filter for varying sequence lengths. We observe that the performance significantly improves when the filtering mechanism is enabled, especially for longer sequences. This can be primarily attributed to the fact that the number of states requiring processing at the subsequent timestamp can exponentially increase, as each state typically has more than one transition, potentially leading to an exponential increase in states at each subsequent timestamp. As the sequence length grows, such an exponential increase can adversely affect the application, which can be significantly mitigated without compromising accuracy through a filtering approach.

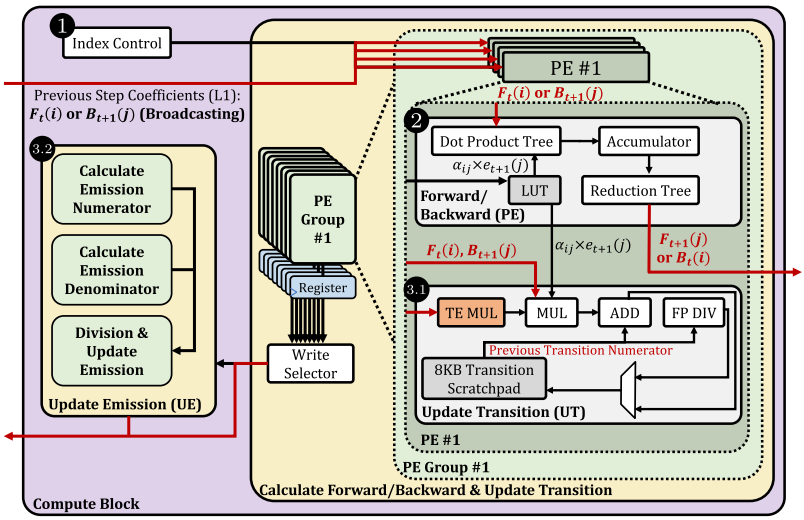

Fig. 7. Overview of a Compute Block. Red arrows show on- and off-chip memory requests.

4.3 Compute Block

Figure 7 shows the overall structure of a Compute Block, which 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)) ❸.1, and (2) a unit for updating the emission probabilities (Equation (4)) ❸.2. Each unit performs the corresponding calculations in the Baum-Welch algorithm.

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 into 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 discuss this trade-off between the number of PEs and memory bandwidth in Section 4.4. To efficiently calculate 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 (TE MUL unit in Figure 7).

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 7. Each UT efficiently calculates the denominator and numerator in Equation (3) for a state v_i. UTs include three key mechanisms.

First, to enable efficient broadcasting of common values between the 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. The incoming red arrows in Figure 7 show the flow of these Forward and Backward values in PEs and UTs. The calculation of $F_{t+1}(j)$ involves a summation over all states i as shown in Equation (1). The $F_t(i)$ term is common to the calculation of $F_{t+1}(j)$ for all states j and hence can be broadcasted. Similarly, the calculation of $B_t(i)$ involves a summation over all states j (Equation (2)). The $B_{t+1}(j)$ term is common to the calculation of $B_t(i)$ for all states i and hence can be broadcasted. This **key design choice** exploits the broadcast opportunities available within the common multiplications in the Baum-Welch equations.

Second, ApHMM cores are designed to directly consume the broadcasted Backward values when updating the emission and transition probabilities 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 directly consumed when updating the transitions and emission probabilities while the Backward values are broadcasted in the current timestamp. ApHMM updates emission and transition probabilities step-by-step as Backward values are calculated, a hardware-software optimization we call the *partial compute approach*. It is worth noting that ApHMM fully computes and stores the Forward values before updating the emission and transition probabilities. **The key benefits** of combining broadcasting with the partial compute approach are (1) decoupling hardware scaling from bandwidth requirements and (2) reducing the bandwidth requirement by 4× (i.e., 32 bits/cycle instead of 128 bits/cycle).

Third, to exploit the spatiotemporal 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 to reduce the data movement overhead within ApHMM. To this end, we use an 8 KB 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 8 KB 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, 8 KB 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.

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 7, 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 7.

An ApHMM core writes and reads both the numerator and denominator values to L1 cache 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 cache 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), then 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 Hardware Configuration Choice

Our goal is to identify the ideal number of memory ports and PEs 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). Since these ports are shared across each PE, 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 due to the broadcasting feature of ApHMM (Section 4.3). Instead, computing these steps depends on (1) memory bandwidth per port, which determines the number of multiplications and accumulations in parallel in a PE, and (2) the number of processing engines (PEs). We conclude that the overall requirement for a ApHMM core is eight memory ports with the same bandwidth per port.

Figure 8 shows the scaling capabilities of ApHMM with the number of PEs and sequence length to decide (1) the overall number of PEs and (2) the longest chunk size for the best acceleration. First, to decide the overall number of PEs to use in ApHMM, in Figure 8(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. 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

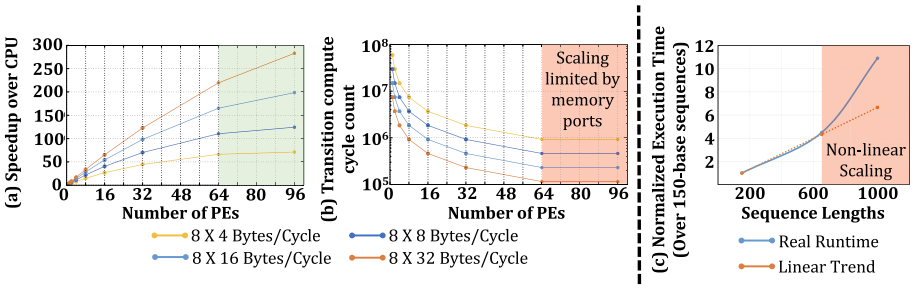

Fig. 8. (a) Acceleration scaling with the number of PEs. (b) Compute cycle acceleration when calculating the transition probabilities with the increased number of PEs. (c) Increase in the execution time with respect to the sequence (chunk) lengths. The data points are for sequence lengths 150, 650, and 1,000, respectively. The linear trend shows the expected linear increase in execution time, and the real runtime shows the actual runtimes.

declining. We explore the reason for such a trend in Figure 8(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. Second, We conclude that the acceleration trend we observe in Figure 8(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.

In our design, the choice of memory bandwidth influences the number of PE Groups and PEs, given a constant number of memory ports. While our hardware can scale to accommodate higher bandwidths, we opt for a 16-bytes/cycle bandwidth. This design choice aligns with the 128-bit line size of our L1 cache, allowing us to operate on four single-precision floating-point values (32-bit) across 4 PEs simultaneously. To fully utilize all 64 PEs, as discussed earlier, we employ 16 PE Groups (64 PEs = 4 PEs $\times$ 16 PE Groups).

Second, to identify the optimal chunk size (i.e., sequence length) that ensures a near-linear increase in execution time with increasing sequence length, we examine the execution time of the Baum-Welch algorithm for chunk sizes of 150, 650, and 1,000 bases, as shown in Figure 8(c). We observe a linear increase in execution time with chunk sizes up to approximately 650 bases. Beyond this point, the execution time begins to increase non-linearly. This non-linear scaling is primarily due to the increased cache space requirements for storing certain parameters (e.g., emission values), as shown in Supplemental Figure S1. This increased cache pressure leads to more data accesses from the upper levels of the memory hierarchy. ApHMM can maintain a linear trend in execution time for longer sequences by either increasing the L1 and L2 cache capacities or utilizing higher-bandwidth memories to mitigate the data movement overheads. We provide further details regarding the data distribution and memory layout in Supplemental Section S2.

We conclude that the memory ports and chunk size primarily constrain the acceleration speedup of ApHMM, as the PEs start to be underutilized due to increased data movement overheads. To further enhance the acceleration with ApHMM, optimizing the utilization of PEs by minimizing these overheads is crucial.

Number of ApHMM Cores. We show our methodology for choosing the ideal number of cores in ApHMM for accelerating the applications. Figure 9 shows the speedup of three bioinformatics applications when using single, 2, 4, and 8 cores in ApHMM. We divide the entire execution time of the applications into three stages: (1) the CPU execution of the application that does not use the Baum-Welch execution, (2) the Baum-Welch execution accelerated using ApHMM, and (3) and the overhead mainly caused due to data movements. Our analysis incorporates the estimated off-

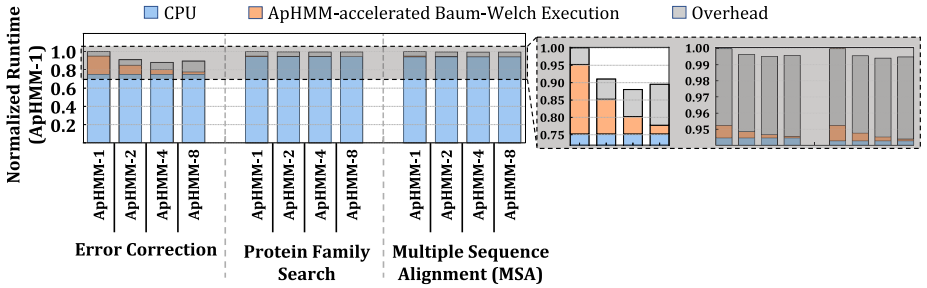

Fig. 9. Normalized runtimes of multi-core ApHMM compared to the single-core ApHMM (ApHMM-1).

Table 1. Microarchitecture Configuration

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

and on-chip data movement overhead. We observe that using four cores in ApHMM provides the best speedup overall. This is because the applications provide smaller rooms for acceleration for two reasons. First, the remaining CPU part of the application becomes the bottleneck in the entire execution of the application due to the significant acceleration of the Baum-Welch execution using ApHMM. Second, the data movement overhead starts causing more latency than the benefits of further accelerating the Baum-Welch algorithm by increasing the number of cores. This suggests ApHMM is bounded by the data movement overhead when scaling it to a larger number of cores, and there is still room for improving the performance of ApHMM by placing ApHMM inside or near the memory (e.g., high-bandwidth memories) to reduce the data movement overheads that limit scaling ApHMM to many cores. Based on our observations, we use a four-core ApHMM to achieve the best overall performance (see Supplemental Section S3 for the execution flow of the system with multiple cores in ApHMM).

5 EVALUATION

We evaluate our acceleration framework, ApHMM, for three use cases: (1) error correction, (2) protein family search, and (3) 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 [134] in a typical 28 nm process technology node at 1 GHz clock frequency with tightly integrated on-chip memory (1 GHz) 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 four ApHMM cores in our evaluation (Section 4.4). We account for an additional 5% of cycles to compensate for the arbitration 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).

We use the CUDA library [135] (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) as 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 [24], 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 [122] to profile the baseline CPU implementations of the use cases on the AMD EPYC 7742 processor (2.26 GHz, 7 nm 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 [24] 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 [24], hmmsearch [33], and hmmalign [33]. 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 [96]. For the FPGA implementation, we use the FPGA **Divide and Conquer (D&C)** accelerator proposed for the Baum-Welch algorithm [95]. When evaluating the FPGA accelerator, we ignore the data movement overhead and estimate the acceleration based on the speedup provided by the earlier work. We acknowledge that the performance and energy comparisons can be attributed to both platform differences and architectural optimizations, especially when comparing ApHMM with the FPGA accelerator. Although our evaluations lack comparisons in the equivalent platforms for FPGAs, we still believe that our evaluations provide valuable insights regarding the benefits of our ASIC implementation compared to the FPGA work.

In terms of accuracy, we ensure the accuracy of our results by faithfully implementing all the equations of the Baum-Welch algorithm and rigorously testing their output during our ASIC design. The only exception is the `Histogram Filter`, where we introduce a binning approach to include all the states a sorting-based software implementation would include, ensuring at least the same minimum accuracy as the original software implementation. Our accuracy evaluation shows that the histogram filter approach usually leads to better accuracy than the sorting approach, with a minimal accuracy difference within a +/−0.2% range. To reproduce the output for comparison purposes, we provide the source code of our software optimizations in the **GPU implementation of ApHMM (ApHMM-GPU)**.

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 with the average read length of 5,128 bases. The accession code of this sample is SAMN06173305. Of 163,482 reads, we randomly select 10,000 sequencing reads for our evaluation. We use minimap2 [136] and miniasm [137] 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 [115] that includes 19,632 pHMMs. To achieve this, the hmmsearch [33] tool performs the Forward and Backward calculations to find similarities between pHMMs and sequences.

To evaluate the 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 [33] tool performs the Forward and Backward calculations to find similarities between a single pHMM graph and sequences.

Data Set Justification. In our study, we carefully chose our datasets for overhead analysis and evaluation. We believe these datasets are comprehensive and relevant to guide our ASIC design and to evaluate ApHMM with other systems for several reasons. First, our datasets cover various use cases with various sequence lengths (i.e., an average read length of 5,168 and an average protein sequence length of 94.2) and alphabet sizes (4 in DNA and 20 in proteins). This diversity ensures that our results are not skewed toward a specific use case or dataset.

Second, for error correction, we use a real-world sample of the *E. coli* genome, a commonly studied bacterial genome. The overall length of randomly selected 10,000 *E. coli* reads is around 50,000,000 bases (the average length of a single read is 5,168). This ensures that these reads cover the entire *E. coli* genome around 10 times (i.e., 10× depth of coverage), which ensures that the Baum-Welch algorithm is executed by performing error correction on the entire genome multiple times without focusing on the specific regions of the genome to avoid potential bias that can be caused on particular regions. For the multiple sequence alignment and the protein family search, we use the most commonly studied protein families as these protein families are among the top 20 families with the largest number of protein sequence alignments,[1] ensuring the relevance and applicability of our work. The bottleneck analysis was conducted on a subset of these datasets, demonstrating that our design is not overfitting to a specific dataset.

Third, the Baum-Welch algorithm operates on a sub-region of the pHMM graph, the size of which is determined by the sequence length or chunk size, whichever is shorter. Thus, the complexity of a single Baum-Welch execution on this sub-region is determined mainly based on the sequence length and the alphabet size, regardless of the overall genome size or the sequence lengths larger than the chunk size. In our case, we cover all these cases: (1) the pHMM subgraph is determined based on the sequence length (around 90 bases) as it is shorter than the chunk size (up to 1,000 bases) in the protein family search and the multiple sequence alignment (2) the length of the pHMM subgraph is determined by the chunk size in error correction as the sequence length is usually larger (around 10,000 bases) than the chunk size, and (3) different alphabet sizes in DNA and protein.

Fourth, for overhead analysis, we discuss in Section 3.1, we ensure our design is not overfitting to a specific dataset by using a subset of these datasets for each use case. The overhead was measured by taking the geometric mean across different runs to further ensure the robustness of our design. Since our ASIC design is mainly influenced based on the observations we make in our overhead analysis and to maximize the performance improvement for the applications mainly bottlenecked

[1]Top 20 protein families can be found at http://pfam-legacy.xfam.org/family/browse?browse=top%20twenty

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
128 KB L1-Memory	0.632	100

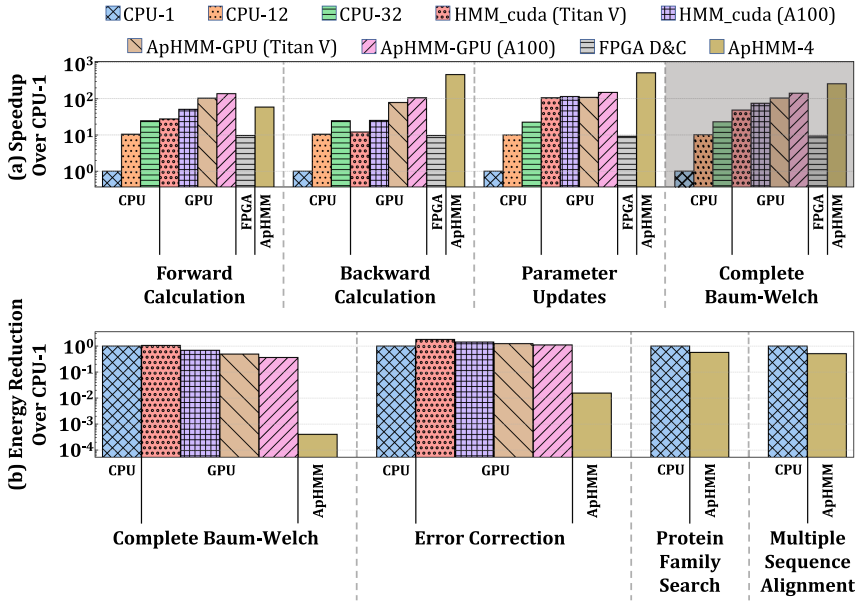

Fig. 10. (a) Normalized speedups of each step in the Baum-Welch algorithm over single-threaded CPU (CPU-1). (b) Energy reductions compared to the CPU-1 implementation of the Baum-Welch algorithm and three pHMM-based applications.

by the Baum-Welch algorithm (i.e., error correction), we believe the comprehensiveness of our data set choice and the overhead analysis enable us improving the robustness of our accelerator across a wide range of potentially many other use cases other than the use cases we evaluate in this work.

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 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, aApHMM core incurs an area overhead of 6.5 mm^2 in 28 nm with a power cost of 0.509 W.

Table 3. Speedup of Each Optimization Over CPU

Optimization	Speedup (×)
Histogram Filter	1.07
LUTs	2.48
Broadcasting and Partial Compute	3.39
Memoization	1.69
Overall	15.20

5.3 Accelerating the Baum-Welch Algorithm

Figure 10 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. Although our evaluations do not directly compare the state-of-the-art FPGA work with the potential FPGA implementation of ApHMM, we believe the performance benefits that ApHMM provides arise not only from the differences in the platform and architecture but also from the optimizations we provide, which are absent in the existing FPGA work. We believe the benefits of these optimizations on the same platform can partly be observed when comparing ApHMM-GPU with the state-of-the-art GPU accelerator. 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 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. 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 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.

Breakdown of the optimizations benefits. Table 3 shows the performance improvements that each ApHMM optimization contributes for executing the Baum-Welch algorithm given the single-core hardware configuration we discuss in Section 4.4 compared to the CPU baseline of the Baum-Welch algorithm. We estimate the speedup of `Histogram Filter` by eliminating the sorting mechanism from filtering while considering the overhead of redundant states included

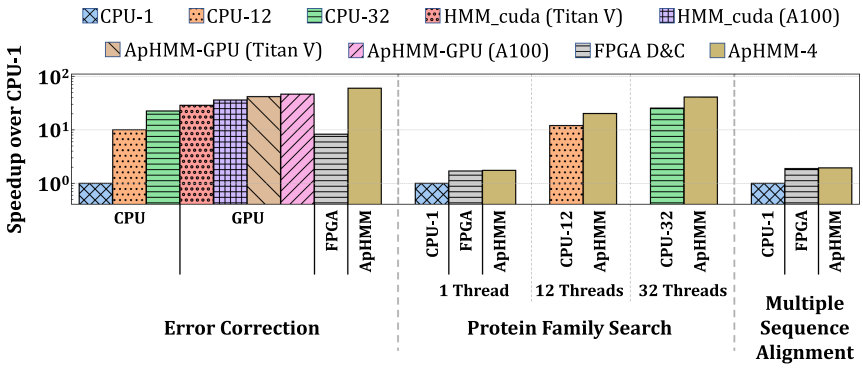

Fig. 11. Speedups over the single-threaded CPU implementations. In the protein family search, we compare ApHMM with each CPU thread separately.

in `Histogram Filter`. For other optimizations, we conservatively estimate the performance speedups by considering the memory bandwidth reductions that each optimization provides, as discussed in Section 4, and the relation between acceleration speedup and the memory bandwidth requirements (Figure 8). We make five key observations. First, we find almost all optimizations contribute significantly to reducing the overall execution time of the Baum-Welch algorithm. Although `Histogram Filter` provides a limited speedup, this is because it constitutes around 8.5% of the overall execution time (Observation 4 in Section 3.1). Second, the tight coupling of the broadcasting and the partial compute approach provides the most significant speedups by avoiding a large number of useless data movements with significant memory bandwidth reductions. Third, the speedup from LUTs is mainly achieved by eliminating many single-precision floating-point operations, causing around 22.7% of the total execution time (Observation 3 in Section 3.1). Fourth, the speedups with the memoization technique are purely achieved by significantly reducing the data movement latency when frequently calculating the transition probabilities. Fifth, we find that the memoization and the partial compute optimizations are utilized only in the training step, and the LUTs can be useful when the alphabet size is small (e.g., 4 in DNAs) due to storage limitations, which is usually the case when the Baum-Welch algorithm is used mainly for inference with the protein sequencing data. Although these benefits cannot be fully utilized in such cases, the remaining optimizations still provide a significant speedup up to 3.63×. We conclude that our optimizations achieve significant speedups for various use cases, from training with DNA sequencing data to inferring with protein sequencing data, allowing the acceleration of many applications that use the Baum-Welch algorithm with pHMMs.

5.4 Use Case 1: Error Correction

Figures 11 and 10 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.

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 11 and 10, 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 the protein family search than error correction, because (1) ApHMM accelerates a smaller portion of the 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 11. We conclude that ApHMM improves the performance and energy efficiency for the protein family search, while there is a smaller room for acceleration compared to the 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 the MSA, as shown in Figures 11 and 10, 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 the 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 the 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 [22, 29, 33, 94–97, 138–147] mainly focus on specific algorithms and designs of HMMs to accelerate the HMM-based applications. Several works [138, 143–147] 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 [29, 94, 97, 139] accelerates the Forward calculation based on the HMM designs different than pHMMs for FPGAs and supercomputers. HMM_cuda [96] 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 [22, 29, 94, 139–142]. 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 causes a significant computational overhead 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

We thank the SAFARI group members and Intel Labs for feedback and the stimulating intellectual environment.

REFERENCES

[1] Sean R. Eddy. 2004. What is a hidden Markov model? *Nat. Biotechnol.* 22 (Oct. 2004), 1315–1316.

[2] Bhavya Mor, Sunita Garhwal, and Ajay Kumar. 2021. A systematic review of hidden Markov models and their applications. *Arch. Comput. Methods Eng.* (2021).

[3] Mohammed Kyari Mustafa, Tony Allen, and Kofi Appiah. 2019. A comparative review of dynamic neural networks and hidden Markov model methods for mobile on-device speech recognition. *Neural. Comput. Appl.* (2019).

[4] Shuiyang Mao, Dehua Tao, Guangyan Zhang, P. C. Ching, and Tan Lee. 2019. Revisiting hidden Markov models for speech emotion recognition. In *Proceedings of the ICASSP*.

[5] Mohamed Hamidi, Hassan Satori, Ouissam Zealouk, Khalid Satori, and Naouar Laaidi. 2018. Interactive voice response server voice network administration using hidden Markov model speech recognition system. In *Proceedings of the WorldS4*.

[6] Chao Xue. 2018. A novel english speech recognition approach based on hidden Markov model. In *Proceedings of the ICVRIS*.

[7] Longfei Li, Yong Zhao, Dongmei Jiang, Yanning Zhang, Fengna Wang, Isabel Gonzalez, Enescu Valentin, and Hichem Sahli. 2013. Hybrid deep neural networkhidden Markov model (DNN-HMM)-based speech emotion recognition. In *Proceedings of the ACII*.

[8] Ibrahim Patel and Y. Srinivasa Rao. 2010. Speech recognition using hidden Markov model with MFCC-subband technique. In *Proceedings of the ITC*.

[9] Zarmeen Nasim and Sayeed Ghani. 2020. Sentiment analysis on Urdu Tweets using Markov chains. *SN Comput. Sci.* (2020).

[10] Mangi Kang, Jaelim Ahn, and Kichun Lee. 2018. Opinion mining using ensemble text hidden Markov models for text classification. *Expert Syst. Appl.* (2018).

[11] Hossein Zeinali, Hossein Sameti, Lukas Burget, and Jan Honza Cernocky. 2017. Text-dependent speaker verification based on i-vectors, Neural Networks and Hidden Markov Models. *Comput. Speech Lang.* (2017).

[12] Irfan Ahmad, Sabri A. Mahmoud, and Gernot A. Fink. 2016. Open-vocabulary recognition of machine-printed Arabic text using hidden Markov models. *Pattern Recognit.* (2016).

[13] A. Seara Vieira, E. L. Iglesias, and L. Borrajo. 2014. T-HMM: A novel biomedical text classifier based on hidden Markov models. In *Proceedings of the PACBB*.

[14] Bruna S. Moreira, Angelo Perkusich, and Saulo O. D. Luiz. 2020. An acoustic sensing gesture recognition system design based on a hidden Markov model. *Sensors* (2020).

[15] Keshav Sinha, Rashmi Kumari, Annu Priya, and Partha Paul. 2019. A computer vision-based gesture recognition using hidden Markov model. In *Innovations in Soft Computing and Information Technology*. Springer.

[16] Markus Haid, Bernhard Budaker, Markus Geiger, Daniel Husfeldt, Marie Hartmann, and Nick Berezowski. 2019. Inertial-based gesture recognition for artificial intelligent cockpit control using hidden Markov models. In *Proceedings of the ICCE*.

[17] Alina Delia Calin. 2016. Gesture recognition on kinect time series data using dynamic time warping and hidden Markov models. In *Proceedings of the SYNASC*.

[18] Nachiket Deo, Akshay Rangesh, and Mohan Trivedi. 2016. In-vehicle hand gesture recognition using hidden Markov models. In *Proceedings of the ITSC*.

[19] Greg Malysa, Dan Wang, Lorin Netsch, and Murtaza Ali. 2016. Hidden Markov model-based gesture recognition with FMCW radar. In *Proceedings of the GlobalSIP*.

[20] Nhan Nguyen-Duc-Thanh, Sungyoung Lee, and Donghan Kim. 2012. Two-stage hidden Markov model in gesture recognition for human robot interaction. *Int. J. Adv. Robot. Syst.* (2012).

[21] Rajat Shrivastava. 2013. A hidden Markov model based dynamic hand gesture recognition system using OpenCV. In *Proceedings of the IACC*.

[22] Xiao Wu, Arun Subramaniyan, Zhehong Wang, Satish Narayanasamy, Reetu Das, and David Blaauw. 2020. 17.3 GCUPS pruning-based pair-hidden-Markov-model accelerator for next-generation DNA sequencing. In *Proceedings of the VLSI*.

[23] Hu Lanyue, Chen Jianhua, Wang Rongshu, Lu Zhiwen, and Hou Bin. 2020. A 5 read hybrid error correction algorithm based on segmented pHMM. In *Proceedings of the ICMCCE*.

[24] Can Firtina, Jeremie S. Kim, Mohammed Alser, Damla Senol Cali, A Ercument Cicek, Can Alkan, and Onur Mutlu. 2020. Apollo: A sequencing-technology-independent, scalable and accurate assembly polishing algorithm. *Bioinform.* (2020).

[25] Martin Steinegger, Markus Meier, Milot Mirdita, Harald Vöhringer, Stephan J. Haunsberger, and Johannes Söding. 2019. HH-suite3 for fast remote homology detection and deep protein annotation. *BMC Bioinform.* (2019).

[26] Ioannis A. Tamposis, Konstantinos D. Tsirigos, Margarita C. Theodoropoulou, Panagiota I. Kontou, and Pantelis G. Bagos. 2019. Semi-supervised learning of Hidden Markov Models for biological sequence analysis. *Bioinform.* (2019).

[27] Can Firtina, Ziv Bar-Joseph, Can Alkan, and A Ercument Cicek. 2018. Hercules: A profile HMM-based hybrid error correction algorithm for long reads. *NAR* (2018).

[28] Xiaole Yin, Xiao-Tao Jiang, Benli Chai, Liguan Li, Ying Yang, James R. Cole, James M. Tiedje, and Tong Zhang. 2018. 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. *Bioinform.* (2018).

[29] Sitao Huang, Gowthami Jayashri Manikandan, Anand Ramachandran, Kyle Rupnow, Wen-mei W. Hwu, and Deming Chen. 2017. Hardware acceleration of the pair-HMM algorithm for DNA variant calling. In *Proceedings of the FPGA*.

[30] Vagheesh Narasimhan, Petr Danecek, Aylwyn Scally, Yali Xue, Chris Tyler-Smith, and Richard Durbin. 2016. BCFtools/RoH: A hidden Markov model approach for detecting autozygosity from next-generation sequencing data. *Bioinform.* (2016).

[31] Lei Zhang, Yu-Fang Pei, Xiaoying Fu, Yong Lin, Yu-Ping Wang, and Hong-Wen Deng. 2014. FISH: Fast and accurate diploid genotype imputation via segmental hidden Markov model. *Bioinform.* (2014).

[32] Travis J. Wheeler, Jody Clements, Sean R. Eddy, Robert Hubley, Thomas A. Jones, Jerzy Jurka, Arian F. A. Smit, and Robert D. Finn. 2012. Dfam: A database of repetitive DNA based on profile hidden Markov models. *NAR* (2012).

[33] Sean R. Eddy. 2011. Accelerated profile HMM searches. *PLoS Comput. Biol.* (2011).

[34] Byung-Jun Yoon. 2009. Hidden Markov models and their applications in biological sequence analysis. *Curr. Genomics* (2009).

[35] Martin Madera. 2008. Profile comparer: A program for scoring and aligning profile hidden Markov models. *Bioinform.* (2008).

[36] Kuo-ching Liang, Xiaodong Wang, and Dimitris Anastassiou. 2007. Bayesian basecalling for DNA sequence analysis using hidden Markov models. *IEEE TCBB* (2007).

[37] Torben Friedrich, Birgit Pils, Thomas Dandekar, Jörg Schultz, and Tobias Müller. 2006. Modelling interaction sites in protein domains with interaction profile hidden Markov models. *Bioinform.* (2006).

[38] Nikolaos G. Sgourakis, Pantelis G. Bagos, Panagiotis K. Papasaikas, and Stavros J. Hamodrakas. 2005. A method for the prediction of GPCRs coupling specificity to G-proteins using refined profile Hidden Markov Models. *BMC Bioinform.* (2005).

[39] Robert C. Edgar and K. Sjolander. 2004. COACH: Profile-profile alignment of protein families using hidden Markov models. *Bioinform.* (2004).

[40] Petros Boufounos, Sameh El-Difrawy, and Dan Ehrlich. 2004. Basecalling using hidden Markov models. *J. Frank. Inst.* (2004).

[41] Zemin Zhang and William I. Wood. 2003. A profile hidden Markov model for signal peptides generated by HMMER. *Bioinform.* (2003).

[42] Alex Bateman, Ewan Birney, Lorenzo Cerruti, Richard Durbin, Laurence Etwiller, Sean R. Eddy, Sam Griffiths-Jones, Kevin L. Howe, Mhairi Marshall, and Erik L.L. Sonnhammer. 2002. The PFAM protein families database. *NAR* (2002).

[43] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. 1998. *Biological Sequence Analysis*. Cambridge University Press.

[44] Sean R. Eddy. 1998. Profile hidden Markov models. *Bioinform.* (1998).

[45] Pierre Baldi, Yves Chauvin, Tim Hunkapiller, and M. A. McClure. 1994. Hidden Markov models of biological primary sequence information. *Proc. Natl. Acad. Sci. U.S.A.* (1994).

[46] Muhammad Ali, Monem Hamid, Jacob Jasser, Joachim Lerman, Samod Shetty, and Fabio Di Troia. 2022. Profile hidden Markov model malware detection and API call obfuscation. In *Proceedings of the ICISSP*.

[47] Satheesh Kumar Sasidharan and Ciza Thomas. 2021. ProDroidAn Android malware detection framework based on profile hidden Markov model. *PMC* (2021).

[48] Xiaolei Liu, Zhongliu Zhuo, Xiaojiang Du, Xiaosong Zhang, Qingxin Zhu, and Mohsen Guizani. 2019. Adversarial attacks against profile HMM website fingerprinting detection model. *Cogn. Syst. Res.* (2019).

[49] Ramandika Pranamulia, Yudistira Asnar, and Riza Satria Perdana. 2017. Profile hidden Markov model for malware classification usage of system call sequence for malware classification. In *Proceedings of the ICoDSE*.

[50] Saradha Ravi, N. Balakrishnan, and Bharath Venkatesh. 2013. Behavior-based Malware analysis using profile hidden Markov models. In *Proceedings of the SECRYPT*.

[51] Srilatha Attaluri, Scott McGhee, and Mark Stamp. 2009. Profile hidden Markov models and metamorphic virus detection. *J. Comput. Virol.* (2009).

[52] A. B. Riddell. 2022. Reliable editions from unreliable components: Estimating ebooks from print editions using profile hidden Markov models. In *Proceedings of the JCDL*.

[53] Ioannis Kazantzidis, Francisco Florez-Revuelta, and Jean-Christophe Nebel. 2018. Profile hidden Markov models for foreground object modelling. In *Proceedings of the ICIP*.

[54] Ismaïl Saadi, Feng Liu, Ahmed Mustafa, Jacques Teller, and Mario Cools. 2016. A framework to identify housing location patterns using profile hidden Markov Models. *Adv. Sci. Lett* (2016).

[55] Wenwen Ding, Kai Liu, Fei Cheng, Huan Shi, and Baijian Zhang. 2015. Skeleton-based human action recognition with profile hidden Markov models. In *Proceedings of the CCCV*.

[56] Feng Liu, Davy Janssens, JianXun Cui, Geert Wets, and Mario Cools. 2015. Characterizing activity sequences using profile Hidden Markov Models. *Expert Syst. Appl.* (2015).

[57] Yan Liu, Pei-Yun Hsueh, Jennifer Lai, Mirweis Sangin, Marc-Antoine Nussli, and Pierre Dillenbourg. 2009. Who is the expert? Analyzing gaze data to predict expertise level in collaborative applications. In *Proceedings of the ICME*.

[58] Onur Mutlu and Can Firtina. 2023. Accelerating genome analysis via algorithm-architecture co-design. In *Proceedings of the DAC*.

[59] Can Firtina, Melina Soysal, Joël Lindegger, and Onur Mutlu. 2023. RawHash2: Accurate and fast mapping of raw nanopore signals using a hash-based seeding mechanism. arXiv: 2309.05771. Retrieved from https://arxiv.org/abs/2309.05771

[60] Joël Lindegger, Can Firtina, Nika Mansouri Ghiasi, Mohammad Sadrosadati, Mohammed Alser, and Onur Mutlu. 2023. RawAlign: Accurate, fast, and scalable raw nanopore signal mapping via combining seeding and alignment. arXiv: 2310.05037. Retrieved from https://arxiv.org/abs/2310.05037

[61] Can Firtina, Nika Mansouri Ghiasi, Joel Lindegger, Gagandeep Singh, Meryem Banu Cavlak, Haiyu Mao, and Onur Mutlu. 2023. RawHash: Enabling fast and accurate real-time analysis of raw nanopore signals for large genomes. *Bioinform.* (2023).

[62] Jeremie S. Kim, Can Firtina, Meryem Banu Cavlak, Damla Senol Cali, Nastaran Hajinazar, Mohammed Alser, Can Alkan, and Onur Mutlu. 2023. AirLift: A fast and comprehensive technique for remapping alignments between reference genomes. In *Proceedings of the APBC*.

[63] Can Firtina, Jisung Park, Mohammed Alser, Jeremie S. Kim, Damla Senol Cali, Taha Shahroodi, Nika Mansouri Ghiasi, Gagandeep Singh, Konstantinos Kanellopoulos, Can Alkan, and Onur Mutlu. 2023. BLEND: A fast, memory-efficient and accurate mechanism to find fuzzy seed matches in genome analysis. *NARGAB* (2023).

[64] Jeremie S. Kim, Can Firtina, Meryem Banu Cavlak, Damla Senol Cali, Can Alkan, and Onur Mutlu. 2022. FastRemap: A tool for quickly remapping reads between genome assemblies. *Bioinform.* (2022).

[65] Mohammed Alser, Joel Lindegger, Can Firtina, Nour Almadhoun, Haiyu Mao, Gagandeep Singh, Juan Gomez-Luna, and Onur Mutlu. 2022. From molecules to genomic variations: Accelerating genome analysis via intelligent algorithms and architectures. *CSBJ* (2022).

[66] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, Rachata Ausavarungnirun, Nandita Vijaykumar, Mohammed Alser, and Onur Mutlu. 2022. GenStore: A high-performance in-storage processing system for genome sequence analysis. In *Proceedings of the ASPLOS*.

[67] Damla Senol Cali, Konstantinos Kanellopoulos, Joël Lindegger, Zülal Bingöl, Gurpreet S. Kalsi, Ziyi Zuo, Can Firtina, Meryem Banu Cavlak, Jeremie Kim, Nika Mansouri Ghiasi, Gagandeep Singh, Juan Gómez-Luna, Nour Almadhoun Alserr, Mohammed Alser, Sreenivas Subramoney, Can Alkan, Saugata Ghose, and Onur Mutlu. 2022. SeGraM: A universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping. In *Proceedings of the ISCA*.

[68] Mohammed Alser, Jeremy Rotman, Dhrithi Deshpande, Kodi Taraszka, Huwenbo Shi, Pelin Icer Baykal, Harry Taegyun Yang, Victor Xue, Sergey Knyazev, Benjamin D. Singer, Brunilda Balliu, David Koslicki, Pavel Skums, Alex Zelikovsky, Can Alkan, Onur Mutlu, and Serghei Mangul. 2021. Technology dictates algorithms: Recent developments in read alignment. *Genome Biol.* (2021).

[69] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Diamantopoulos Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. 2021. FPGA-based near-memory acceleration of modern data-intensive applications. *IEEE Micro* (2021).

[70] Mohammed Alser, Zulal Bingöl, Damla Senol Cali, Jeremie Kim, Saugata Ghose, Can Alkan, and Onur Mutlu. 2020. Accelerating genome analysis: A primer on an ongoing journey. *IEEE Micro* (2020).

[71] Mohammed Alser, Taha Shahroodi, Juan Gómez-Luna, Can Alkan, and Onur Mutlu. 2020. SneakySnake: A fast and accurate universal genome pre-alignment filter for CPUs, GPUs and FPGAs. *Bioinform.* (2020).

[72] Shaahin Angizi, Jiao Sun, Wei Zhang, and Deliang Fan. 2020. PIM-aligner: A processing-in-MRAM platform for biological sequence alignment. In *Proceedings of the DATE*.

[73] Sneha D. Goenka, Yatish Turakhia, Benedict Paten, and Mark Horowitz. 2020. SegAlign: A scalable GPU-based whole genome aligner. In *Proceedings of the SC20*.

[74] Damla Senol Cali, Gurpreet S. Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S. Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, Anant Norion, Allison Scibisz, Sreenivas Subramoneyon, Can Alkan, Saugata Ghose, and Onur Mutlu. 2020. GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In *Proceedings of the MICRO*.

[75] Anirban Nag, C. N. Ramachandra, Rajeev Balasubramonian, Ryan Stutsman, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. 2019. GenCache: Leveraging in-cache operators for efficient sequence alignment. In *Proceedings of the MICRO*.

[76] Damla Senol Cali, Jeremie S. Kim, Saugata Ghose, Can Alkan, and Onur Mutlu. 2019. Nanopore sequencing technology and tools for genome assembly: computational analysis of the current state, bottlenecks and future directions. *Brief. Bioinform.* (2019).

[77] Mohammed Alser, Hasan Hassan, Akash Kumar, Onur Mutlu, and Can Alkan. 2019. Shouji: A fast and efficient pre-alignment filter for sequence alignment. *Bioinformatics* (2019).

[78] Yatish Turakhia, Gill Bejerano, and William J. Dally. 2018. Darwin: A genomics co-processor provides up to 15,000X acceleration on long read assembly. In *Proceedings of the ASPLOS*.

[79] Jeremie S. Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. 2018. GRIM-filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC Genomics* (2018).

[80] Mohammed Alser, Hasan Hassan, Hongyi Xin, Oğuz Ergin, Onur Mutlu, and Can Alkan. 2017. GateKeeper: A new hardware architecture for accelerating pre-alignment in DNA short read mapping. *Bioinformatics* (2017).

[81] Johannes Söding, Andreas Biegert, and Andrei N. Lupas. 2005. The HHpred interactive server for protein homology detection and structure prediction. *NAR* (2005).

[82] Robert D. Finn, Jaina Mistry, John Tate, Penny Coggill, Andreas Heger, Joanne E. Pollington, O. Luke Gavin, Prasad Gunasekaran, Goran Ceric, Kristoffer Forslund, Liisa Holm, Erik L. L. Sonnhammer, Sean R. Eddy, and Alex Bateman. 2010. The Pfam protein families database. *NAR* (2010).

[83] Martin Madera and Julian Gough. 2002. A comparison of profile hidden Markov model procedures for remote homology detection. *NAR* (2002).

[84] Sudipta Mulia, Debahuti Mishra, and Tanushree Jena. 2012. Profile HMM based multiple sequence alignment for DNA sequences. *Procedia Eng.* (2012).

[85] Jimin Pei and Nick V. Grishin. 2007. PROMALS: Towards accurate multiple sequence alignments of distantly related proteins. *Bioinformatics* (2007).

[86] Robert C. Edgar and Kimmen Sjölander. 2003. SATCHMO: Sequence alignment and tree construction using hidden Markov models. *Bioinformatics* (2003).

[87] Vahid Rezaei, Hamid Pezeshk, and Horacio Pérez-Sa'nchez. 2013. Generalized Baum-Welch algorithm based on the similarity between sequences. *PLoS ONE* (2013).

[88] Steven J. Lewis, Alpan Raval, and John E. Angus. 2008. Bayesian monte carlo estimation for profile hidden Markov models. *Math. Comput. Model.* (2008).

[89] Leonard E. Baum. 1972. An inequality and associated maximization technique in statistical estimation of probabilistic functions of a Markov process. *Inequalities* (1972).

[90] Steven L. Scott. 2002. Bayesian methods for hidden Markov models. *JASA* (2002).

[91] Yves Boussemart, Jonathan Las Fargeas, Mary L. Cummings, and Nicholas Roy. 2009. Comparing learning techniques for hidden Markov models of human supervisory control behavior. In *Proceedings of the I@A*.

[92] Rune B. Lyngsø and Christian N. S. Pedersen. 2002. The consensus string problem and the complexity of comparing hidden Markov models. *JCSS* (2002).

[93] Robel Y. Kahsay, Guoli Wang, Guang Gao, Li Liao, and Roland Dunbrack. 2005. Quasi-consensus-based comparison of profile hidden Markov models for protein sequences. *Bioinformatics* (2005).

[94] Shanshan Ren, Vlad-Mihai Sima, and Zaid Al-Ars. 2015. FPGA acceleration of the pair-HMMs forward algorithm for DNA sequence analysis. In *Proceedings of the BIBM*.

[95] M. Pietras and P. Klęsk. 2017. FPGA implementation of logarithmic versions of Baum-Welch and Viterbi algorithms for reduced precision hidden Markov models. *B Pol. Acad. Sci.-Tech.* (2017).

[96] Leiming Yu, Yash Ukidave, and David Kaeli. 2014. GPU-accelerated HMM for speech recognition. In *Proceedings of the ICPADS*.

[97] Stefania-Iuliana Soiman, Ionela Rusu, and Stefan-Gheorghe Pentiuc. 2014. A parallel accelerated approach of HMM Forward Algorithm for IBM Roadrunner clusters. In *Proceedings of the DAS*.

[98] T. K. Moon. 1996. The expectation-maximization algorithm. *IEEE Signal Process. Mag.* (1996).

[99] Amirhossein Tavanaei and Anthony S. Maida. 2018. Training a hidden Markov model with a bayesian spiking neural network. *J. Signal Process. Syst.* (2018).

[100] David Volent Lindberg and Dario Grana. 2015. Petro-elastic log-facies classification using the expectationmaximization algorithm and hidden Markov models. *Math. Geosci.* (2015).

[101] Aliaksandr Hubin. 2019. An adaptive simulated annealing EM algorithm for inference on non-homogeneous hidden Markov models. In *Proceedings of the AIIPCC*.

[102] Robert Vaser, Ivan Sović, Niranjan Nagarajan, and Mile Šikić. 2017. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome Res.* (2017).

[103] Jiang Hu, Junpeng Fan, Zongyi Sun, and Shanlin Liu. 2020. NextPolish: A fast and efficient genome polishing tool for long-read assembly. *Bioinformatics* (2020).

[104] Neng Huang, Fan Nie, Peng Ni, Feng Luo, Xin Gao, and Jianxin Wang. 2021. NeuralPolish: A novel Nanopore polishing method based on alignment matrix construction and orthogonal Bi-GRU Networks. *Bioinformatics* (2021).

[105] Bruce J. Walker, Thomas Abeel, Terrance Shea, Margaret Priest, Amr Abouelliel, Sharadha Sakthikumar, Christina A. Cuomo, Qiandong Zeng, Jennifer Wortman, Sarah K. Young, and Ashlee M. Earl. 2014. Pilon: An integrated tool for comprehensive microbial variant detection and genome assembly improvement. *PLoS ONE* (2014).

[106] Aleksey V. Zimin and Steven L. Salzberg. 2020. The genome polishing tool POLCA makes fast and accurate corrections in genome assemblies. *PLoS Comput. Biol.* (2020).

[107] Chen-Shan Chin, David H. Alexander, Patrick Marks, Aaron A. Klammer, James Drake, Cheryl Heiner, Alicia Clum, Alex Copeland, John Huddleston, Evan E. Eichler, Stephen W. Turner, and Jonas Korlach. 2013. Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nat. Methods* (2013).

[108] A. Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inf.* (1967).

[109] Nicola J. Mulder and Rolf Apweiler. 2001. Tools and resources for identifying protein families, domains and motifs. *Genome Biol.* (2001).

[110] Matt Jeffryes and Alex Bateman. 2018. Rapid identification of novel protein families using similarity searches. *F1000Research* (2018).

[111] Seokjun Seo, Minsik Oh, Youngjune Park, and Sun Kim. 2018. DeepFam: Deep learning based alignment-free method for protein family modeling and prediction. *Bioinformatics* (2018).

[112] R. Vicedomini, J.P. Bouly, E. Laine, A. Falciatore, and A. Carbone. 2022. Multiple profile models extract features from protein sequence data and resolve functional diversity of very different protein families. *Mol. Biol. Evol.* (2022).

[113] Pablo Turjanski and Diego U. Ferreiro. 2018. On the natural structure of amino acid patterns in families of protein sequences. *J. Phys. Chem. B.* (2018).

[114] Maxwell L. Bileschi, David Belanger, Drew H. Bryant, Theo Sanderson, Brandon Carter, D. Sculley, Alex Bateman, Mark A. DePristo, and Lucy J. Colwell. 2022. Using deep learning to annotate the protein universe. *Nat. Biotechnol.* (2022).

[115] Jaina Mistry, Sara Chuguransky, Lowri Williams, Matloob Qureshi, Gustavo A. Salazar, Erik L. L. Sonnhammer, Silvio C. E. Tosatto, Lisanna Paladin, Shriya Raj, Lorna J. Richardson, Robert D. Finn, and Alex Bateman. 2021. Pfam: The protein families database in 2021. *NAR* (2021).

[116] Peter Skewes-Cox, Thomas J. Sharpton, Katherine S. Pollard, and Joseph L. DeRisi. 2014. Profile hidden Markov models for the detection of viruses within metagenomic Sequence Data. *PLoS ONE* (2014).

[117] Winfried Just. 2001. Computational complexity of multiple sequence alignment with SP-score. *J. Comput. Biol.* (2001).

[118] Lusheng Wang and Tao Jiang. 1994. On the complexity of multiple sequence alignment. *J. Comput. Biol.* (1994).

[119] Biswanath Chowdhury and Gautam Garai. 2017. A review on multiple sequence alignment from the perspective of genetic algorithm. *Genomics* (2017).

[120] Qing Zhan, Nan Wang, Shuilin Jin, Renjie Tan, Qinghua Jiang, and Yadong Wang. 2019. ProbPFP: A multiple sequence alignment algorithm combining hidden Markov model optimized by particle swarm optimization with partition function. *BMC Bioinform.* (2019).

[121] Intel. 2022. Vtune Profiler. Retrieved from https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[122] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. 2004. Gprof: A call graph execution profiler. *SIGPLAN Not.* (2004).

[123] Bonnie Kirkpatrick and Kay Kirkpatrick. 2012. Optimal state-space reduction for pedigree hidden Markov models. arXiv: 1202.2468. Retrieved from https://arxiv.org/abs/1202.2468

[124] István Miklós and Irmtraud M. Meyer. 2005. A linear memory algorithm for Baum-Welch training. *BMC Bioinform.* (2005).

[125] J.Alicia Grice, Richard Hughey, and Don Speck. 1997. Reduced space sequence alignment. *Bioinformatics* (1997).

[126] Raymond Wheeler and Richard Hughey. 2000. Optimizing reduced-space sequence analysis. *Bioinformatics* (2000).

[127] C. Tarnas and R. Hughey. 1998. Reduced space hidden Markov model training. *Bioinformatics* (1998).

[128] Pei Chen, Rui Liu, Yongjun Li, and Luonan Chen. 2016. Detecting critical state before phase transition of complex biological systems by hidden Markov model. *Bioinformatics* (2016).

[129] Fábio Madeira, Young mi Park, Joon Lee, Nicola Buso, Tamer Gur, Nandana Madhusoodanan, Prasad Basutkar, Adrian R. N. Tivey, Simon C. Potter, Robert D. Finn, and Rodrigo Lopez. 2019. The EMBL-EBI search and sequence analysis tools APIs in 2019. *NAR* (2019).

[130] Simon C. Potter, Aurélien Luciani, Sean R. Eddy, Youngmi Park, Rodrigo Lopez, and Robert D. Finn. 2018. HMMER web server: 2018 update. *NAR* (2018).

[131] Sara El-Gebali, Jaina Mistry, Alex Bateman, Sean R. Eddy, Aurélien Luciani, Simon C. Potter, Matloob Qureshi, Lorna J. Richardson, Gustavo A. Salazar, Alfredo Smart, Erik L. L. Sonnhammer, Layla Hirsh, Lisanna Paladin, Damiano Piovesan, Silvio C. E. Tosatto, and Robert D. Finn. 2019. The Pfam protein families database in 2019. *NAR* (2019).

[132] Wenjun Li, Kathleen R. O'Neill, Daniel H. Haft, Michael DiCuccio, Vyacheslav Chetvernin, Azat Badretdin, George Coulouris, Farideh Chitsaz, Myra K. Derbyshire, A Scott Durkin, Noreen R. Gonzales, Marc Gwadz, Christopher J. Lanczycki, James S. Song, Narmada Thanki, Jiyao Wang, Roxanne A. Yamashita, Mingzhang Yang, Chanjuan Zheng,

Aron Marchler-Bauer, and Françoise Thibaud-Nissen. 2021. RefSeq: Expanding the Prokaryotic genome annotation pipeline reach with protein family model curation. *NAR* (2021).

[133] Hernan A. Lorenzi, Daniela Puiu, Jason R. Miller, Lauren M. Brinkac, Paolo Amedeo, Neil Hall, and Elisabet V. Caler. 2010. New assembly, reannotation and analysis of the entamoeba histolytica genome reveal new genomic features and protein content information. *PLoS Negl. Trop. Dis.* (2010).

[134] Synopsys. 2016. Design Compiler (Version L-2016.03-SP2). (Mar. 2016).

[135] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue* (2008).

[136] Heng Li. 2018. Minimap2: Pairwise alignment for nucleotide sequences. *Bioinformatics* (2018).

[137] Heng Li. 2016. Minimap and miniasm: Fast mapping and de novo assembly for noisy long sequences. *Bioinformatics* (2016).

[138] Atef Ibrahim, Hamed Elsimary, Abdullah Aljumah, and Fayez Gebali. 2016. Reconfigurable hardware accelerator for profile hidden Markov models. *Arab J. Sci. Eng.* (2016).

[139] Enliang Li, Subho S. Banerjee, Sitao Huang, Ravishankar K. Iyer, and Deming Chen. 2021. Improved GPU implementations of the pair-HMM forward algorithm for DNA sequence alignment. In *Proceedings of the ICCD*.

[140] Rick Wertenbroek and Yann Thoma. 2019. Acceleration of the pair-HMM forward algorithm on FPGA with cloud integration for GATK. In *Proceedings of the BIBM*.

[141] Subho S. Banerjee, Mohamed el Hadedy, Ching Y. Tan, Zbigniew T. Kalbarczyk, Steve Lumetta, and Ravishankar K. Iyer. 2017. On accelerating pair-HMM computations in programmable hardware. In *Proceedings of the FPL*.

[142] Xiao Wu, Arun Subramaniyan, Zhehong Wang, Satish Narayanasamy, Reetuparna Das, and David Blaauw. 2021. A high-throughput pruning-based pair-hidden-Markov-model hardware accelerator for next-generation DNA sequencing. *IEEE Solid-State Circ. Lett.* (2021).

[143] Hanyu Jiang, Narayan Ganesan, and Yu-Dong Yao. 2018. CUDAMPF++: A proactive resource exhaustion scheme for accelerating homologous sequence search on CUDA-enabled GPU. *IEEE TPDS* (2018).

[144] Saddam Quirem, Fahian Ahmed, and Byeong Kil Lee. 2011. CUDA acceleration of P7Viterbi algorithm in HMMER 3.0. In *Proceedings of the IPCCC*.

[145] Steven Derrien and Patrice Quinton. 2008. Hardware acceleration of HMMER on FPGAs. *J. Signal Process. Syst.* (2008).

[146] Tim Oliver, Leow Yuan Yeow, and Bertil Schmidt. 2007. High performance database searching with HMMer on FPGAs. In *Proceedings of the IPDPS*.

[147] Tim Oliver, Leow Yuan Yeow, and Bertil Schmidt. 2008. Integrating FPGA acceleration into HMMer. *Parallel Comput.* (2008).

Received 19 August 2022; revised 1 June 2023; accepted 6 October 2023