BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES 65 (6, 2017
DOI: 10.1515/bpasts-2017-0101

FPGA implementation of logarithmic versions of Baum-Welch and Viterbi algorithms for reduced precision hidden Markov models

M. PIETRAS and P. KLĘSK*

Faculty of Computer Science and Information Technology, West Pomeranian University of Technology,
49 Żołnierska Street, Szczecin, Poland

Abstract. This paper presents a programmable system-on-chip implementation to be used for acceleration of computations within hidden Markov models. The high level synthesis (HLS) and "divide-and-conquer" approaches are presented for parallelization of Baum-Welch and Viterbi algorithms. To avoid arithmetic underflows, all computations are performed within the logarithmic space. Additionally, in order to carry out computations efficiently – i.e. directly in an FPGA system or a processor cache – we postulate to reduce the floating-point representations of HMMs. We state and prove a lemma about the length of numerically unsafe sequences for such reduced precision models. Finally, special attention is devoted to the design of a multiple logarithm and exponent approximation unit (MLEAU). Using associative mapping, this unit allows for simultaneous conversions of multiple values and thereby compensates for computational efforts of logarithmic-space operations. Design evaluation reveals absolute stall delay occurring by multiple hardware conversions to logarithms and to exponents, and furthermore the experiments evaluation reveals HMMs computation boundaries related to their probabilities and floating-point representation. The performance differences at each stage of computation are summarized in performance comparison between hardware acceleration using MLEAU and typical software implementation on an ARM or Intel processor.

Key words: hidden Markov models, numerical stability, Viterbi algorithm, parallel architectures, field-programmable gate arrays.

1. Introduction

Due to their rich mathematical structure, hidden Markov models (HMMs) find numerous applications. Most of those concern different types of pattern recognition and classification tasks, and can be divided into two main categories, depending on computation latency requirements. The first category covers general temporal data mining, where HMMs operate on data in order to find statistically relevant patterns and extract important information [1–2]. In many cases, such data mining includes big data analysis, and the computation required for this analysis can be made in a cloud platform without any special computation latency constraints. The second category of application covers general signal recognition and signal processing, where HMMs operate on data in order to classify it. The latency between data input and system response is restricted and exceeding it leads to application malfunction [3–4]. This real time HMM processing is used in a wide range of applications, including speech synthesis [5] and recognition [6], image recognition, movement recognition [7], radar [8] and sonar [9–10] detection and sense-and-avoid systems. However, in many application, due to computation deficiency, only features pre-processing is performed locally while further HMM calculations are performed on the server side (on a cloud platform). This approach limits usage in an obvious way, i.e. the application needs access to the network, and as the operation of the network is affected by external factors, real time computation is not guaranteed. Hence, for low latency applications, full HMM processing and computation within a dedicated embedded system is a reliable choice, and this has been shown for diverse systems such as speech recognition [11–12], pattern detection [13] and for AAV/AUV [14].

Unfortunately, the increasing complexity of HMMs is paralleled by the growing demand for computational resources, especially memory and data throughput. Hence, when considering the hardware acceleration of HMM algorithms (forward-backward, Viterbi, Baum-Welch), the size of the HMM has to be minimized while the stability of numerical calculation still has to be guaranteed. As mentioned in [15], typical calculations associated with HMM rapidly exhaust the precision of numerical representation. This is related to the necessity of performing long sequences of multiplications of probability values (which are close to zero) for state transitions or observation emissions. Therefore, key algorithms are computed within the logarithmic space instead of applying some scaling factors [16]. Decreasing the size of an HMM can be immediately achieved by reducing the precision of its numerical representation (transition and emission matrices), e.g. down to: 32 bits (single), 16 bits (half) or 8 bits (quarter). This action directly affects and restricts the maximum length of the observation sequences examined, which means that computations on longer sequences may become numerically unstable.

1.1. Related work. Acceleration and parallelization of HMM algorithms is not an easy task but many applications benefit

[*] e-mail: pklesk@wi.zut.edu.pl

Manuscript submitted 2016-11-04, revised 2017-03-13, initially accepted for publication 2017-04-11, published in December 2017.

from it, especially in the field of bioinformatics [17] and in embedded systems (such as robotics, autonomous vehicles, IoT, etc.). In recent years, there has been a lot of research concerning FPGA acceleration of HMM algorithms [18–21]. Most systems that have been implemented based on this research focus on HMM decoding, using the Viterbi algorithm (computed in linear as well as in logarithmic space), and on evaluation problems, using the forward-backward algorithm (computed in linear memory with scaling factors) [22]. Because most of the research is aimed at obtaining the best possible performance results, the issue of accuracy and numerical stability is often underestimated or completely overlooked.

2. Motivation

The main obstacles to using HMMs for low latency applications are the limitations of fast memory resources and the numerical instability of computations. The complex structure of HMMs requires dozens of megabytes to store transition and emission matrices (e.g. for HMM with 102 states and 105 observations, the required storage memory stands at about 8 MB). This is a major impediment to computing the model directly in the FPGA system or within a processor cache. The obvious solution for this issue is to reduce the floating-point number representation for both matrices. However, precision reductions have a significant impact on numerical stability, especially in HMM algorithms (forward-backward, Viterbi), where numerous probability values (mostly values close to zero) are multiplied together, which may rapidly lead to arithmetical underflows [23]. A possible solution for this in the forward-backward (and further on in the Baum-Welch) algorithm is to use scaling factors which shift away from the underflow. However, scaling factors have to be additionally computed and stored throughout the whole examined sequence of observations, which is disadvantageous (due to additional memory demand), especially for direct implementation in the FPGA as well as for further computation of the Baum-Welch algorithm. Therefore, it is much more convenient for numerical stability to perform the whole calculation in logarithmic space (as is done for the Viterbi algorithm [15]), where instead of multiplications, sums of logarithms are calculated. To benefit from this solution, expensive logarithmic and exponent computations have to be substituted with fast and efficient approximation. For this purpose, a special module with a parallel access lookup table for concurrent mapping of multiple values is evaluated here. Computing all HMM-related algorithms in logarithmic space makes it possible to reduce the precision of number representation, but to ensure numerical stability, certain assumptions (also presented in this paper) associated with the maximum length of the sequence examined for a given HMM have to be satisfied.

3. Divide and conquer methodology

The parallelization of highly iterative dynamic programming algorithms requires full utilization of the capabilities offered by modern FPGAs. Designing acceleration architecture in a typical hardware description language (VHDL) is time-consuming and poses quite a challenge. Additionally, design configuration changes and relevant verification is difficult to perform. In contrast, as is well described in [24], by using high level synthesis (HLS), it is possible to create complex processing architecture at a higher level of abstraction, where design modification and testing is reliable and hassle-free (e.g. in the case of changing the precision of calculation for the entire system or changing the degree of parallelism). Most of the design presented in this paper is created and verified by using Vivado System Design and Vivado HLS framework [25]. The divide and conquer (D&C) strategy provides a very elegant and efficient framework for HMM algorithm implementation. Standard dynamic programming algorithms for HMMs, in each time step, iterate over all states and perform some sub-computations for each of them (typically summation or maximization). Hence, the obvious way to accelerate execution is to perform state-related sub-computation for all states simultaneously. Then the degree of parallelization is dependent on the number of states in the Markov model. Most likely, however, iteration through all states is nested in another state iteration or in observation sequence iteration. If necessary, dependencies resulting from the nested iteration have to be solved in a separate way (e.g. by pipelining).

3.1. Processing system and programmable logic. The design presented in this paper is dedicated for a programmable system on chip (PSoC), where an HMM-based application can benefit from hardware acceleration in programmable logic, but typical general purpose computation is still held by the CPU. The processing system provides general infrastructure for the interfaces (Ethernet, HMI, etc.), and above all is responsible for observation codebook pre-processing and decoding, while the main HMM algorithms such as the logarithmic space Baum-Welch, forward-backward and Viterbi ones are computed in programmable logic. In whatever HMM utilized in the application, the following typical processing stages can be distinguished:

1) Analyzing and pre-processing some raw data in order to extract the chain of some expressions for further analysis.
2) Acquiring the emissions sequence, where the chain of some expressions is assembled to form the emissions.
3) Codebook decoding, where the examined emission is compared with the predefined token in order to map it onto the corresponding observation.
4) If the HMM have to be trained, the Baum-Welch algorithm is selected, or if the purpose of the HMM is to explore the path, the Viterbi algorithm is selected.
5) Loading an HMM to the “computation engine”.
6) Loading the observation sequence to the “computation engine” and performing the algorithm.
7) Collecting the results.

All stages except stage 6 are application-specific and should definitely be performed by the processing system. In most HMM-based applications, the pre-processing dictionary and codebook for decoding expressions has from just a few to dozens of thousands of tokens [26]. Furthermore, because of the potentially large size of the HMM, the containers are stored in DDR memory. The HMM container includes basic information about the model, such as transition and emission probabilities and the observation codebook with corresponding mapping.

Stage 6 can be performed outside the processing system. "Computation engine" is not defined by how the computation is made – whether by the CPU, an external system (in a cloud) or programmable logic. For low latency processing, the most suitable way of making the computation is by using programmable logic, so, in order to accelerate HMM computation, direct (instant on) access to both matrices (transitions and emissions) is required, which involves loading the HMM to the FPGA (stage 5). Further calculations are performed parallelly for each state. For this purpose, each dedicated state processing unit (SPU) stores its own transition and emission probability vectors. Fig. 1 shows the general idea of the computational architecture, where HMM-related calculations are performed by the SPUs controlled by the control and data management (CDM) unit.

3.2. Multiple logarithm exponent approximation unit.

In order to efficiently carry out computationally expensive arithmetic within logarithmic space, a special multiple logarithm exponent approximation unit (MLEAU) is introduced. Extended logarithmic arithmetic is carried out according to the functions defined in [27]:

- *eexp(x)* extended exponent,
- *eln(x)* extended logarithm,
- *elnsum(eln(x), eln(y))* extended sum of logarithms,
- *elnproduct(eln(x), eln(y))* extended product of logarithms.

In the above set of functions, the *elnsum (eln(x), eln(y))* is of particular interest. We should note, in this context, that while the Viterbi algorithm can be easily formulated in its logarithmic version (as its induction step involves maximization), the forward-backward algorithm is not straightforward for such a formulation since its induction step involves the summation of probabilities.

Thus, when converting, a useful mathematical expression is needed for the calculation of the logarithm of the sum of probabilities, which themselves are given in terms of logarithms. A solution proposed by Mann tackles the above difficulty. For more details on this, see [27].

The MLEAU dedicated for HMM calculation should provide high performance of the computation and simultaneously high precision of the results for both logarithm and exponent conversion. Therefore, the proposed solution extends the ICSIlog algorithm [28] approach by adding exponentiation approximation functionality and parallel access interfaces. The logarithmic and exponential conversions, as reverse operations, require similar actions. Hence, the calculation of both is based on a single cycle array indexing operation, where the mantissa part of the floating-point argument is transformed using a lookup table (LUT) for a corresponding value. For the logarithmic operation, the logLUT is used to convert the significand part to an appropriate coefficient, while the characteristic

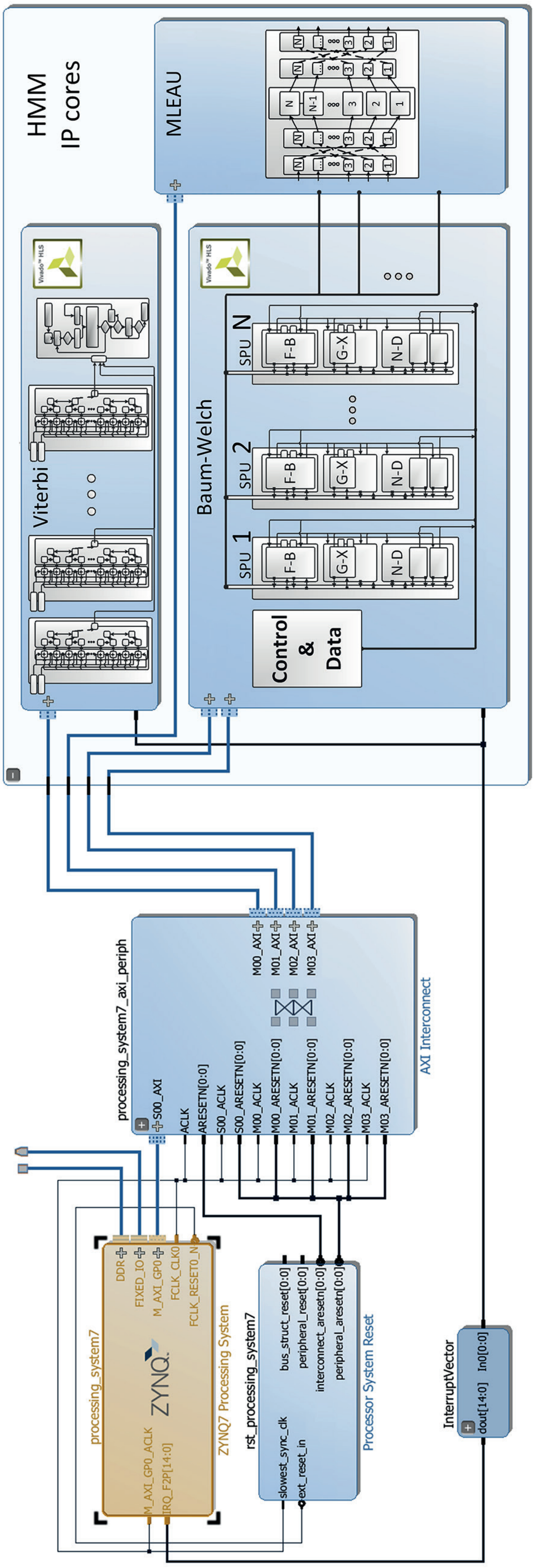

Fig. 1. PSoC computational architecture dedicated for HMMs

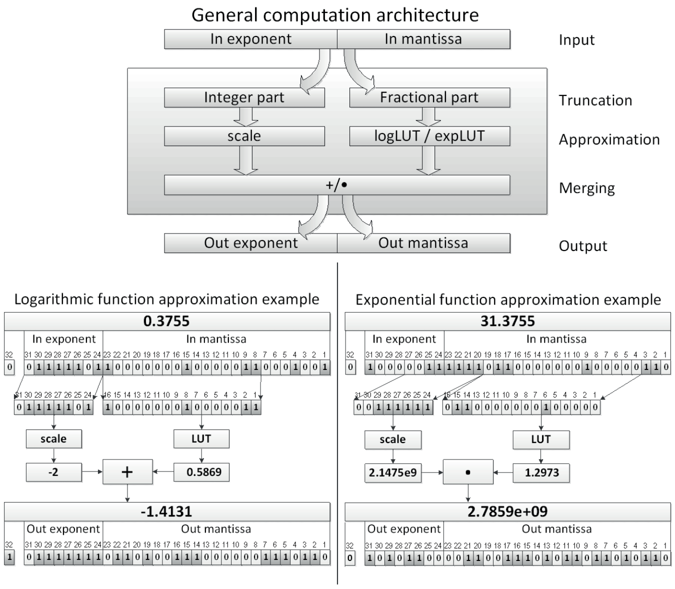

Fig. 2. Logarithm and exponentiation conversion

part is scaled by a constant (which corresponds to the base of the logarithm) and added to the result obtained from the LUT transformation. For the exponentiation operation, the expLUT is used to convert the significand part to an appropriate coefficient while the characteristic part is scaled by a constant (which corresponds to the base of exponentiation) and multiplied by the result obtained from the LUT transformation. The conversion flow is shown in Fig. 2. Both LUTs are addressed with the 16 MSB mantissa bits (which together occupy 256 Kbytes of memory) of input argument and provide 16 bits output value. The output value is cast onto the desired floating-point format (e.g. half, single or custom).

Both LUTs are accessed globally within the MLEAU. The number of access interfaces is fully generically configurable and, in HMM processing, depends on the number of states (each SPU has separate interface access). Standard FPGA resources provide only single or dual port memory. Fully parallel access in programmable logic is provided by distributed memory. However, usage of this memory for huge LUT implementation causes serious timing and routing issues. Hence, to provide parallel access for multiple SPU ports, logLUT and expLUT actually utilize multiple mini LUTs, which are accessible as content-addressable memory (see Fig. 3). Dividing huge single LUT into dozens of mini LUTs significantly improves memory utilization and memory access parallelization. Mini LUTs form a distinctive part of logLUT/expLUT and are selected by the less significant bits (LSB) content part of the input value. Although each mini LUT supports single access (it is built on single port block memory), through near-associative input and output mapping [29], the whole module allows for parallel memory access with certain restrictions. If all input values (each provided from an SPU) have a different LSB part (for content-addressable input decoding), then conversion will be performed immediately. However, if the LSB part is the same in several input values, then the operation will be performed in a queue, causing stall and conversion delay. If some input values are fully the same, then the result will be copied without a stall. The main idea here lies in the assumption that the input values or at least their LSB parts are varied. Absolute conversion stalls in computing the logarithmic version of the forward-backward algorithm were about 30% regardless of the parallelization degree and this is the absolute delay cost of the approach presented herein.

Due to near-associative mapping for input and output, the demand for the FPGA resources increases geometrically as compared to the number of inputs. However, considering the benefits of access parallelization (savings in memory utilization), greater resource utilization is fully acceptable.

3.3. State processing unit. When applying D&C methodology to HMM-related algorithms, special parallel computation architecture is required. The main processing element of such architecture is an SPU, responsible for appropriate computation on a state level where multiple floating-point computation and above all fast logarithm/exponent conversion are performed. Pre-computed logarithms equivalent for probability values of transitions and emissions for a given state are also stored locally in the SPU. The number of SPUs determines the degree of parallelization of the Baum-Welch computation and is dependent on the complexity of the Markov model (i.e. the number of states). Each SPU implements three algorithmic processing blocks: forward-backward, Gamma-Xi, and numerator-denominator, and utilizes eight half-precision floating-point units and two independent MLEAU access interfaces. Only one algorithmic processing block can be selected for current computation. The forward-backward block is active in order to deliver all posteriors over all possible sequences. However, the state posterior and transition posterior are calculated stepwise (in Gamma-Xi block) for the given observation and the partial information thus obtained is delivered to the numera-

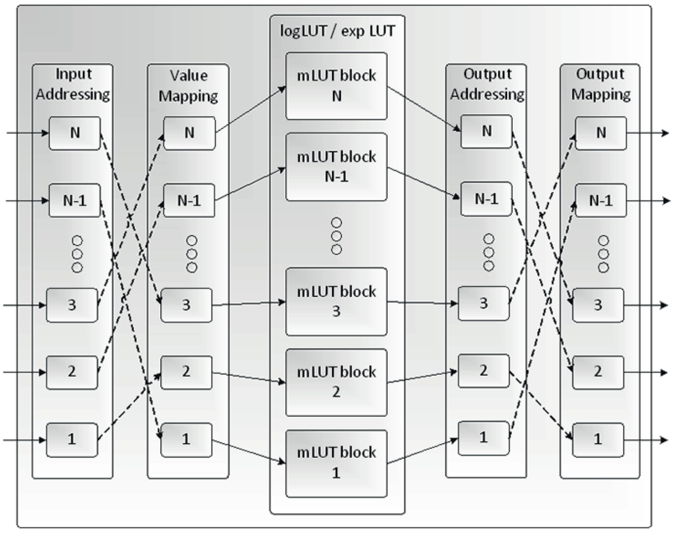

Fig. 3. Near-associative input and output mapping for LUT conversion parallelization

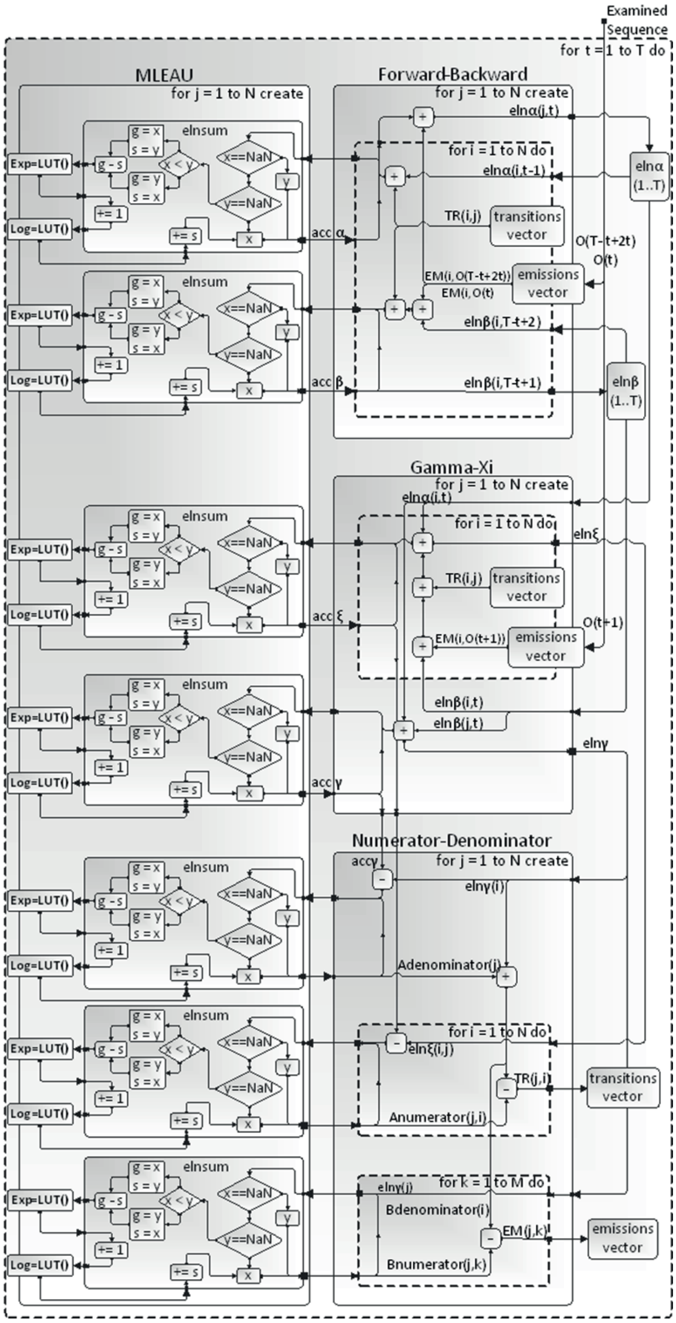

Fig. 4. State processing unit for the forward-backward and Baum-Welch algorithm computations

tor-denominator block. Fig. 4 presents a general outline of SPU processing. For better understanding, not all control signals are shown on the diagram. Moreover, the dashed lines illustrate iterative usage of the resources (e.g. e$ln\alpha(i, t-1)$ and $TR(i, j)$ are added N times where i is iterated to N). All signal names are consistent with those used in [27] (e.g. e$ln\alpha(1 \ldots T)$ refers to the storage of the extended logarithm value of α). The decision flow (related with the appropriate order of arguments and "Not a Number" detection) shown in the figure in each *elnsum* block is essential for the numerical stability of extended logarithmic arithmetic.

3.3.1. Forward-backward algorithm. The advantage of implementing a logarithmic version of the forward-backward algorithm is that scaling factors are not required. The idea of introducing scaling factors only concerned shifting calculation results onto a numerically stable level. However, in logarithmic space the calculations are natively quite numerically stable and no additional precaution is required to make them more so. Without scaling factors, the two steps of forward and backward can be performed simultaneously. From the viewpoint of algorithm acceleration, scaling factors not only require an additional computation effort, but also they need to be stored in the FPGA for (Baum-Welch) further computation throughout the whole observation sequence examined. Computation of the forward-backward algorithm is performed by utilizing SPUs (shown in Fig. 4), where the most important function is the acceleration of the sum of logarithms computation, which involves arithmetic operations in logarithmic space and transition from argument through exponent conversion and successive logarithm conversion. This computationally expensive operation is performed using the MLEAU. Additionally, product operations with corresponding transition and emission probabilities (precomputed logarithm equivalents) are carried out as well. Each forward-backward block produces α and β values of a state corresponding to the observation sequence examined. The α and β vectors are stored in the FPGA for further Gamma-Xi computation.

3.3.2. Gamma-Xi algorithm. The values of γ and ξ are essential for re-estimating the HMM parameters. In the standard approach, these matrices are calculated for the whole observation sequence examined and stored for further computation. From the perspective of FPGA resource utilization, storing an enormous number of intermediate values for the two-dimensional γ matrix, as well as for the ξ matrix, which is three dimensional (i.e. for an observation sequence with 100 elements and HMM with 10 states, $\xi(100, 10, 10)$ will occupy 10 000 half-precision numbers), is problematic. Hence, the Baum-Welch re-estimation algorithm is reorganized to work in iterative steps. The values of γ and ξ are now calculated and normalized for a single time point t and on this basis, a partial numerator-denominator value is computed. Instead of delivering γ and ξ output values for all elements in the observation sequence examined, only a single partial output is needed. However, partial information for the numerator-denominator has to be stored and accumulated separately for each state and for each observation. This means that FPGA memory has to be able to simultaneously store both old HMM parameters (A, B, π) and the new ones (A', B', π'). Final parameter re-estimation is carried out in the numerator-denominator block.

3.3.3. Logarithm numerator-denominator algorithm. After full accumulation of partial information for the numerator-denominator algorithm, final normalization takes place. In the logarithmic space, results normalization is carried out by subtracting the denominator from an accumulated partial numerator for all parameters (A', B', π'). The new HMM parameters are then ready for collection.

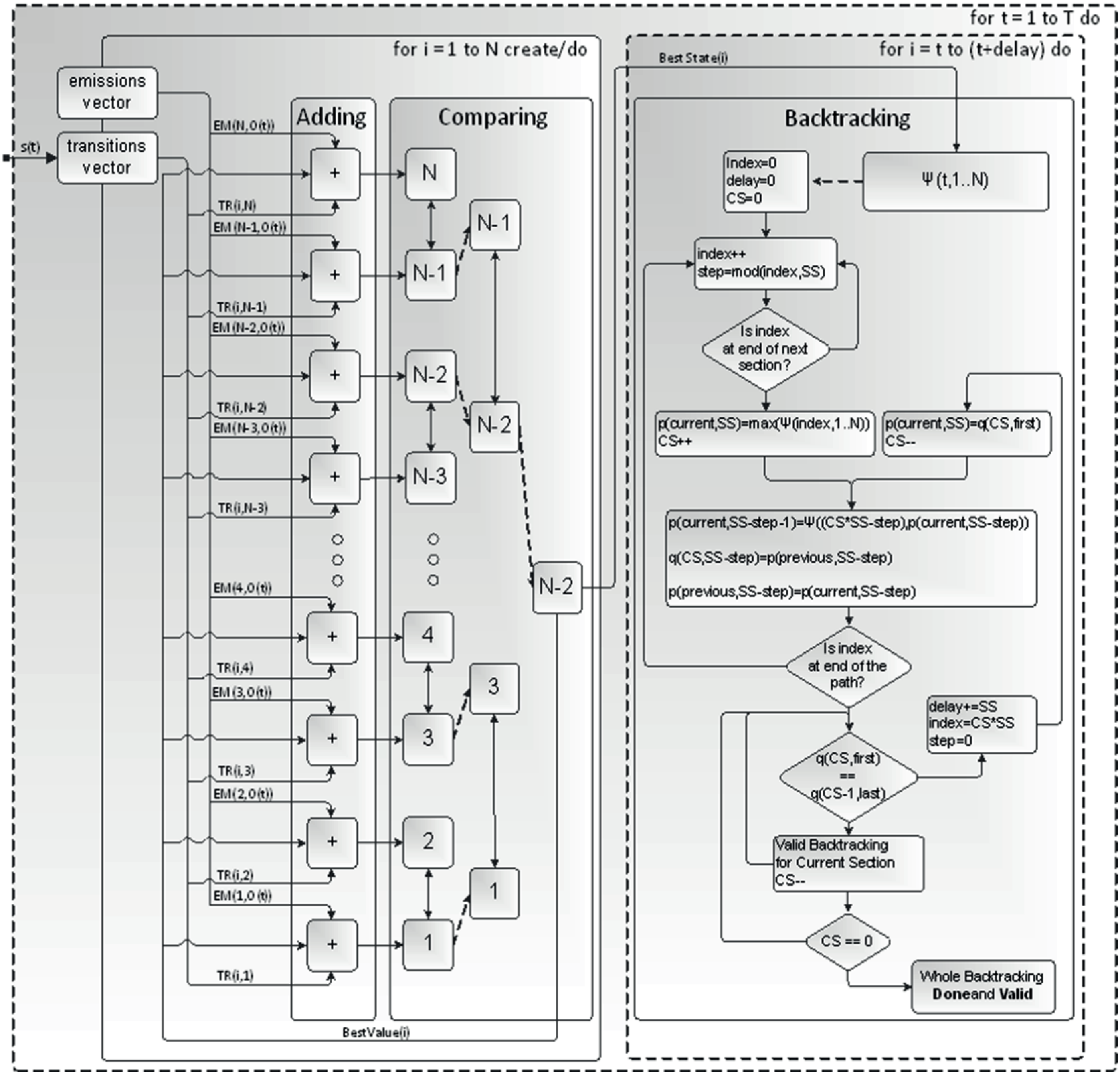

Fig. 5. Viterbi D&C with the backtracking algorithm

3.4. Viterbi D&C algorithm. To avoid numerical issues, the Viterbi algorithm is computed in logarithmic space where the multiplication of transitions, emissions and maximum probabilities is replaced with the summation operation. This is performed parallelly for all states. The last stage in recursive iteration is to find a state with a maximum value of transition probability. Comparing all probability values is fully sufficient for this, but the simple sequential procedure requires n (the number of states) comparisons. To speed up this step, typical D&C methodology is applied. The general idea is presented in Fig. 5, where three blocks are distinguished and dashed lines illustrate iterative usage of the resources. In the "adding" block, triple floating-point addition is performed for the maximum value (from the previous iteration) and the logarithm equivalent of emission and transition probabilities. In the "comparing" block, all states are compared simultaneously in pairs, which reduces the procedure from n steps to $\log_2 n + 1$ steps. The approach here is analogical to the merge-sort algorithm [29], which takes $O(n \log n)$ comparisons to sort all values in the array. In the case of the Viterbi algorithm, only the maximum value in the array is valid, so only $O(n \log_2 n)$ comparisons are needed. These two blocks can be instantiated once or for all states separately. This

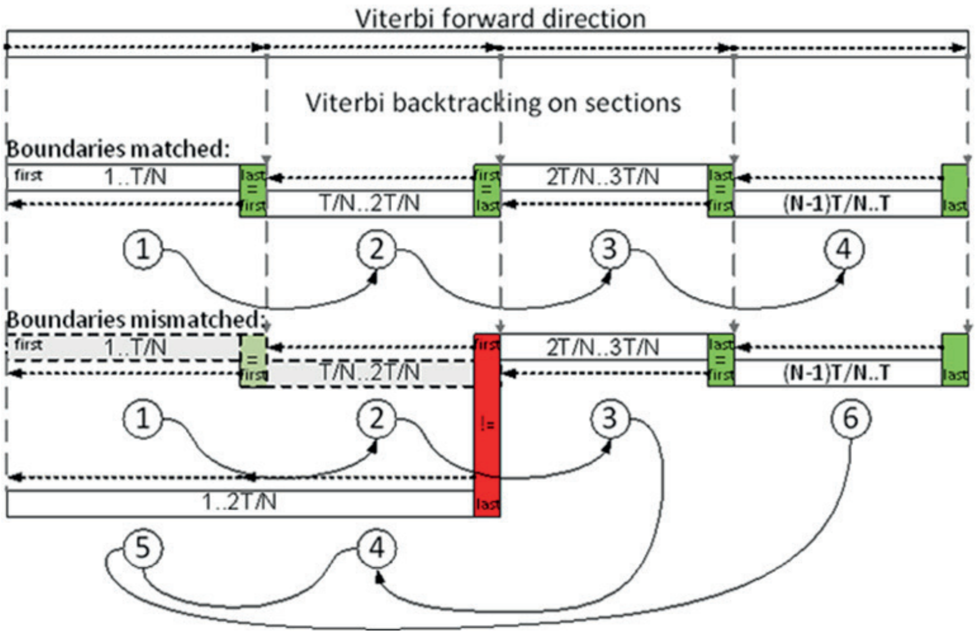

Fig. 6. Backtracking on sections in the case of full match and in the case of section boundaries mismatched

allows for massive parallel processing, although the resource demand increases geometrically as compared to the number of states. For a single instantiation, time division multiplexing is applied and calculations are performed for each state in the iterations.

D&C methodology is also applied in the "backtracking" block in order to reduce the latency associated with tracing back through the entire length of the sequence examined. In this case, backtracking is divided into equal sections (with section size *SS*, e.g. 32). Thus, the procedure is performed on subsequent sections in parallel to the Viterbi recursive iteration. If the boundary elements of the neighboring section are identical, then backtracking for the current section is valid (see Fig. 6). Otherwise, backtracking has to be repeated from the last computed section to the section where the boundary elements are again identical (or to the beginning of the sequence). In the best case scenario, backtracking sectioning produces an additional delay equal to SS steps, whereas in the worst case scenario backtracking has to be performed for the whole sequence, which produces a relevantly longer delay.

4. Numerical stability in the reduced representation of HMM

As already mentioned in [26], HMM computations may become troublesome especially for long sequences. Even if those computations are performed within the logarithmic space, one should be aware that there always exists some limit to the length of sequences, beyond which the computations may become numerically unstable. This phenomenon should be considered along with with the size (width) of the representation for floating-point numbers that is being applied. As mentioned before, for the purposes of FPGA implementation, we postulate reductions from the typical double type (64 bits) representations down to single (32 bits), half (16 bits) or quarter (8 bits). The lemma presented below demonstrates quantitatively how, for an m-bit wide mantissa, the length of the shortest sequence that may potentially be numerically unsafe, can be expressed in terms of extreme probabilities (smallest and largest) that are present in the given HMM.

Lemma 1. Let $p > 0$ and $P > 0$ denote the smallest and largest probabilities, respectively, that are present in an HMM (i.e. extreme probabilities either of transition, emission or initial distribution matrices). Suppose that computations are performed under the logarithmic version of the Viterbi algorithm, and the model is stored in a reduced floating-point representation with mantissa of width *m*. Then, the shortest sequence that may become numerically unsafe – i.e. such which may cause additional errors implied by the reduced representation rather than standard round-off errors only – is of the following length:

$$\Theta = \left(2^{m - (\log_2|\log p| - \log_2|\log P|) + 1}\right). \tag{1}$$

Please be reminded that the *Θ* notation represents the exact asymptotic range (as opposed to the *O* – notation for an upper bound or *Ω* – notation for a lower bound).

Proof: Sketch – we shall arrange a situation where for a certain observation sequence only two Viterbi paths are possible. Those will be paths of very similar probabilities. The initial long fragment of the observation sequence shall indicate a tie between the paths; however, the very last two observations shall break the tie in favor of one of the paths (in the case of exact computations). We shall demonstrate that in the case of numerical computations, under the logarithmic Viterbi algorithm, a contrary outcome (tie-breaker) is possible. Please note: lower-case letters indicate small probabilities (and logarithms associated with them); upper-case letters indicate large probabilities (and their logarithms). Consider the minimum (non-zero) and maximum probabilities present in π, A and B matrices:

$$\begin{aligned} p_\pi &= \min\{\pi_i : 1 \le i \le N, \pi_i > 0\}, & l_\pi &= \log p_\pi, \\ p_A &= \min\{a_{ij} : 1 \le i, j \le N, a_{ij} > 0\}, & l_A &= \log p_A, \\ p_B &= \min\{b_{ik} : 1 \le i \le N, 1 \le k \le M, b_{ik} > 0\}, & & \\ & & l_B &= \log p_B, \\ P_A &= \max\{a_{ij} : 1 \le i, j \le N\}, & L_A &= \log P_A, \\ P_B &= \max\{b_{ik} : 1 \le i \le N, 1 \le k \le M\}, & L_B &= \log p_B, \end{aligned} \tag{2}$$

Now, consider the following particular HMM (Fig. 7), where only states and emissions shown in the figure are relevant. Note that transitions between states S_1 and S_2 are not possible, $a_{12} = a_{21} = 0$; and that all other states $S_{i>6}$ (omitted in figure) have zero probability of emission observations: O_1, O_2, O_3. Apart from transitions shown in figure, states $S_1,\ldots,S_6$ are able to transit to other states $S_{i>6}$, yet such transitions are irrelevant for the proof. Emission probabilities q and r can be treated as parameters that can be suitably chosen for breaking the tie. Let us denote their logarithms as $l_q = \log q$ and $l_r = \log r$.

Now, suppose the following observation sequence is given as the input:

$$(\underbrace{O_1, O_1, \ldots, O_1}_{T-2}, O_2, O_3). \tag{3}$$

It is easy to see that only two paths of states could have produced the sequence above. They are:

$$\begin{gathered} (S_1, S_1, \ldots, S_1, S_3, S_5), \\ (S_1, S_1, \ldots, S_2, S_4, S_6). \end{gathered} \tag{4}$$

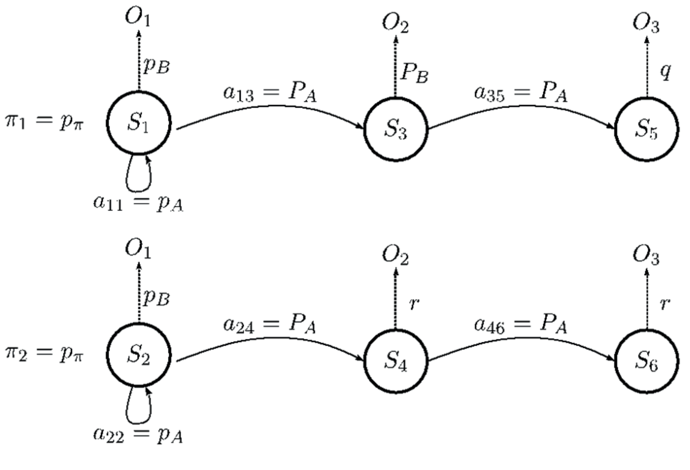

Fig. 7. Model for HMM lemma

Let us call them "odd" and "even" paths, respectively (because of the indexes involved). The product of probabilities and the corresponding sum of logarithms for the "odd" path are, respectively, equal to:

$$\begin{gathered} \underbrace{p_\pi \cdot p_B}_{t=1} \cdot \underbrace{p_A \cdot p_B}_{t=2} \cdot \ldots \cdot \underbrace{p_A \cdot p_B}_{t=T-2} \cdot \underbrace{P_A \cdot P_B}_{t=T-1} \cdot \underbrace{P_A \cdot q}_{t=T}, \\ \underbrace{l_\pi + l_B}_{t=1} + \underbrace{l_A + l_B}_{t=2} + \ldots + \underbrace{l_A + l_B}_{t=T-2} + \underbrace{L_A + L_B}_{t=T-1} + \underbrace{L_A + l_q}_{t=T}. \end{gathered} \tag{5}$$

Analogically, for the "even" path we have:

$$\begin{gathered} \underbrace{p_\pi \cdot p_B}_{t=1} \cdot \underbrace{p_A \cdot p_B}_{t=2} \cdot \ldots \cdot \underbrace{p_A \cdot p_B}_{t=T-2} \cdot \underbrace{P_A \cdot r}_{t=T-1} \cdot \underbrace{P_A \cdot r}_{t=T}, \\ \underbrace{l_\pi + l_B}_{t=1} + \underbrace{l_A + l_B}_{t=2} + \ldots + \underbrace{l_A + l_B}_{t=T-2} + \underbrace{L_A + L_r}_{t=T-1} + \underbrace{L_A + l_r}_{t=T}. \end{gathered} \tag{6}$$

Let us suppose that under exact calculation it is the "even" path that is actually slightly more probable (and should be returned as the result). This means that the following inequality must be satisfied (by reduction to the crucial end part, once the initial equal terms are cancelled out):

$$P_B \cdot q < r^2. \tag{7}$$

Simultaneously, we would like to show a possibility of such a numerical error that the algorithm indicates incorrectly the "odd" path as the winning one. That error can happen when at time moment $t = T - 1$ the addition of the L_B summand (being close to zero, since P_B is maximal) to the running sum (which in turn is suitably large) shall not alter the result. Therefore, we would like to induce a contrary direction of the inequality (7) by replacing the P_B term with a one due to the indistinguishable numerical result, i.e.:

$$1 \cdot q > r^2, \tag{8}$$

which in turn, when looking at the sum of logarithms, corresponds to:

$$\log 1 + \log q > 2 \log r. \tag{9}$$

Therefore, in order to satisfy (7) and (8) simultaneously, one has to choose q to be:

$$r^2 < q < {r^2}/{P_B}. \tag{10}$$

To simplify considerations, let us neglect the distinction between probabilities coming from different matrices (A or B or π; i.e. let us identify all of them as follows:$p_A = p_B = p_\pi = p$ and $P_A = P_B = P_\pi = P$. Thus, let us also assume that: $l_A = l_B = l_\pi = l$ and $L_A = L_B = L_\pi = L$. In other words, we are now interested only in small and large terms (in the sense of order of magnitude) that are present in the sum of logarithms. For the “odd” path, sum (4) is now simplified to:

$$\underset{t=1}{l} + l + \underset{t=2}{l} + l + \ldots + \underset{t=T-2}{l + l} + \underset{t=T-1}{L + L} + \underset{t=T}{L + l_q}, \tag{11}$$

whereas sum (6) for the “even” path is simplified to:

$$\underset{t=1}{l} + l + \underset{t=2}{l} + l + \ldots + \underset{t=T-2}{l + l} + \underset{t=T-1}{L + L} + \underset{t=T}{L + l_r}. \tag{12}$$

Just before the critical time moment $t = T - 1$ the current value of both sums above arises from additions of $2(T - 2)$ logarithms of small probabilities, that is values being of large order in the absolute sense; and it is easy to impose $|l| \gg |L|$. Hence, we try to exhaust, at “fast tempo”, *m* of significant digits (under base of 2) of the mantissa just before the *L* summand is to be added. Note that *L* is in turn of small order. The value of $\log_2|L|$ can in fact be treated as *the order of magnitude* of the number *L* under base 2. Finally, the wanted error situation – consisting of an indistinguishable result even though *L* has been added – can be described by the following inequality:

$$\frac{2^{m - \left|\log_2|L|\right| + 1}}{2(T-2)|l|} \leq 1. \tag{13}$$

The numerator in the fraction represents the order of magnitude for the largest number representable under a mantissa of width *m* once we subtract from it $\left|\log_2|L|\right|$, i.e. significant digits needed by the critical summand. On the other hand, the denominator represents the “tempo” at which we try to exhaust the order of magnitude at our disposal by adding $2(T - 2)$ summands equal to *l* during the initial stage of computations. Solving (13) with respect to *T* (length of sequence), we obtain:

$$\begin{aligned} T &\geq 2 + \frac{2^{m - \left|\log_2|L|\right| + 1}}{2|l|} \\ &= 2 + 2^{m + \log_2|L| + 1} \cdot 2^{\log_2|2l|^{-1}} \\ &= 2 + 2^{m + \log_2|L| - \log_2|2l| + 1} \\ &= 2 + 2^{m - \log_2|2l| - \log_2|L| + 1} = T^*. \end{aligned} \tag{14}$$

Up to now, we have shown that there can exist numerically unsafe sequences of length greater or equal to a certain value, i.e. we have shown the case of: $T > T^*$. Taking into account the sense of the Θ – notation in the lemma, we still need to show the case of $T \leq T^*$ to conclude the proof. By doing so the minimum sequence from all such unsafe sequences shall be the upper bound. It is sufficient to note that if, in our example in its initial fragment, we apply probabilities larger than the minimum *p*, then the logarithms being added shall not exhaust the numerical range at disposal so fast. This implies that the described error situation can only occur later. Therefore, the shortest unsafe sequence is of length $\Theta(T^*)$. □

The following two additional remarks can be given in the context of the lemma.

Remark 1. The formula for T^* (14) shows that the important element is the difference between the orders of magnitudes of the logarithms (being the consequence of extreme probabilities in an HMM). That difference is: $\log_2|2l| - \log_2|L|$. The greater the difference, the fewer free digits there remain in the mantissa.

Remark 2. While the exterior logarithms, yielding the order of magnitude, must be base-2 logarithms, the internal (nested) logarithms hidden inside $l = \log p$ and $l = \log P$ can be expressed in any base. We leave it as a simple exercise for the reader.

A certain precision of numerical representation restricts the HMM structure and possibilities of data processing. This phenomenon is particularly observable for very low precision. However, in order to secure the stability of numerical calculations, some crucial constraints arising from precision reduction have to be considered. In particular, for the *m*-bit mantissa, the length of the shortest numerically unstable sequence can be expressed in terms of extreme probabilities ($p > 0$ being the smallest, $P > 0$ being the greatest) occurring in the HMM, and this length is of the following order:

$$2^{m - \left(\log_2|\log p| - \log_2|\log P|\right) + 1}. \tag{15}$$

4.1. Example of HMM boundaries determination. To present a general idea, let us consider a synthetic case of an HMM with 30 states and 1000 observations. The parameter estimation is carried out on a sequence of 100 000 elements generated synthetically by the following formula:

$$\begin{aligned} q_t &= \{t \bmod (t \bmod N)\}, & 1 \leq t \leq T, \\ o_t &= \{t \bmod (t \bmod M)\}, & 1 \leq t \leq T. \end{aligned} \tag{16}$$

The probabilities of transition *A* and emission *B* are obtained by calculating maximum likelihood based on the sequences generated. The obtained maximum *P* and minimum *p* of non-zero probabilities for both matrices are as follows:

$$\begin{aligned} P_A &= 1.00, & p_A &= 6.111e - 4, \\ P_B &= 43.478e - 3, & p_B &= 1.170e - 4. \end{aligned} \tag{17}$$

Knowing the necessary conditions for the numerical stability of HMM calculations (shown in this paper), it is possible to meet the requirements of the application. For an HMM with the given transition and emission matrices, there is a minimum precision of floating-point number representation for which the model is numerically stable. Moreover, in terms of decoding state paths, there is a maximum length of observation sequence, on which the calculations carried out (sum and logarithm accumulation) will not lead to arithmetic overflow and related result indistinguishability.

By reducing the precision of number representation from double (64 bit) to half (16 bit), all probabilities and corresponding logarithm values will be rounded to fit the new format. To ensure numerically stable computation in reduced half-precision number representation, formula (14) has to be applied. Hence, the maximum length of the observation sequences examined is limited to **228** elements. More examples of applying the lemma given in this paper and the resulting numerically stable length for different probability sets calculated at different precisions of floating-point number are presented in Table 1.

Table 1
Evaluation of numerically stable length at different precision of numbers representation

Probabilities set [*p*, *P*]	Half$_{(16\ \text{bit})}$ precision	Single$_{(32\ \text{bit})}$ precision	Double$_{(64\ \text{bit})}$ precision
p = 0.5e – 2; P = 0.5e – 2	1927	15785253	stable
p = 1.17e-4; P = 1.0	**228**	**1863880**	stable
p = 1.0e – 6; P=1.0e – 1	10	75915	stable
p = 1.0e – 3; P=1.0e – 3	1	2372	stable
p = 2.0e – 7; P=1.0e – 6	0	2	556885799

The numerically stable length depends on the probability set and the precision of number representation. HMM computation in double precision can be treated as numerically stable. However, in acceleration systems (also GPUs as in [31–32]), where single and half-precision computations are widely used, the length constraints have to be applied to the observation sequence in order to ensure numerical stability.

4.2. Practical application example. The methodologies presented in this paper were applied to the practical application of syntactic and semantic text analysis, where HMM were trained on the Penn Treebank database. As a result of training, the model contains 47 states related with syntactic category and 9055 observations related with separated expressions (more detailed information about the methodology and the models obtained can be found in [33]). Obtained maximum P and minimum p of non-zero probabilities for matrices of HMM are as follows:

$$\begin{aligned} P_A &= 0.7492, \quad p_A = 0.0292, \\ P_B &= 0.6042, \quad p_B = 6.1277E - 10. \end{aligned} \tag{18}$$

By reducing the model representation to half-precision, the maximum length of examined observation sequences (for numerically stable computation) is then theoretically limited to **49** elements. By the examination of an observation test sequence, after **76** elements the result was indeed indistinguishable, as it is correctly revealed by formula (14).

5. Results and design evaluation

The computation methodology presented in this paper is dedicated for HMM algorithm acceleration in low latency applications. The entire implementation has been carried out on PSoC Zynq from Xilinx. Hence, the comparison concerns the performance achieved on an embedded system with ARM Cortex-A9, with and without FPGA acceleration and on a typical workstation with Intel Core i7. On both CPUs, the same piece of software (without any special optimization) was evaluated. The performance of HMM-related algorithms was measured by computing a synthetic HMM structure with 30 states and 1000 observations and an observation sequence of length $T = 200$ for numerically stable computation. The time, needed for complete examination of a given observations sequence by a specified model, is used as a performance indicator. This approach gives a good performance approximation in the perspective of a real application.

All modules were verified by behavioral simulation in a Vivado HLS environment, where test goals were to compare the results obtained by the algorithms compiled to RTL with the results of equivalent functions written in a high-level language like C/C++. Additionally, the MLEAU, as a very fast logarithm and exponent approximation unit, is subject to separate evaluation.

5.1. Baum-Welch. The logarithmic version of the Baum-Welch algorithm was computed on three different systems. The CPU-based systems (Intel Core i7 4770 and ARM Cortex-A9) use single (32 bits) precision, while the FPGA-based acceleration uses half precision (16 bits) floating-point number representation. The numerical representation for each system was chosen in order to achieve the best performance. Despite the differences in number representation, it is definite (in the sense of formula (14)) that the results are numerically stable. For both CPUs (Intel and ARM), computation was performed based on a code compiled with gcc 4.6 for a single core. The FPGA system is clocked with 200MHz. Performance comparison in the Baum-Welch algorithm computation is presented in Table 2.

Table 2
Performance comparison in Baum-Welch algorithm computation, time in [ms]

Baum-Welch stages	Intel Core i7 4770	ARM Cortex-A9	FPGA D&C acceleration
Forward-backward	31.9562	345.6694	3.3060
Gamma-Xi	17.7534	195.0388	2.1360
Numerator-denominator	16.4776	176.9418	1.5893
Whole Baum-Welch	66.1872	717.6500	7.0313 (4.8953)

5.2. Viterbi. The configuration for comparing the CPUs is the same as in the Baum-Welch algorithm evaluation. However, FPGA acceleration is different for D&C acceleration, where the Viterbi D&C module (presented in Fig. 5) is instantiated only once and computation is multiplexed for all states, and for full parallelization, where the Viterbi D&C module is instantiated for each state and computation is performed fully parallel for each state. Path backtracking is executed for the observation sequence of 200 in length. The calculation time is relative to length (or number of chunks) of the observation sequence. Performance comparison in the Viterbi algorithm computation is presented in Table 3.

Table 3
Performance comparison in Viterbi algorithm computation, time in [ms]

Viterbi stages	Intel Core i7 4770	ARM Cortex-A9	FPGA D&C acceleration	FPGA full parallelization
Initialization	1.3236	9.3750	0.7875	0.0158
Recursion	3.2766	15.2877	0.9510	0.0210
Backtracking	1.76e – 4	7.50e – 4	3.20e – 4	3.20e – 4
Whole Viterbi	4.6004	24.6635	1.7388	0.0371

5.3. MLEAU. Logarithm and exponent conversion requires computationally expensive operations. In HMM-related logarithmic space calculations the most important operation is the extended sum of logarithms, which involves a sequence of extended operations: logarithm comparison, pre-product of logarithms, conversion to exponent, conversion to logarithm and post-product of logarithms. Hence, for performance evaluation, the execution time for logarithm conversion and for the extended sum of logarithms (*elnsum*) on different platforms is measured. Configuration for the CPUs compared is the same as in the Baum-Welch algorithm evaluation. The MLEAU implemented in the FPGA is clocked with a 200 MHz clock, and in order to show the performance on different parallelization degrees, the number of interfaces is set to 1, 8 and 16 (see Fig. 8).

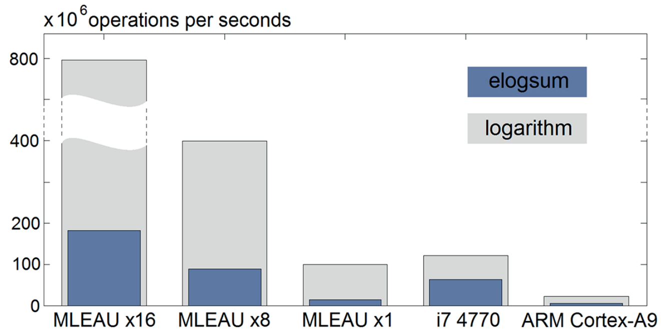

Fig. 8. Performance comparison of extend logarithm arithmetic

5.4. Synthesis and resources utilization. Designing the computation architecture at high abstraction level results in RTL description which is strongly dependent on the HLS directives applied. It is possible to obtain either a resources-effective structure, which performs the calculations more sequentially, using multiplexing techniques, or a more demanding structure with a higher degree of parallelization. In this paper, the degree of parallelization is equal to the number of states in the calculated model (i.e. HMM with 30 states and 1000 observations). Hence, Table 4 presents a summary of resource utilization of Baum-Welch, Viterbi and MLEAU×16 algorithmic blocks designated for the exemplary model.

Table 4
Resource utilization for Baum-Welch, Viterbi and MLEAU blocks

Name / Resource	Viterbi block	Baum-Welch block	MLEAU block
BRAM_18K	32	219	256
DSP48E	70	147	48
Flip-flops	21002	54477	4524
LUT	35175	91014	7607

The logic complexity is strongly related with the size of the HMM. Although utilization can vary, depending on directives and synthesis optimization settings, in general to fully use the potential of parallelism in the solution presented, a high capacity FPGA is recommended.

6. Discussion

The presented implementation minimizes the cost of numerically stable logarithmic space calculations. However, FPGA implementation utilizes reduced (half-precision) floating-point number representation and for this reason, the length of examined observation sequence is limited by formula (14). After all, storing and performing a computation for a long observation sequence inside a FPGA is impractical. Programmable logic is expected to speed up the computation through process parallelization. Hence, disassembling the observation sequence into shorter chunks in order to perform hardware accelerated computation with a result identical (from the perspective of the Viterbi path and HMM parameter re-estimation) to standard computation (in double precision) is fully acceptable.

The computational architecture presented is based on SPUs whose number depends on the complexity of the HMM (number of states) but is limited by the FPGA resources available. Each SPU utilizes 8 floating-point units (each FPU utilizes 2 DSP slices), 2 independent MLEAU access interfaces, additional flow control logic and local memory blocks for transition and emission matrices. Theoretically, in PSoC Zynq Z-7100 it is possible to implement up to 128 SPUs, however due to routing and timing issues that arise, a maximum of 64 SPUs is recommended. For more complex HMMs, the architecture should support time division multiplexing with a sophisticated flow controller. An FPGA full parallelization of the Viterbi algorithm is a very resource-consuming solution, but allows for even hundred-times acceleration to be achieved. Computing the best transition for the HMM evaluated for a single state on a given element of observation sequence took approx. 100 ns. In embedded, automotive and robotic applications, where low latency, real-time computation is required, that kind of speed-up of the Viterbi algorithm certainly allows for the processing of much more data (e.g. IoT sensors or video data). The numerically stable computation of HMM algorithms in reduced (half-precision) floating-point number representation can be secured on any acceleration system (e.g. GPGPU) by applying the lemma and above all the formula (14) presented in this paper. Despite the fact that numerical stability will be preserved for the sequence length in the sense of the lemma, still the Viterbi decoding accuracy may be negatively affected due to round-off error associated with the precision reductions of HMM representation.

During the authors' research about the applicability of reduced precision HMM in the context of natural language processing and syntactic analysis, it has been noticed that strong precision reduction leads to erroneous recognition results. Nevertheless, this paper is concerned with the HMM hardware implementation and not the applicability issues, and therefore wider discussion is omitted here.

7. Conclusion

This paper explained the most important aspects of hardware implementation and parallelization of HMM algorithms concerning D&C methodology for Baum-Welch and Viterbi algorithms. The full parallelization FPGA implementation of the widely used Viterbi algorithm presented herein is suitable for low-latency detection and recognition systems, and its peak performance surpasses the software equivalent implementations. Moreover, for computationally expensive arithmetic operations in logarithmic space, a special MLEAU has been introduced and utilized in SPU computations. The MLEAU reduces the cost of logarithmic space calculations and it is not limited only to HMMs. The main emphasis of this paper is the numerical stability of reduced precision HMMs computed fully in logarithmic space. For this purpose, the lemma concerned with determining the maximum length of observation sequences for numerically stable computation has been comprehensively described. The theoretical basis presented reveals computation boundaries related with probabilities and floating-point representation of HMMs, which in practice allows for securing numerical stability of computation even with significant reduction in the precision.

Although the data flow and control functions in the system are imposed by the Vivado HLS framework, still by applying the relevant directives (for architectural constraints) the results achieved are very satisfying. The results showed that this FPGA-acceleration has a positive impact on the performance, especially for processors designed for mobile devices (such as ARM). Nevertheless, there are still many optimization possibilities, especially at the level of arithmetic calculations, where by the use of dedicated solutions (fast FPUs or partial fixed-point arithmetic) it would be possible to increase the performance. These ideas are, however, potential subjects for future research.

Acknowledgements. This work was supported in part by a grant from the West Pomeranian University of Technology in Poland.

References

[1] J.-F. Mari and F. L. Ber, "Temporal and spatial data mining with second-order hidden Markov models", *Soft Computing* 10 (5), 406–414, 2005. doi 10.1007/s00500–005–0501–0

[2] A. Panuccio, M. Bicego, and V. Murino, "A hidden Markov model-based approach to sequential data clustering", *Structural, Syntactic, and Statistical Pattern Recognition*, 734–743, 2002. doi 10.1007/3–540–70659–3_77

[3] M. Narasimhan, P.A. Viola, and M. Shilman, "Online decoding of Markov models under latency constraints", in ICML, pp. 657–664, 2006. doi 10.1145/1143844.1143927

[4] R.E.F. Behnam, "Stats-calculus pose descriptor feeding a discrete HMM low-latency detection and recognition system for 3D skeletal actions", arXiv preprint arXiv:1509.09014, 2015.

[5] M. Shannon, H. Zen, and W. Byrne, "Autoregressive models for statistical parametric speech synthesis", *EEE Trans. Audio Speech Language Process.* 21, 587–597, 2013. doi 10.1109/tasl.2012.2227740

[6] M. Kubanek, J. Bobulski, and L. Adrjanowicz, "Characteristics of the use of coupled hidden Markov models for audio-visual Polish speech recognition", *Bull. Pol. Ac.: Tech.* 60 (2), 307–316, 2012. doi 10.2478/v10175–012–0041–6

[7] A. Mannini, V. Genovese, and A. M. Sabatini, "Online decoding of hidden Markov models for gait event detection using foot-mounted gyroscopes", *IEEE Journal of Biomedical and Health Informatics* 18 (4), 1122–1130, 2014. doi 10.1109/jbhi.2013.2293887

[8] A. Manandhar, P.A. Torrione, L.M. Collins, and K.D. Morton, "Multiple-instance hidden Markov model for gpr-based landmine detection", *IEEE Transactions on Geoscience and Remote Sensing* 53 (4), 1737–1745, 2015. doi 10.1109/tgrs.2014.2346954

[9] K. Taehwan and B. Keunsung, "HMM-based underwater target classification with synthesized active sonar signals", *IEICE Trans. Fundamentals* E94-A (10, 2039–2042, 2011. doi 10.1587/transfun.e94.a.2039

[10] U.K. Singh, V. Padmanabhan, and A. Agarwal, "Dynamic classification of ballistic missiles using neural networks and hidden Markov models", *Applied Soft Computing* 19, 280–289, 2014. doi 10.1016/j.asoc.2014.02.015

[11] O.A. Bapat, R.M. Fastow, and J. Olson, "Acoustic coprocessor for hmm based embedded speech recognition systems", *IEEE Transactions on Consumer Electronics* 59 (3), 629–633, 2013. doi 10.1109/TCE.2013.6626249

[12] Y. Choi, K. You, J. Choi, and W. Sung, "A real-time FPGA-based 20,000-word speech recognizer with optimized DRAM access", *IEEE Trans. Circuits Syst. I, Reg. Papers* 57 (8), 2119–2131, 2010. doi 10.1109/tcsi.2010.2041501

[13] C.H. Hsieh, H.P. Chu, and Y. H. Huang, "An HMM-based eye movement detection system using EEG brain-computer interface", *2014 IEEE International Symposium on Circuits and Systems* (ISCAS), 662–665, Melbourne VIC, 2014. doi 10.1109/ISCAS.2014.6865222

[14] Y. Lyu, Q. Pan, C. Zhao, H. Zhu, T. Tang, and Y. Zhang, "A vision based sense and avoid system for small unmanned helicopter", *2015 International Conference on Unmanned Aircraft Systems* (ICUAS), 586–592, Denver, 2015. doi 10.1109/ICUAS.2015.7152339

[15] L.R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition", *Proceedings of the IEEE* 77 (2), 257–286, 1989. doi 10.1016/b978–0–08–051584–7.50027–9

[16] D.G. Brown and D. Golod, "A tutorial of techniques for improving standard hidden Markov model algorithms", J. Bioinformatics and Computational Biology 7, 737–754, 2009. doi 10.1142/s0219720009004242

[17] G. Chrysos et al., "Opportunities from the use of FPGAs as platforms for bioinformatics algorithms", *International Conference on Bioinformatics & Bioengineering* (BIBE), 559–565, Larnaca, 2012. doi 10.1109/BIBE.2012.6399733

[18] I. Atef, E. Hamed, A. Abdullah, and G. Fayez, "Reconfigurable hardware accelerator for profile hidden Markov models", *Arabian Journal for Science and Engineering* 41 (8), 3267–3277, 2016. doi 10.1007/s13369–016–2162-y

[19] F.L. Vargas, R.D.R. Fagundes, and D.B. Junior, "A FPGA-based Viterbi algorithm implementation for speech recognition systems", *IEEE International Conference on Acoustics, Speech, and Signal Processing* (ICASSP'01) 2, 1217–1220, 2001. doi 10.1109/ICASSP.2001.941143

[20] Y. Sun, P. Li, G. Gu, Y. Wen, Y. Liu, and D. Liu, "Accelerating HMMer on FPGAs using systolic array based architecture", *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium* (IPDPS '09), 2009. doi 10.1109/ipdps.2009.5160927

[21] T. Majumder, P. Pande, and A. Kalyanaraman, "Hardware accelerators in computational biology: application, potential, and challenges", *IEEE Design & Tes*t 31 (1), 8–18, 2014. doi 10.1109/MDAT.2013.2290118

[22] A. Churbanov and S. Winters-Hilt, "Implementing EM and Viterbi algorithms for hidden Markov model in linear memory", *BMC Bioinformatics* 9, 1–15, 2008. doi 10.1186/1471–2105–9–224

[23] U.R. Tatavarty, "Implementation of numerically stable hidden Markov model", UNLV Theses, Dissertations, Professional Papers, and Capstones. Paper 1018, 2011.

[24] D. Varma, D. Mackay, and P. Thiruchelvam, "Easing the verification bottleneck using high level synthesis", *28th VLSI Test Symposium* (VTS), 253–254, Santa Cruz, 2010. doi 10.1109/VTS.2010.5469565

[25] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan, "Xilinx vivado high level synthesis: case studies", *Irish Signals and Systems Conference 2014*, Limerick, 352–356, 2014. doi 10.1049/cp.2014.0713

[26] A.T. Anisha and C. Sunitha, "A hybrid Parts Of Speech tagger for Malayalam language", *International Conference on Advances in Computing, Communications and Informatics*, 1502–1507, Kochi, 2015. doi 10.1109/ICACCI.2015.7275825

[27] T.P. Mann, "Numerically stable hidden Markov model implementation", HMM scaling tutorial, 1–8, 2006.

[28] O. Vinyals and G. Friedland, "A Hardware-independent fast logarithm approximation with adjustable accuracy", *IEEE International Symposium on Multimedia*, 61–65. Berkeley, 2008. doi 10.1109/ISM.2008.83

[29] U. Dhawan and A. DeHon, "Area-efficient near-associative memories on FPGAs", *ACM Transactions on Reconfigurable Technology and Systems* 7 (4), 30–41, 2015. doi 10.1145/2435264.2435298

[30] D. Knuth, "The art of computer programming", Section 5.2.4: Sorting by Merging. Sorting and Searching. 2nd ed. Addison-Wesley. 158–168, 1998.

[31] Jun Li, Shuangping Chen, and Yanhui Li. "The fast evaluation of hidden Markov models on GPU", *IEEE International Conference on Intelligent Computing and Intelligent Systems*, 426–430, Shanghai, 2009. doi 10.1109/icicisys.2009.5357649

[32] L. Yu, Y. Ukidave, and D. Kaeli, "GPU-Accelerated HMM for speech recognition", *43rd International Conference on Parallel Processing Workshops*, Minneapolis, 395–402, 2014. doi 10.1109/ICPPW.2014.59

[33] M. Pietras, "Hidden Markov models with affix based observation in the field of syntactic analysis", In: Hard and Soft Computing for Artificial Intelligence, Multimedia and Security, S. Kobayashi, A. Piegat, J. Pejaś, I. El Fray, J. Kacprzyk J. (eds) 534, 17–26, Springer, Cham, 2016. doi 10.1007/978–3–319–48429–7_2