FAuto: An Efficient GMM-HMM FPGA Implementation for Behavior Estimation in Autonomous Systems

Junde Li, Navyata Gattu, Swaroop Ghosh
School of Electrical Engineering and Computer Science
Pennsylvania State University, University Park, PA 16802 USA
{jul1512, nug323, szg212}@psu.edu

***Abstract*—Driving behavior estimation in car-following scenario based on contextual traffic information is an essential capability for autonomous driving systems. Real-time motion planning based on incomplete environment perception requires complicated probabilistic model for interactions with surrounding objects and road conditions. Hidden Markov Model (HMM) with Gaussian emissions has been used to model driving behaviors for its ability of inferring unobserved states. While the high-dimensional contextual data is continuously processed, the system should be high-performance and power-efficient to make real-time decisions for safe operations. Field Programmable Gate Array (FPGA) is being increasingly used on embedded System-on-Chip (SoC) for mobile applications mainly because of its parallel computation and low-power consumption. This paper implements *FAuto*: the framework of HMM coupled with GMM algorithm on a Xilinx PYNQ-Z2 board for autonomous systems. We design the hybrid GMM-HMM model in python, and train the model using Next Generation SIMulation (NGSIM) trajectory data on a CPU platform. The hardware accelerator is designed through Vivado HLS 2018.2, and verified with Jupiter notebook. *FAuto* achieves 2.59 TOPS/W power efficiency, and 10.39x speedup compared to Python software implementation running on quad-core i7-7500U CPU.**

I. Introduction

Safety and reliability are top priorities for autonomous driving. Accurate driving behavior estimation for motion planning is the key to achieve the above goal. One particular example is estimating lane keep and lane change behavior from target vehicles that should be continuously estimated either for human drivers or self-driving cars. Fig. 1 shows the basic idea of behavior estimation based on contextual traffic information. Host vehicle (Veh_h) extracts contextual features such as longitudinal and lateral velocity of target vehicle (Veh_t), distance and speed differences between target vehicle and surrounding vehicles to decide whether to change lane or keep it.

Such intention prediction problem has been addressed by using classification algorithms, such as Support Vector Machine [1], Quantile Regression Forests [2], and Hidden Markov Model (HMM) [3]. Further, it has been claimed [4] that two HMMs with Gaussian emissions performs better for lane change or lane keep estimation because of HMM's capability for handling temporal contextual information. Limited by training difficulty caused by NGSIM data imbalance between

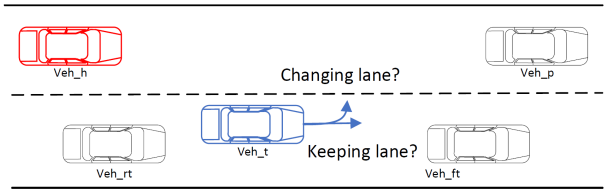

Fig. 1. Driving behavior estimation based on contextual traffic information. The self-driving car is the red-colored vehicle denoted by Veh_h; and Veh_t denotes the target vehicle whose behavior intention needs to be estimated using features from surrounding vehicles and itself.

lane change and keep scenarios, and possible vanishing gradient issue for large sequence length, recurrent neural network of high model complexity is not considered in this study. Therefore, the hybrid Hidden Markov Model with Gaussian emissions (GMM-HMM) is focused on in this work for software and hardware implementation. Fig. 2 shows the high-level GMM-HMM framework for driving behavior estimation. For both lane change and lane keep scenarios, contextual traffic information is extracted, then fed into Gaussian Mixture Model (GMM) for outputting emission probability b_j for each hidden state. Finally, the b_j combined with transition probability a_{ij} are processed for obtaining the respective lane change and keep probabilities.

The mission critical nature of the autonomous vehicles demands timely estimation of behavior and planning for safe operation. Software implementation of behavior estimation is slow and power intensive. Efficient hardware accelerators can address both of these bottlenecks. This observation aligns well with the recent trend of adoption of accelerators in the CPUs and GPUs to address challenges faced by autonomous systems. Few examples include NVidia's ray tracing accelerator using RTX technique in GPUs [5] and Intel's Vision accelerator using DLDT kit in Intel Arria 10 FPGAs [6].

Field Programmable Gate Array (FPGA) has been increasingly exploited in embedded systems due to customized reconfiguration, high parallelism and power efficiency. FPGA has been used for pedestrian detection in autonomous vehicles, and real-time path planning in unmanned aerial vehicles [7], [8]. FPGA based Gaussian Mixture Model has been designed for pattern recognition [9] where linear piece-wise approximation

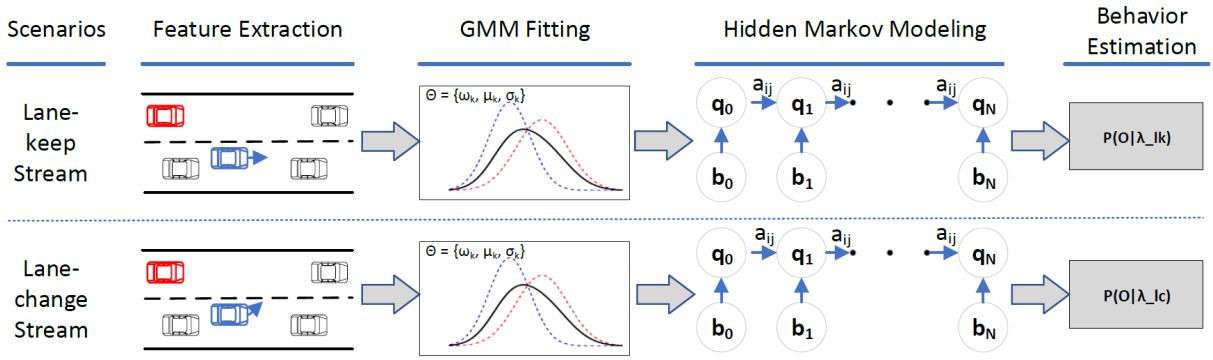

Fig. 2. Hybrid GMM-HMM framework for driving behavior estimation.

is developed for handling exponential calculation. Similarly, FPGA based HMM is designed for speech recognition in [10] and DNA sequence analysis in [11]. GMMs are able to cluster contextual information into different groups, and HMM models the dynamics of contextual information over a time period. However, HMM with Gaussian emissions for autonomous systems has never been attempted before.

In this paper, we propose the *FAuto* framework for acceleration of HMM with Gaussian emissions for driving behavior estimation using FPGAs. *FAuto* can estimate millions of lane change or lane keep intention per second, greatly exceeding traffic contextual data sampling rate. The hardware design of HMM with Gaussian emissions is fully implemented on the embedded system using Vivado High-Level Synthesis (HLS) for higher portability and lower resource utilization. Overall *FAuto* consumes 1.787W, achieving 2.59 teraoperations/second/W (TOPS/W). To the best of our knowledge, *FAuto* is the fastest behavior inference classifier for NGSIM trajectory data. It is an ideal prediction and control platform for autonomous driving. We make following contributions in this paper:

- We holistically introduce the application of hybrid Hidden Markov Model with Gaussian emissions;
- We design a customized FPGA implementation of HMM with Gaussian emissions;
- Flexible batch size of input vectors can be processed with a parameterized hardware design;
- *FAuto* is effectively optimized, and serves as a highly power efficient embedded system for real-time driving behavior estimation;
- *FAuto* is suitable for other applications such as, robotics and speech recognition systems. This design performance can be a FGPA design benchmark in these fields.

The rest of this paper is organized as follows. Section II presents the background of GMM and HMM with Gaussian emissions; Section III describes software implementation, FPGA accelerator design, and experimental setup; Section IV presents the experimental results; and Section V concludes the paper.

II. Background and Approach

This section presents an overview of Gaussian Mixture Model, and describes the approach to driving behavior estimation, i.e., Hidden Markov Model with Gaussian emissions in detail, as no previous work introduced such GMM-HMM framework for applications in a holistic manner.

A. Gaussian Mixture Model

GMMs are probabilistic models used to represent subpopulation distributions within an overall larger population. Data distribution of a Gaussian mixture model with two components would follow the sum of two scaled and shifted normal distributions as indicated in an example Fig. 3. For a Gaussian mixture model of K components, the k^{th} component would have a mean of μ_k and a variance of σ_k. The components can be univariate or multivariate in nature. A multivariate component has a mean $\vec{\mu}_k$ and the variance $\vec{\sigma}_k$. The mixture component weights are defined as ϕ_k for a component C_k, provided that,

$$\sum_{i=1}^{K} \phi_i = 1 \tag{1}$$

This will achieve a normalized probability distribution. For a one-dimensional model, the probability distribution function is given by

$$p(x) = \sum_{i=1}^{K} \phi_i \mathcal{N}(x \mid \mu_i, \sigma_i) \tag{2}$$

where,

$$\mathcal{N}(x \mid \mu_i, \sigma_i) = \frac{1}{\sigma_i \sqrt{2\pi}} exp\left(-\frac{(x-\mu_i)^2}{2\sigma_i^2}\right) \tag{3}$$

For a multi-dimensional model, the probability distribution function is given by

$$p(\vec{x}) = \sum_{i=1}^{K} \phi_i \mathcal{N}(\vec{x} \mid \vec{\mu}_i, \Sigma_i) \tag{4}$$

where,

$$\mathcal{N}(\vec{x} \mid \vec{\mu}_i, \Sigma_i) = \frac{1}{\sqrt{(2\pi)^K |\Sigma_i|}} exp\left(-\frac{1}{2}(\vec{x}-\vec{\mu}_i)^T \Sigma_i^{-1} (\vec{x}-\vec{\mu}_i)\right) \tag{5}$$

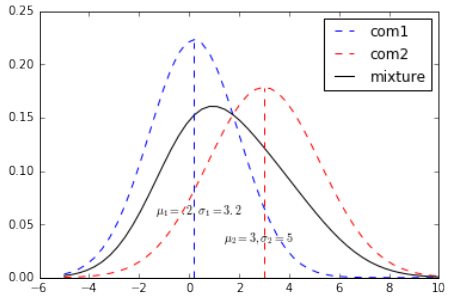

Fig. 3. Probability density function of a GMM in 1D. An simple example of GMM with two components, com1 with weight of 0.4 and com2 with weight of 0.6.

An iterative algorithm called Expectation Maximization (EM) is used to estimate the mixture model's parameters. It is a maximum likelihood estimation technique which starts with an initial estimate of the parameters of the GMM and then iteratively updates these parameters until the model converges. Each iteration constitutes an E-step and an M-step. E-step or the estimation step, involves the calculation of the expectation of the component assignments C_k for each data point $x_i \epsilon X$ when provided with the parameters ϕ_k, μ_k and Σ_k. M-step or the maximization step, is the maximization of the expectation calculated in the E-step. The model parameters ϕ_k, μ_k and σ_k are updated according to the maximum expectation at the end of this step. After every parameter update, the maximum likelihood method is used to find the θ that maximizes the occurrence probability of data sequence $\vec{x}$.

$$L(\theta) = \sum_{t=1}^{M} ln(p(x_t; \theta)) \tag{6}$$

where $\theta = \{\phi_k, \mu_k, \Sigma_k\}$ are the parameters of the GMM. This iterative process repeats until the log-likelihood $L(\theta)$ converges, i.e., the magnitude of difference between its estimates at time t and t-1 measures less than a certain threshold ϵ. In [12], the E-step for a multivariate GMM is given by

$$\hat{\gamma}_{ik} = \frac{\hat{\phi}_k \mathcal{N}(\vec{x} \,|\, \vec{\mu}_i, \Sigma_i)}{\sum_{j=1}^{K} \hat{\phi}_j \mathcal{N}(\vec{x} \,|\, \vec{\mu}_i, \Sigma_i)} \; \forall i, k \tag{7}$$

Where, $\hat{\gamma}_{ik}$ is the probability of the component C_k generating the data point x_i. The M-step, calculates the model parameters by using the $\hat{\gamma}_{ik}$ that is computed in the E-step.

$$\hat{\phi}_k = \sum_{i=1}^{N} \frac{\hat{\gamma}_{ik}}{N}; \quad \hat{\mu}_k = \frac{\sum_{i=1}^{N} \hat{\gamma}_{ik} x_i}{\sum_{i=1}^{N} \hat{\gamma}_{ik}} \tag{8}$$

$$\hat{\Sigma}_k = \frac{\sum_{i=1}^{N} \hat{\gamma}_{ik} (x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T}{\sum_{i=1}^{N} \hat{\gamma}_{ik}} \tag{9}$$

In this work, 11 features of the car following scenario have been observed, and 4 Gaussian components are chosen (referred from [4]) to form the GMM. Each of 4 Gaussian components, which maps all input vectors with these features shown in Table I, is a weighted part of multivariate Gaussian distributions, B. The likelihood of the GMM is maximized by the E-M steps and the final observation probability matrix B is given as input to the HMM as indicated in Fig. 2, to compute the maximum likelihood of the observation sequence and compute the probability of car lane change and lane keep.

TABLE I
FEATURES OF CAR FOLLOWING SCENARIO.

Feature
Longitudinal velocity of veh_t
Lateral velocity of veh_t
Lateral offset from target lane market to veh_t
Longitudinal speed difference between veh_t and veh_p
Longitudinal speed difference between veh_t and veh_h
Longitudinal speed difference between veh_t and veh_ft
Longitudinal speed difference between veh_t and veh_rt
Longitudinal distance between veh_t and veh_p
Longitudinal distance between veh_t and veh_h
Longitudinal distance between veh_t and veh_ft
Longitudinal distance between veh_t and veh_rt

B. Hybrid GMM-HMM Algorithm

A Markov Chain constitutes a sequence of observable states whose probability is dependent only on the current state and not previous states. Given a set of states $Q = \{q_1, q_2, \ldots, q_N\}$, the chain may start in any state q_i and moves to the next state q_j and so on in successive steps. The probability with which the chain steps from one state to another is transition probability a_{ij}. According to [13], the Markov assumption states that only the present matters when predicting the future:

$$P(q_i = a \,|\, q_1 \ldots q_{i-1}) = P(q_i = a \,|\, q_{i-1}) \tag{10}$$

Formally, the components of the Markov Chain are a set Q of N states, a transition probability matrix $A = \{a_{11} a_{12} \ldots a_{nn}\}$ where each element is the transition probability a_{ij} such that $\sum_{j=1}^{n} a_{ij} = 1 \; \forall i$, and, an initial probability distribution $\pi = \{\pi_1 \pi_2 \ldots \pi_N\}$ which is the probability with which the Markov Chain will start in state i, such that $\sum_{j=1}^{n} \pi_i = 1$ [13]. While the markov chain is useful to compute the probability of observable events, there exist some cases where the events are hidden in nature. In these cases, the HMM can be used to consider both types of events to build a probabilistic model. The components of an HMM are the same as those of a Markov Model, augmented with the following components, a sequence T of observations denoted by $O = \{o_1 o_2 \ldots o_T\}$, a sequence $B = b_i(o_t)$ of observation likelihoods or emission probabilities, each expressing the probability of an observation o_t being generated by a state i.

In addition to following the Markov Assumption, the HMM assumes output independence. The probability of the output observation o_i depends only on the state q_i, that produced it.

$$P(o_i \,|\, q_1 \ldots q_i, \ldots, q_T, o_1, \ldots o_i, \ldots, o_T) = P(o_i \,|\, q_i) \tag{11}$$

Assuming the emission probabilities and the transition matrix is provided, the hidden states can be determined by HMM. Given an observation sequence O, its maximum likelihood is determined by using the Forward Algorithm. The best hidden state sequence Q that most likely produces the observation sequence O is then determined by decoding using the Viterbi Algorithm. Further, the HMM can be trained to learn its parameters, given an O and a Q.

Given an HMM $\lambda = (A, B)$ and an observation sequence O, the likelihood is given by $P(O \mid \lambda)$. In an HMM, each hidden state produces a single observation which implies that the length of hidden state sequence Q and O are equal. Therefore,

$$P(O \mid Q) = \prod_{i=1}^{T} P(o_i \mid q_i) \tag{12}$$

The total probability of an observation sequence can be computed by summing over all possible hidden state sequences.

$$P(O) = \sum_{Q} P(O, Q) = \sum_{Q} P(O \mid Q) P(Q) \tag{13}$$

For an HMM with T observations and N states, the total number of possible hidden state sequences would be N^T. In real scenarios, the N and T can be very large numbers, which would make the likelihood computation exponential in nature. To overcome this, an $O(N^2T)$ algorithm called the Forward Algorithm that cumulatively computes the likelihood probability by storing intermediate values [13]. As shown in Fig. 4, the probability of being in state j after generating the first t observations, is denoted as $\alpha_t(j)$. Each $\alpha_t(j)$ is a single cell of the forward trellis, where,

$$\alpha_t(j) = P(o_1, o_2 \ldots o_t, q_t = j \mid \lambda) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t) \tag{14}$$

Where, α_{t-1} is the previous forward path probability from the previous time step. Therefore, the forward algorithm constitutes the following steps:
Initialization:

$$\alpha_1(j) = \pi_j b_j(o_1), \ 1 \leq j \leq N \tag{15}$$

Recursion:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1} a_{ij} b_j(o_t), \ 1 \leq j \leq N, 1 < t \leq T \tag{16}$$

Termination:

$$P(O \mid \lambda) = \sum_{j=1}^{N} \alpha_T(i) \tag{17}$$

Similar to the parameters of the GMM, the parameters of the HMM i.e. the transition matrix A and Gaussian emission matrix B, can be learned. The Forward-Backward algorithm trains the parameters A and B by taking an observation sequence O and a set of hidden states Q as inputs.

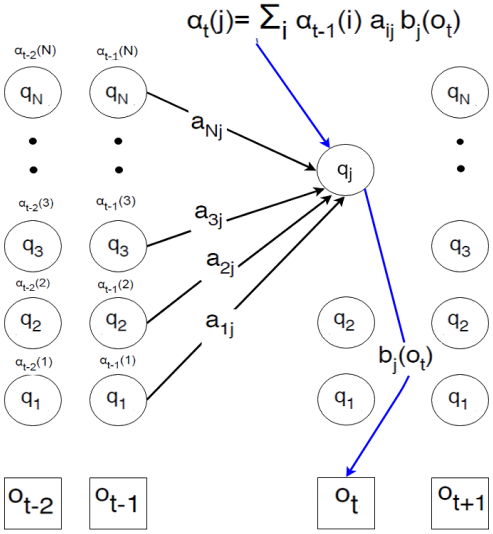

Fig. 4. Visualization of Forward Algorithm path trellis.

The Backward algorithm starts with an estimate of A and B, following which these parameters are updated with better values with each iteration. Given that the state under consideration is i at a time t, the probability of seeing the observations from time $t-1$ to the start is computed in each iteration. This probability is called the Backward probability β.

$$\beta_t(i) = P(o_{t+1}, o_{t+2} \ldots o_T \mid q_t = I, \lambda) \tag{18}$$

The Backward Algorithm has three steps:
Initialization:

$$\beta_T(i) = 1, \ 1 \leq i \leq N \tag{19}$$

Recursion:

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j), \ 1 \leq i \leq N, 1 \leq t < T \tag{20}$$

Termination:

$$P(O \mid \lambda) = \sum_{j=1}^{N} \pi_j b_j(o_1) \beta_1(j) \tag{21}$$

The Baum Welch Algorithm is a special case of EM algorithm, and is used here for finding HMM parameters. It starts with an estimate of parameters, $\lambda = (A, B)$. In the E-step, the expected state occupancy count γ and the expected state transition count ξ is computed from the previous A and B probabilities. New A and B probabilities are computed using the γ and ξ in the M-step. E-step:

$$\gamma_t(j) = \frac{\alpha_t(j) \beta_t(j)}{\alpha_T(q_F)}, \ \forall \, t \, and \, j \tag{22}$$

$$\xi_t(i, j) = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{\alpha_T(q_F)}, \ \forall \, t, \ i, \ and \, j \tag{23}$$

M-step: The re-estimated HMM parameters are given by:

$$\hat{\phi}_j = \frac{1}{T}\sum_{t=1}^{T}\gamma_t(j) \tag{24}$$

$$\hat{\mu}_j = \frac{\sum_{t=1}^{T}\gamma_t(j)x_t}{\sum_{t=1}^{T}\gamma_t(j)} \tag{25}$$

$$\hat{\Sigma}_j = \frac{\sum_{t=1}^{T}\gamma_t(j)(x_t-\hat{\mu}_j)(x-\hat{\mu}_j)^T}{\sum_{t=1}^{T}\gamma_t(j)} \tag{26}$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T}\xi_t(i,j)}{\sum_{k=1}^{N}\sum_{t=1}^{T}\xi_t(i,k)} \tag{27}$$

In this work, we employ separate GMM-HMM hybrid models to compute lane change and lane keep probabilities. Fig. 5 shows a detailed data flow chart of each of these models. The parameters used in the flow chart are defined in Table II. The lane change GMM and HMM are denoted as GMM_{lc} and HMM_{lc} respectively. Similarly, GMM_{lk} and HMM_{lk} are lane keep models. In general, each of the GMM-HMM hybrid models take the 11 features listed in Table I as input. These features have different values based on whether they are considered for lane change or lane keep. Initially, the weights w_{lc}, w_{lk}, and the parameters μ_{lc}, μ_{lk}, Σ_{lc} and Σ_{lk} are assigned to untrained values. The GMM_{lc} and GMM_{lk} compute the Gaussian emissions B_{lc} for the lane change and B_{lk} for the lane keep case respectively, using their parameters. The EM steps train the parameters to maximize the likelihood of the GMMs until the final probabilities converge to a value $< \epsilon$. The trained values are fed back to the GMM and a final B_{lc} and B_{lk} is computed. The B_{lc} is then fed to the HMM_{lc} to compute lane change probability $P(O \mid \lambda_{lk})$. Similarly, the B_{lk} is fed to the HMM_{lk} to compute lane keep probability $P(O \mid \lambda_{lk})$.

The transition matrices A_{lc} and A_{lk} are assigned to initial values and later estimated to values corresponding to maximum likelihood of the observation sequences, by the EM steps in the HMMs. The Forward algorithm in each of the HMMs takes the A_{lc}, B_{lc} and the A_{lk}, B_{lk} matrices as inputs respectively and computes the probabilities of lane change ($P(O \mid \lambda_{lc})$) and lane keep ($P(O \mid \lambda_{lk})$). Finally, the decision whether to change lane or keep lane is made by comparing $P(O \mid \lambda_{lc})$ and $P(O \mid \lambda_{lk})$.

III. Implementation

A. Python Framework Deployment

The labeled NGSIM trajectory dataset from [4] is used for intention estimation classifier training. Fig. 5 shows the data flow chart for GMM-HMM training. 11 features (p = 11) including relative distance and speed differences between target

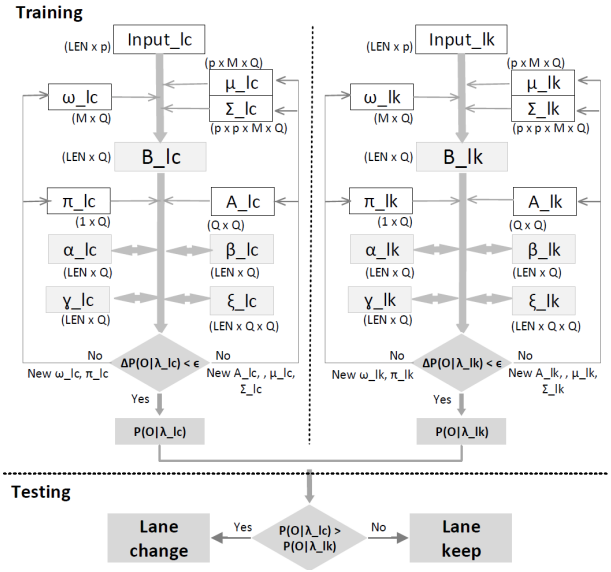

Fig. 5. GMM-HMM data flow chart (lc: lane change; lk:lane keep). The steps and parameters are described in Section 2.2 and Table II.

TABLE II
FAuto Parameters and their Definitions.

Parameter	Lane Change	Lane Keep
Length of input	LEN	LEN
Number of hidden states	Q	Q
Number of Gaussian components	M	M
Number of features	p	p
Input stream	Input_{lc}	Input_{lk}
Weights of GMM	w_{lc}	w_{lk}
Mean	μ_{lc}	μ_{lk}
Covariance	Σ_{lc}	Σ_{lk}
Gaussian Emissions	B_{lc}	B_{lk}
Forward probability	α_{lc}	α_{lk}
Transition matrix	A_{lc}	A_{lk}
Initial probabilities	π_{lc}	π_{lk}
Backward probability	β_{lc}	β_{lk}
Expected state occupancy count	γ_{lc}	γ_{lk}
Expected state transition count	ξ_{lc}	ξ_{lk}
Threshold	ϵ	ϵ
Final probability	$P(O \mid \lambda_{lc})$	$P(O \mid \lambda_{lk})$

vehicle and surrounding vehicles are extracted from separated lane change and lane keeping scenarios. Two separate HMMs with Gaussian emissions are developed for estimating driving behavior and further supporting host vehicle control. The lane keeping framework is of the same structure as lane change in this study. Basically the hybrid model is trained based on the parameter setting described in [4]. Gaussian components of 4 (M = 4) are mixed for modeling each of the three hidden states (Q = 3), which represents the beginning, the end, and the middle movements of target vehicles during certain observation period. Length of input vectors LEN varies from period to period as driving behavior is different in duration of lane keeping or change.

It is worth noting that diagonal covariance matrices of Gaussian components are chosen for modeling each hidden state, as sparsity in the framework helps to reduce system

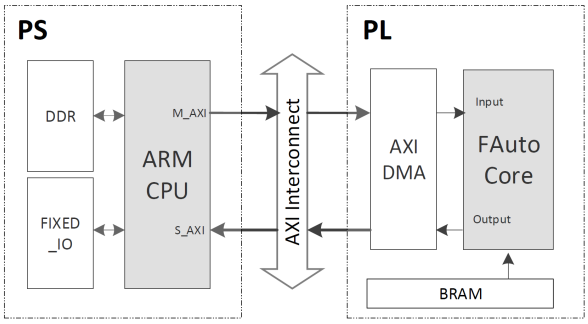

Fig. 6. Acceleration system architecture. Input vectors are transferred to FAuto accelerator and written back to DRAM after computation.

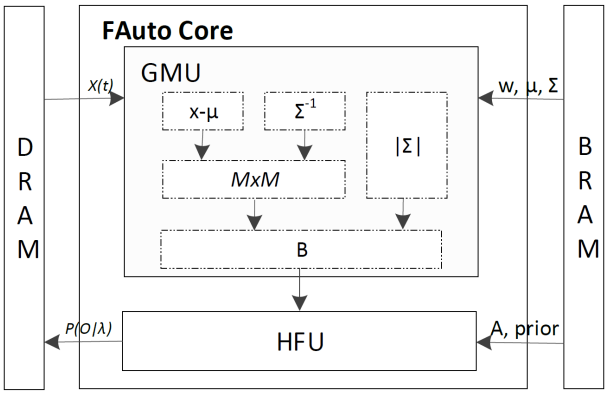

Fig. 7. Block diagram of FAuto accelerator. Weights are stored in BRAM, while every batch of input vectors are fetched from DRAM through Direct Memory Access (DMA).

power consumption. Zero elements in the matrix-vector multiplication results in zero partial sums where multiplication and accumulation are skipped for saving memory access. Initial values of prior vector π is set to 1/3 for each element; and all other parameters (ω, μ, σ, A) are randomly initialized. Termination threshold ϵ is set to $1e^{-7}$ for convergence checking. After HMMs are trained in Python, only the parameters are saved, rather than the whole model, to reduce chip memory.

B. FAuto Accelerator Design

FAuto accelerator is designed using HLS technique that converts C++ description of the architecture into custom logic. It also reduces design and compile time relative to Hardware Description Language such as, Verilog. Optimization techniques such as, for-loop unrolling and pipelining can be efficiently implemented by specifying relative optimization directives. Finally IP is created by exporting RTL in Vivado HLS. Fig. 6 shows the acceleration system architecture. The Programmable Logic (PL) design is implemented on an Xilinx PYNQ-Z2 FPGA chip running at 126 MHz. The Processing System (PS) is the dual-core Cortex-A9 processor. Data transfer between PL and PS is realized by AXI DMA module which maps the data of AXI Stream to memory address, and vice versa.

As shown in Fig. 7, the top-level block diagram of *FAuto* accelerator consists of Gaussian Mixture Unit (**GMU**) and HMM Forward Unit (**HFU**). When running **GMU** input vectors x are subtracted from mean vector μ, and determinant and inverse of covariance matrix are calculated. Given a diagonal covariance matrix Σ, determinant $|\Sigma|$ is the product of non-zero elements in matrix diagonal; and inverse Σ^{-1} is calculated by only getting the reciprocal of non-zero values. Next, matrix multiplication **MxM** between $x - \mu$ and Σ^{-1} can possibly be optimized for concurrency by adopting a *heterogeneous streaming* architecture [14], though it is not adopted in this work. Finally **GMU** performs exponential operation for getting emission probability. As forward probability α is expressed in recurrence relation, **HFU** requires a register file to store previous time α_{t-1}.

C. Design Optimizations

This section explains how GMU and HFU are optimized using HLS design optimization techniques. The system is optimized separately by adding compiler directives (pragmas) with high-level synthesis tools.

Pipelining Speedup: a set of nested loops in GMU are imperfect because Gaussian emission probability is initialized for every hidden state iteration. Thus directives of `PIPELINE` with default Initialization Interval (target II = 1) and full `UNROLL` are performed for achieving highest possible throughput. After synthesis with pipelining and unrolling, the achieved II is 64 for GMU and 18 for HFU. Since input data length is parameterized in final system design, `ARRAY_PARTITION` cannot be applied here for increasing number of block RAM ports.

Data Flow Optimization: data flow pipelining creates a parallel process architecture that allows execution of tasks to overlap, thereby increasing design throughput. In this design, input arrays are set as streaming interface of type `ap_fifo` to reduce FIFO elements to minimum level.

Latency of system with input length of 10 is reduced from initial 216021 clock cycles to 8418 clock cycles. Then input length is parameterized in the design for meeting the demand for variable data length of lane change or keep sections. Data dependencies between sequential functions, as shown in Fig. 7, restrict further pipeline rate improvement of the system. The overall design optimization effect is later shown in Section 4.3.

D. Experimental Setup

Accelerator block design is developed by adding the IP created from HLS into the repository. Then the design is synthesized and implemented for bit stream generation on Vivado 2018.2. Fig. 8 shows the implementation of acceleration system on an Xilinx PYNQ-Z2 board. *FAuto* is designed to accelerate HMM with Gaussian emissions that is the core part of the behavior estimation classifier for enabling further motion planning and control in autonomous driving. As indicated in [15], energy consumption is dominated by memory access more than arithmetic operations. Overlay API manages the data transfer between PL and PS, and also read or write between PL master interface and DRAM. After input vectors

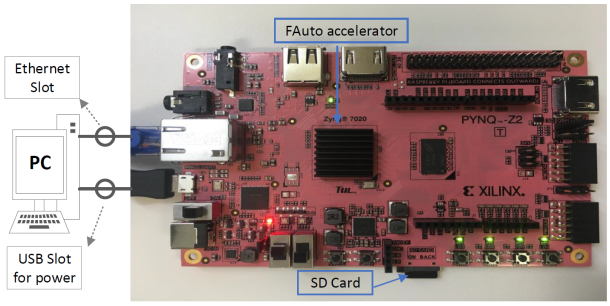

Fig. 8. System-on-chip setup on Xilinx PYNQ-Z2. PYNQ is connected with PC for uploading .tcl and .bit files, powered by connecting to the PC USB slot. An SD Card storing the PYNQ image is inserted.

are calculated by *FAuto* accelerator, result is immediately written back to DRAM.

IV. Experimental Results

This section provides the overview of experimental results in terms of throughput, hardware resource utilization, speedup relative to CPU, accuracy, and power efficiency.

A. Throughput

According to our accelerator design, the number of arithmetic operations required for getting a single behavior estimation likelihood is shown in Table III. Note that the numbers of comparisons are not counted and computational complexity between different arithmetic calculations are not discriminated. The effective throughput is defined as below:

$$\textit{Eff. throughput} = \frac{\textit{fp operations}}{\textit{time}} \tag{28}$$

Assuming a batch of 10 input vectors is streamed to *FAuto*, effective throughput is evaluated by measuring the time required for a single likelihood output. Time used for getting the final likelihood is 7.936 ns, thus effective throughput of the accelerator is 4.63 TOPS.

TABLE III
Operations in the Accelerator.

Ops	mul	add	div	sub	pow	exp	total
#	19023	15933	120	1320	240	120	36756

B. Hardware Resource Utilization

The hardware resource utilization and percentages are reported in Table IV. Since NGSIM data is sampled 10 times per second, input data of dimension 10x11 is transferred to *FAuto* for estimating driving behavior. Thus lower percentage of FPGA resources is utilized. Since the accelerator contains many MAC operations, DSP utilization efficiency is additionally reported according to the method proposed in [16].

As a DSP block is potentially instantiated for floating-point multiplication and accumulation, its utilization efficiency can be evaluated in a similar way of measuring MAC computation

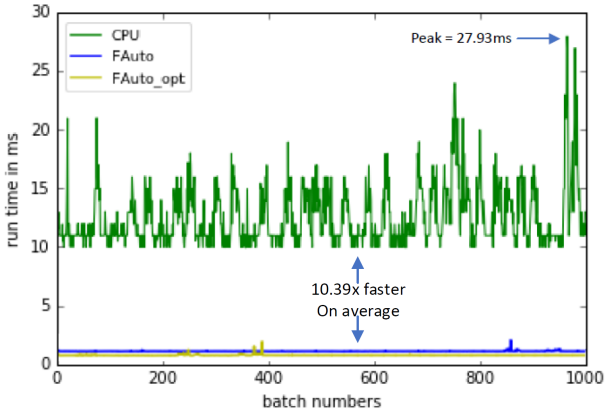

Fig. 9. Runtime for *FAuto*, *FAuto* with optimization and CPU with Python. Mean and variance of CPU runtime are 12.60ms and 6.234ms, respectively. The corresponding values for FAuto are 1.21ms and 0.002ms, respectively.

TABLE IV
FAuto Hardware Utilization.

Resource	BRAM	DSP	FF	LUT
Used	26	111	17039	13713
Available	280	220	106400	53200
Percentage	9.29%	50.45%	16.01%	25.78%

efficiency without considering the factor of memory access bandwidth limit. The formula is defined as follows:

$$\textit{Potential Peak Throughput} = 2 * (\#DSP) * \textit{frequency} \tag{29}$$

$$\textit{DSP Utilization Efficiency} = \frac{\textit{Eff. throughput}}{\textit{Potential Peak Throughput}} \tag{30}$$

C. Computation Speedup

To estimate the computation speedup of *FAuto* accelerator, 10,000 input vectors in 1,000 batches are randomly chosen from NGSIM I-80 dataset for intention estimation inference. Input vectors and all parameters are processed in floating point double precision format both in *FAuto* accelerator and i7-7500U CPU. Fig. 9 shows the runtime in 1000 batches on both platforms. As PYNQ board supports Python language for constructing SoC, we compare the performance of *FAuto* accelerator to CPU platform with Python implementation instead of more efficient languages like C.

Overall *FAuto* accelerator runs 10.39x faster and is more stable than CPU platform in inferencing 1000-batch behavior intentions. Mean and variance of CPU runtime are 12.60ms and 6.234ms, respectively. The corresponding values for *FAuto* are found to be 1.21ms and 0.002ms, respectively. The optimized design denoted as FAuto_opt in Fig. 9 shows another 32.72% decrease in mean runtime (mean = 0.82ms) than *FAuto*, while it maintains the same level of variance (var = 0.004ms).

Besides, it is worth noting that accelerator function is called and input vectors are fetched for each batch, *FAuto* speed calculation is pessimistic due to the extra latency overhead incurred.

D. Prediction Accuracy

In total 200 batches of input vectors of variable length, 100 batches from each datasets due to limited lane change scenarios, are selected for testing driving behavior estimation accuracy of FAuto system. Each batch of testing data is separately feed to GMM-HMM frameworks for lane change and lane keep estimation. The driving behavior is estimated by comparing the forward probabilities generated from two frameworks. Estimation results from accelerator match with that from software implementation on CPU as both use double precision floating point format. The overall prediction accuracy is 89.5%, while the accuracy for lane change and keep is 84% and 95%, respectively. The overall prediction accuracy shows comparable result with [4]. The relative lower accuracy of lane change is attributed to the less number of lane change training sets from NGSIM dataset.

E. Power Efficiency and Overall Comparison

Total on-chip power is 1.787W based on Vivado power analysis from implemented netlist. Since effective throughput is 4.63 TOPS, the power efficiency of the accelerator (i.e., throughput/power) is 2.59 TOPS/W.

In absence of prior GMM-HMM FPGA accelerator performance, the comparison for this work is made with separate GMM and HMM FPGA designs even though they are implemented for different applications. The performance of *FAuto* is summarized in Table V.

TABLE V
FAUTO PERFORMANCE SUMMARY.

Parameter	This work	GMM [17]	HMM [18]
Platform	PYNQ	Virtex-6	Kintex
Frequency (MHz)	126	400	250
Throughput (TOPS)	4.63	0.5	~ 0.71[a]
Power (Watts)	1.79	22	12.4

[a]Sign $\sim$ indicates our estimated value.

V. CONCLUSION

In this paper, we presented a low-power, real-time GMM-HMM accelerator *FAuto* using HLS for driving behavior estimation. The hybrid model is first trained using NGSIM dataset with basically similar parameter settings as [4]. The hardware design is implemented on an Xilinx PYNQ board running at 126 MHz. Input vectors and trained parameters are processed in floating point double precision. Flexible batch size of input vectors can be processed with an LEN-parameterized hardware design. The *FAuto* accelerator achieves an effective throughput of 4.63 TOPS, and DSP utilization efficiency 187.6x. The embedded system consumes estimated on-chip power of 1.787W, and achieves power efficiency of 2.59 TOPS/W. The design performance of this work can be a benchmark for later FPGA embedded autonomous systems. In our future work, hardware system co-design with HLS and Verilog and multiple data precision types will be further studied; and energy consumption will be measured at the block-level for further optimization.

ACKNOWLEDGEMENT

This work is supported by SRC (2847.001) and NSF (CNS-1722557, CCF-1718474, DGE-1723687 and DGE-1821766).

REFERENCES

[1] H. M. Mandalia and D. Salvucci, "Using support vector machines for lane-change detection," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 49, 09 2005.

[2] Y. Hu, W. Zhan, and M. Tomizuka, "Probabilistic prediction of vehicle semantic intention and motion," *CoRR*, vol. abs/1804.03629, 2018. [Online]. Available: http://arxiv.org/abs/1804.03629

[3] T. Streubel and K. H. Hoffmann, "Prediction of driver intended path at intersections," in *2014 IEEE Intelligent Vehicles Symposium Proceedings*, June 2014, pp. 134–139.

[4] Y. Zhang, Q. Lin, J. Wang, S. Verwer, and J. M. Dolan, "Lane-change intention estimation for car-following control in autonomous driving," *IEEE Transactions on Intelligent Vehicles*, vol. 3, no. 3, pp. 276–286, Sep. 2018.

[5] "Introduction to real-time ray tracing with vulkan," *https://devblogs.nvidia.com/vulkan-raytracing/*, 2018.

[6] "Intel vision accelerator design with an intel arria 10 fpga," *https://software.intel.com/en-us/vision-accelerator-design-fpga-user-guide/*, 2018.

[7] A. Moussawi, K. Haddad, and A. Chahine, "An fpga-accelerated design for deep learning pedestrian detection in self-driving vehicles," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1809.05879

[8] F. C. J. Allaire, M. Tarbouchi, G. Labonté, and G. Fusina, "Fpga implementation of genetic algorithm for uav real-time path planning," *Journal of Intelligent and Robotic Systems*, vol. 54, no. 1, pp. 495–510, Mar 2009.

[9] M. Shi, A. Bermak, S. Chandrasekaran, and A. Amira, "An efficient fpga implementation of gaussian mixture models-based classifier using distributed arithmetic," in *2006 13th IEEE International Conference on Electronics, Circuits and Systems*, Dec 2006, pp. 1276–1279.

[10] M. Mosleh, S. Setayeshi, M. M. Lotfinejad, and A. Mirshekari, "Fpga implementation of a linear systolic array for speech recognition based on hmm," in *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, vol. 3, Feb 2010, pp. 75–78.

[11] S. Ren, V. Sima, and Z. Al-Ars, "Fpga acceleration of the pair-hmms forward algorithm for dna sequence analysis," in *2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, Nov 2015, pp. 1465–1470.

[12] D. A. Reynolds and R. C. Rose, "Robust text-independent speaker identification using gaussian mixture speaker models," *IEEE Transactions on Speech and Audio Processing*, vol. 3, no. 1, pp. 72–83, Jan 1995.

[13] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[14] C. J. Newburn, G. Bansal, M. Wood, L. Crivelli, J. Planas, A. Duran, P. Souza, L. Borges, P. Luszczek, S. Tomov, J. Dongarra, H. Anzt, M. Gates, A. Haidar, Y. Jia, K. Kabir, I. Yamazaki, and J. Labarta, "Heterogeneous streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 611–620.

[15] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 10–14.

[16] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "Deltarnn: A power-efficient recurrent neural network accelerator," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174261

[17] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," *SIGPLAN Not.*, vol. 50, no. 4, pp. 223–238, Mar. 2015.

[18] M. Ito and M. Ohara, "A power-efficient fpga accelerator: Systolic array with cache-coherent interface for pair-hmm algorithm," in *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*, April 2016, pp. 1–3.