GAME: Gaussian Mixture Model Mapping and Navigation Engine on Embedded FPGA

Yuanfan Xu*‡, Zhaoliang Zhang*‡, Jincheng Yu*‡, Jianfei Cao†, Haolin Dong*‡, Zhengfeng Huang†, Yu Wang*‡ and Huazhong Yang*‡
*Department of Electronic Engineering, Tsinghua University, Beijing, China
†School of Microelectronics, Hefei University of Technology, Hefei, China
‡Beijing National Research Center for Information Science and Technology (BNRist), Beijing, China
Email: {xuyf20, zhaolian20, yjc16}@mails.tsinghua.edu.cn, jianfei-cao@mail.hfut.edu.cn, donghl17@mails.tsinghua.edu.cn
huangzhengfeng@hfut.edu.cn, {yu-wang, yanghz}@tsinghua.edu.cn

***Abstract*—3D mapping is a fundamental task in robot applications. The traditional mapping methods mainly rely on spatial discretization, in which the amount of data that needs to be stored is large, and the representation ability is limited. As a continuous probability model, the Gaussian Mixture Model (GMM) has a small memory footprint and high-fidelity representation ability. Thus the GMM map is superior to discrete map representations in basic robot tasks such as navigation and localization. The general method of building GMM maps is the iterative Expectation-Maximization (EM) algorithm with K-means initialization. The EM and K-means algorithms are computation-intensive, making it challenging to meet real-time 30 fps mapping requirements on the embedded robot systems. This paper proposes a Gaussian mixture model mapping and navigation engine (GAME) on embedded FPGA to accelerate the mapping process. To achieve fully pipelined with minimal hardware resource cost, we design a unified dataflow and hardware architecture for both K-means and EM for GMM. We analyze different quantization strategies for higher parallelism and find a low-bit quantization method with mixed 8/16-bit data representation, bringing negligible loss in accuracy. Combining the unified dataflow and the mixed-bit data quantization, GAME enables real-time GMM mapping and navigation on embedded robots. The experimental results on ZCU102 show that our proposed hardware-software co-optimization framework on FPGA can run over 60× faster than on a GeForce 1080Ti GPU and over 490× faster than on an Nvidia Jetson TX2, and achieves 59 fps.**

I. Introduction

3D Map building is a fundamental task in autonomous robot applications such as rescue, navigation, and exploration. Occupancy grid maps [1][2] are widely used for map representation throughout the mobile robotics community. However, these discretized occupancy grid methods suffer from the huge storage and communication consumption, especially when the communication bandwidth is limited. For example, the data rate of the Mars-to-Earth communications is constrained to about 10KB/s [3]. The occupancy grid map consumes 1MB/s to model a typical Mars scene [4], which precludes the transfer of perceptual models and human operation between Mars and Earth. Gaussian Mixture Models (GMMs) are used for mapping under communication constrained and storage limited environments. GMM map reduces the data volume more than 200× than the occupancy grid maps (from 1MB/s to 5KB/s) and meets the Earth-to-Mars communication bound [4]. The data efficiency of GMM map makes it possible to perform Earth-to-Mars interaction. Besides the low communication and storage demand, the accuracy of GMM map is also better in some particular environments such as tunnels and mines [5].

However, GMM mapping is computation-intensive [6], making it challenging to achieve real-time mapping and navigation on embedded robots. For example, we implement the GMM mapping on the embedded Nvidia Jetson TX2 and use an RGB-D camera to obtain a filtered point cloud with about 90K points. The mapping speed on TX2 is 0.12 fps, far from meeting the 30 fps real-time requirement, which is a typical frame rate of an RGB-D camera.

GMM uses the iterative Expectation-Maximization (EM) algorithm [7] to maximize the data likelihood of the input point cloud. When the number of points increases, the single iteration time and the number of iterations increase. Since GMM provides a compressed map representation, the path planning requires little computing and can be deployed on the embedded CPU. Thus, the GMM mapping naturally becomes the bottleneck of the autonomous navigation system. Based on the fact that the EM algorithm for GMM is computation-constraint [6], the traditional GPU accelerators speed up EM by increasing parallelism while using more GPU cores. However, the embedded GPU, such as TX2 with only 256 cores, cannot achieve the real-time performance required for mapping due to the limited number of computing units.

Using low-bit data representation and fixed-point arithmetic is a general method to reduce computing resource usage and improve parallelism. However, due to the lack of hardware flexibility, GPU cannot fully support fixed-point operations. On the other hand, the low-bit data representation fully leverages the flexibility on FPGA and improves the hardware utilization and parallelism. For example, the FPGA resource consumption for multiplier and adder with 32-bit fixed-point data is about 15 times more than that with 8-bit fixed-point data [8]. For GMM mapping, we adopt a mixed-bit fixed-point data representation with 8-bit/16-bit. The accuracy of the GMM map with the mixed-bit number drops within 0.06%, which meets navigation requirements. Compared with the floating-point map, the mixed-bit one can still retain the critical objects in the scene for perception, and describe the occupancy

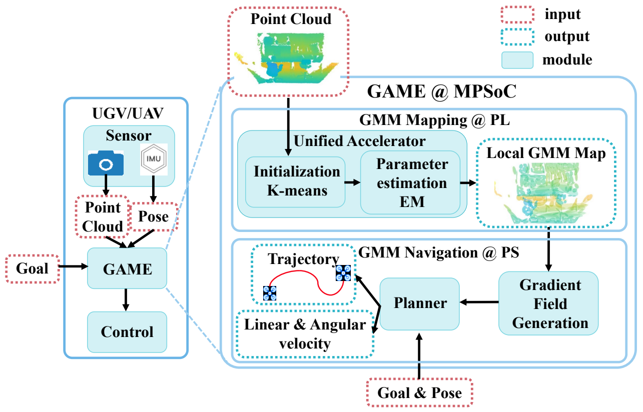


Fig. 1. The overview of our embedded robot system with GAME. The left part is an UGV (Unmanned Ground Vehicle) or UAV (Unmanned Aerial Vehicle) with three modules. GAME receives the pointcloud and run the GMM mapping on FPGA. The process of GMM mapping contains K-means initialization and EM for parameter estimation and finally outputs a local GMM map. **We design a unified and mixed-bit fixed-point accelerator for both algorithms**. Then the constructed map is fed to the navigation module on CPU. The GMM Navigation firstly generates a gradient field based on the probabilistic map. A planner completes path planning according to the gradient field and the goal.

for decision-making and navigation.

Besides, GMM mapping's performance is strongly related to the initialization method of the iterative EM algorithm [9][10]. Random and clustering initialization are two primary initialization methods, which have different application scenarios. We find that for GMM mapping, clustering instead of random initialization can significantly reduce the number of EM iterations and improve GMM maps' accuracy. Nevertheless, this clustering process also needs speedup. Fortunately, some widely-used clustering initialization methods, such as K-means [11], have a similar processing flow with EM. Thus, we can design a unified fully-pipeline algorithm and hardware architecture instead of two independent IPs to accelerate K-means and EM simultaneously with minimal hardware resource consumption.

This paper designs a GAussian mixture model Mapping and navigation Engine (GAME) on FPGA, which builds GMM maps from high-density point clouds in real-time, and further processes path planning based on the GMM map. The whole embedded robot system with GAME is illustrated in Figure 1.

We summarize our contributions as follow:

- We analyze the influence of different data representations on GMM mapping and propose a mixed-bit fixed-point data representation strategy to improve the parallelism with limited hardware resources.
- We propose a unified data flow and hardware architecture for both iterative EM algorithm and K-means initialization of GMM map building, thereby saving half of the hardware resources.
- We deploy GAME in a real FPGA-based robot platform and achieve a real-time GMM mapping and navigation system, running at the speed of 59 fps.

The rest of this paper is organized as follows. Section II introduces the principle of GMM mapping and navigation. Section III details the data flow optimization and data representation analysis. Section IV designs the unified hardware architecture. The results and evaluation are listed in Section V. Section VI introduces the related works. Section VII concludes this paper.

II. GMM-based 3D Mapping and Navigation

A. GMM Map

GMM mapping assumes that an observed point cloud $\mathcal{Z}$ of size N is a sample from an underlying continuous distribution and models it with a weighted sum of multiple Gaussian distributions. A GMM map, $\mathcal{G}$, with K Gaussian components can be represented in the form

$$\mathcal{G}(\mathcal{Z}) = \prod_{n=1}^{N} p(\boldsymbol{z_n}|\boldsymbol{\mu},\boldsymbol{\Sigma}) = \prod_{n=1}^{N}\sum_{k=1}^{K}\pi_k\mathcal{N}(\boldsymbol{z_n}|\boldsymbol{\mu_k},\boldsymbol{\Sigma_k}) \quad (1)$$

where π_k is the importance of weight and $\mathcal{N}$ is a Gaussian probability density function (PDF) controlled by the mean $\boldsymbol{\mu_k} \in R^3$ and the covariance matrix $\boldsymbol{\Sigma_k} \in R^{3\times3}$. The $p(\boldsymbol{z_n}|\boldsymbol{\mu},\boldsymbol{\Sigma})$ describes the spatial density of a point $\boldsymbol{z_n} \in \mathcal{Z}$.

Because the GMM map only uses Gaussian models' parameters to represent the scene, it achieves a small memory footprint and is communication-efficient. However, constructing GMM maps is time-consuming, which can be divided into two steps, 1) *parameter initialization* and 2) *EM for parameter estimation*. There are two main methods for initializing GMMs, the stochastic initialization methods such as *emEM* [10] and *RndEM* [12] and the clustering initialization method such as K-means [11]. In most robot mapping tasks, the scene's objects are well-separated, where K-means should be preferred [9]. The EM algorithm is a general way to estimate optimal GMM parameters [13]. The expectation step (E step) introduces latent variables for each point z_n and each Gaussian component k, and computes the posterior probability r_{nk} using the current estimation of parameter estimates.

$$r_{nk} = \frac{\pi_k\mathcal{N}(\boldsymbol{z_n}|\boldsymbol{\mu_k},\boldsymbol{\Sigma_k})}{\sum_{j=1}^{K}\pi_j\mathcal{N}(\boldsymbol{z_n}|\boldsymbol{\mu_j},\boldsymbol{\Sigma_j})} \quad (2)$$

where r_{nk} denotes the responsibility that the kth Gaussian component takes for point $\boldsymbol{z_n}$.

The maximization step (M step) updates $\{\pi_k^+,\boldsymbol{\mu_k^+},\boldsymbol{\Sigma_k^+}\}$ using r_{nk} computed in the E step by maximizing the expected log-likelihood as shown in Equation 3-5. The detailed original EM algorithm for GMM can be referred in [13].

$$\boldsymbol{\mu_k^+} = \sum_{n=1}^{N}\frac{r_{nk}\boldsymbol{z_n}}{\sum_{n=1}^{N} r_{nk}} \quad (3)$$

$$\boldsymbol{\Sigma_k^+} = \sum_{n=1}^{N}\frac{r_{nk}(\boldsymbol{z_n}-\boldsymbol{\mu_k^+})(\boldsymbol{z_n}-\boldsymbol{\mu_k^+})^T}{\sum_{n=1}^{N} r_{nk}} \quad (4)$$

$$\pi_k^+ = \frac{\sum_{n=1}^{N} r_{nk}}{N} \quad (5)$$

The covariance matrix $\boldsymbol{\Sigma}$ can be full rank or constrained to be diagonal. On reconfigurable platforms, the diagonal matrix GMMs are more computationally efficient because the matrix inversion is not required, which can be replaced by reciprocal. In this study, we assume that all maps are diagonal-matrix GMMs as a variety of studies about GMMs do [6][14][15]. Thus the covariance matrix should satisfy

$$\boldsymbol{\Sigma_k} = diag(\boldsymbol{\sigma_k^2}) \tag{6}$$

where $\boldsymbol{\sigma_k^2} = (\sigma_{k1}^2, \sigma_{k2}^2, \sigma_{k3}^2)$ is a vector of variance value from xyz-axis respectively. In this form, the Gaussian PDF $\mathcal{N}$ can be decomposed into three independent one-dimensional standard Gaussian distribution functions.

$$\begin{aligned}\mathcal{N}(\boldsymbol{z_n}|\boldsymbol{\mu_k}, \boldsymbol{\Sigma_k}) &= \prod_{d=1}^{3} \mathcal{N}(z_{nd}|\mu_{kd}, \sigma_{kd}^2) \\ &= \prod_{d=1}^{3} \frac{1}{\sigma_{kd}} \prod_{d=1}^{3} \phi(\frac{z_{nd}-\mu_{kd}}{\sigma_{kd}})\end{aligned} \tag{7}$$

where $\phi(u)$ is the standard Gaussian distribution function:

$$\phi(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2} \tag{8}$$

B. GMM-based Navigation

Given a target, navigation is the process of planning a collision-free path for the robot. While moving along the path, the robot carries out real-time trajectory replanning based on map information, which promises safety for each movement.

In this study, we improve the perception-aware trajectory replanning algorithm proposed in [16]. Instead of ESDF map [17], we use the gradient field of the probability distribution described by the GMM map to evaluate the distance from obstacles. At every moment, our robot calculates the gradient value of its current position for trajectory replanning, which guarantees a safe solution for navigation. We manually set the navigation goal in the camera's observation range to prevent a complex path-planning solution. In this way, we can evaluate the GMM map merely according to the replanning trajectory. The computation of path planning requires little computing and can be deployed on the embedded CPU.

III. Algorithm Optimization for GMM Mapping

In this section, we introduce the extended pipeline-friendly EM algorithm and K-means for GMM mapping and how we combine the two iterative algorithms into one unified dataflow. Then we analyze the dynamic range of different types of data in the GMM mapping process, and adopt a mixed-bit quantization strategy to find the optimal bit width and precision for each data.

A. A Unified Pipelined Algorithm for GMM mapping

The original EM algorithm [13] cannot be pipelined at point level because every step in its M step needs the input of all points (Equation 3-5). Guo *et al.* [18] introduce several intermediate variables and reconstruct the *E_step* and *M_step* to avoid repetitive all points reading and make EM pipeline-friendly, which is shown in Algorithm 1. However, they use random initialization, resulting in poor GMM mapping performance, which should be replaced by K-means initialization.

Algorithm 1 Pipeline-friendly EM for GMM Mapping

```
 1: Init: $\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma} \leftarrow$ random_init($\mathcal{Z}$)
 2: while stop condition not met do
 3:     $\rho \leftarrow \{0\}_{K\times3}, \tau \leftarrow \{0\}_{K\times3}, \eta \leftarrow \{0\}_K$
 4:     for $n \leftarrow 1$ to $N$ do
 5:         $s \leftarrow 0, g \leftarrow \{0\}_K, r \leftarrow \{0\}_K$
 6:         for $k \leftarrow 1$ to $K$ do
 7:             $g_k \leftarrow \pi_k \mathcal{N}(\boldsymbol{z_n}|\boldsymbol{\mu_k}, \boldsymbol{\Sigma_k})$
 8:             $s \leftarrow s + g_k$
 9:         for $k \leftarrow 1$ to $K$ do
10:             $r_{nk} \leftarrow \frac{g_k}{s}$
11:             $\eta_k \leftarrow \eta_k + r_{nk}$
12:             for $d \leftarrow 1$ to $3$ do
13:                 $\rho_{kd} \leftarrow \rho_{kd} + r_{nk} z_{nd}$
14:                 $\tau_{kd} \leftarrow \tau_{kd} + r_{nk} z_{nd}^2$
15:     for $k \leftarrow 1$ to $K$ do
16:         $\pi_k \leftarrow \frac{\eta_k}{N}$
17:         for $d \leftarrow 1$ to $3$ do
18:             $\mu_{kd} \leftarrow \frac{\rho_{kd}}{\eta_k}$
19:             $\sigma_{kd}^2 \leftarrow \frac{\tau_{kd}\eta_k - \rho_{kd}^2}{\eta_k^2}$
20:         $\Sigma_k \leftarrow diag(\sigma_k^2)$
```

Algorithm 2 Pipeline-friendly K-means for GMM Initialization

```
 1: Init: $\boldsymbol{\mu} \leftarrow$ sample($\mathcal{Z}, K$),$\boldsymbol{\pi}, \boldsymbol{\Sigma} \leftarrow \{1\}_K, \{1\}_{K\times3}$
 2: while stop condition not met do
 3:     for $n \leftarrow 1$ to $N$ do
 4:         for $k \leftarrow 1$ to $K$ do
 5:             $g_k \leftarrow ||\boldsymbol{z_n} - \boldsymbol{\mu_k}||$
 6:         $\hat{k} \leftarrow \arg\min_k g_k$
 7:         $\eta_{\hat{k}} \leftarrow \eta_{\hat{k}} + 1$
 8:         for $d \leftarrow 1$ to $3$ do
 9:             $\rho_{\hat{k}d} \leftarrow \rho_{\hat{k}d} + z_{nd}$
10:             $\tau_{\hat{k}d} \leftarrow \tau_{\hat{k}d} + z_{nd}^2$
11:     for $k \leftarrow 1$ to $K$ do
12:         $\pi_k \leftarrow \frac{\eta_k}{N}$
13:         for $d \leftarrow 1$ to $3$ do
14:             $\mu_{kd} \leftarrow \frac{\rho_{kd}}{\eta_k}$
15:             $\sigma_{kd}^2 \leftarrow \frac{\tau_{kd}\eta_k - \rho_{kd}^2}{\eta_k^2}$
16:         $\Sigma_k \leftarrow diag(\sigma_k^2)$
```

We extend this idea to the K-means algorithm in Algorithm 2. Compared with the original K-means algorithm, we make the following modifications. Firstly, this is a pipeline-friendly algorithm that has a similar dataflow to Algorithm 1. Secondly, the original K-means only needs to update the 3D coordinates of the cluster center, corresponding to the means $\boldsymbol{\mu}$ of GMMs. We add Line 10-16 to maintain the weights and covariances of different clusters for efficient GMM initialization.

Given the similarity in the two algorithms and dataflows,

Algorithm 3 GMM Mapping in GAME

```
Input: pointcloud $\mathcal{Z}$; model number $K$
Output: GMM parameters $\{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\}$
 1: $\boldsymbol{\mu} \leftarrow$ sample($\mathcal{Z}, K$)
 2: $\{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\}_{init} \leftarrow Pipelined\_KmEM(\mathcal{Z}, K, \mathbf{1}, \boldsymbol{\mu}, \boldsymbol{I}$,"Km")
 3: $\{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\} \leftarrow Pipelined\_KmEM(\mathcal{Z}, K, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}$,"EM")
 4: function PIPELINED_KMEM($\mathcal{Z}, K, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, ALG$)
 5:     while stop condition not met do
 6:         $\eta \leftarrow \{0\}_K, \rho \leftarrow \{0\}_{K\times3}, \tau \leftarrow \{0\}_{K\times3}$
 7:         for $n \leftarrow 1$ to $N$ do
 8:             $g \leftarrow \{0\}_K, r \leftarrow \{0\}_K$
 9:             $g \leftarrow$ CALPROB($\boldsymbol{z_n}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}$)
10:             $r \leftarrow$ ALLOCATERESP($g, K, ALG$)
11:             $\eta, \rho, \tau \leftarrow$ COLLECTSTAT($\boldsymbol{z_n}, r$)
12:         for $k \leftarrow 1$ to $K$ do
13:             $\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma} \leftarrow$ UPDATEPARAM($\eta, \rho, \tau$)
    return $\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}$
```

we do not need to waste valuable logic resources to deploy two independent modules for K-means and EM. We design a unified dataflow and introduce a switch signal ALG to distinguish the algorithms in only one step, as shown in Algorithm 3, which is mapped to a single hardware IP in Section IV.

The entire GMM mapping process can be divided into three steps: 1) sample the initial cluster centers at equal intervals from the input point cloud (Line 1); 2) use *Pipelined_KmEM* to initialize the parameters of GMM (Line 2); 3) employ *Pipelined_KmEM* to estimate the optimal parameters iteratively (Line 3). *Pipelined_KmEM* (Line 4-13) is the core function that unifies the dataflows of EM and K-means, which consists of four sub-functions:

CalProb calculates K probabilities that the point belongs to K models respectively given the input point $\boldsymbol{z_n}$ and model parameters. For K-means initialization, we use Gaussian distribution probability to replace Euclidean distance. As long as we ensure that the weights of each Gaussian model and the diagonal elements of all covariance matrices are equal, the monotonicities of the probability function and the Euclidean distance are completely consistent. As K-means only uses the maximum value, this algorithmic transformation is lossless. *CalProb* corresponds to Line 6 to 8 in Algorithm 1 and Line 4 to 5 in Algorithm 2.

AllocateResp calculates K responsibilities that K Gaussian models take for the point. *AllocateResp* corresponds to Line 8 and Line 10 in Algorithm 1 and Line 6 in Algorithm 2. There are some differences between the two algorithms that cannot be unified, so we use an extra switch signal to choose which operation will be done. For EM, it firstly sums up K probabilities g_k and then assign responsibility according to the proportion. For K-means, it compares the K probabilities g_k, and then assign 1 to the maximum value and 0 to the rest.

CollectStat collects statistical information required to update parameters, corresponding to Line 11 to 14 in Algorithm 1 and Line 7 to 10 in Algorithm 2.

UpdateParam uses statistical evidence to update model parameters, corresponding to Line 15 to 20 in Algorithm 1 and Line 11 to 16 in Algorithm 2.

By deploying Algorithm 3 on FPGA, we can use a single IP to complete the GMM mapping that originally needs two IPs, thus reducing the hardware resource consumption greatly.

B. Fixed-point GMM Mapping

Data quantization has been widely used to reduce memory footprint and computation resources [19][20]. We adopt the data quantization flow to GMM mapping and find an optimal mixed-bit data quantization strategy. For a fixed-point number n, its value can be expressed as:

$$n = \sum_{i=0}^{bw-1} B_i \cdot 2^{-f_l} \cdot 2^i \tag{9}$$

where bw is the bit width and f_l is the fractional length which can be negative. B_i is the digit on the i^{th} bit, which is 0 or 1. We aim to find the optimal fractional length for the corresponding data:

$$f_l = \arg\min_{f_l} \sum |D_{float} - D(bw, f_l)| \tag{10}$$

where D is the input data (z_n, π, μ, Σ) or the intermediate data $(g, r, s, \eta, \rho, \tau)$, and $D(bw, f_l)$ is the fixed-point representation under given bw and f_l. We analyze the dynamic range of the data, and then find the optimal f_l.

1) Input Data: The input data is mainly fed to *CalProb* and *CollectStat*. We use Equation (7) to obtain the probability, which needs to firstly do standardized transformation ($u = \frac{z_{nd}-\mu_{kd}}{\sigma_{kd}}$), then get three results of standard Gaussian function ($\phi(u)$), and calculate the product of seven multipliers (3 results of standard Gaussian function, 3 reciprocals of σ and 1 weight π). We find that all the operations only use the reciprocals of the square root of diagonal elements $\frac{\mathbf{1}}{\boldsymbol{\sigma_k}} = \{\frac{1}{\sigma_{k1}}, \frac{1}{\sigma_{k2}}, \frac{1}{\sigma_{k3}}\}$. We represent $\frac{\mathbf{1}}{\boldsymbol{\sigma}}$ as *RStd*. And we actually store and use *RStd* instead of $\boldsymbol{\Sigma_k} = diag(\boldsymbol{\sigma_k^2})$ in the whole mapping process to avoid extra division and simplify the calculation. $\boldsymbol{z_n}$ and $\boldsymbol{\mu}$ are strongly related to the point cloud coordinates. The bw and fl of $\boldsymbol{z_n}$ might change the original distribution of point clouds. But we think that the sensing range of the robot sensor is limited, so the dynamic range of the coordinates will not be very large. We will use 8 bits to represent them, which is verified to be reasonable in Section V-C. For GMM kth component, π_k represents the number of points and $\frac{1}{\sigma_k}$ represents the spreading range of points in it. Most real scenes are regular, so we believe 8-bit width is also enough for them. Once the bw is given, we can find out an optimal f_l according to Equation (10). As intermediate results of calculation, the input (u) and the output of Gaussian PDF are both 16-bit width.

2) Intermediate Data for Each Point: g, s, r are re-initialized to operate each input point. g, s are used in the division operation in Algorithm 1 Line 10. r is used in the multiplication operation in Algorithm 1 Line 13 & 14. These division and multiplication are resource costing. Because g is obtained by multiplying several numbers, its original bit width is over 72 bits, resulting in the huge area of the corresponding divider and needs to be shortened. However, **g measures the relative probability of a single point on different Gaussian models. It can be huge (more than 10^4) or small (less than 10^{-4}). Since g is the output of E_step, its high precision is the key to effective iteration.** Therefore, we truncate g to 16 bits rather than 8 bits for subsequent operations. The mapping performance comparison between 8-bit and 16-bit for g will be given in Section V-C. Since s is accumulated by g, in order to ensure no overflow, we use 24-bit width for s. We present the quotient of g and s, which is r, with 8-bit width.

3) Intermediate Data across Points : η, ρ, τ are accumulated across different points in Algorithm 1, Line 11-14. Because they are only used for accumulation, compared with multiplication and division, the accumulation consumes few resources. To guarantee that the cumulative results do not overflow, we use 32-bit numbers to represent them.

IV. Hardware for GMM Mapping

A. Overall Architecture

The hardware is implemented on Xilinx MPSoC, with ARM cores (Processing System, PS) and FPGA fabric (Programmable Logic, PL) integrated on the same chip.

As mentioned in Section III, due to the similarity between K-means and EM, we use one single hardware IP with a control signal to determine which algorithm to apply. The overall architecture is shown in Figure 2. The hardware IP contains four main modules managed in pipeline: 1) *GatData*, 2) *CalProb*, 3) *AllocateResp* and 4) *CollectStat*, which correspond to the steps between Line 9 and Line 11 in Algorithm 3. We adopt FIFOs to connect these modules and cache the input/output data from/to DDR. We implement the *CalProb*, *AllocateResp*, and *CollectStat* on PL because they apply repetitive operations for each input point and the number of points is large. The *UpdateParam* module is implemented on PS because it only works once in an iteration and contains many division operations. Deploying this module on FPGA will wastes logic resources and cannot obtain equivalent returns. The PS controls the PL through the AXI interface, and PS and PL use the same DDR to share the parameters of GMMs. PS determines how many iterations to perform by calling PL.

Because the Gaussian model amount is limited (128 models in our implementation), the data volume of model parameters is much smaller than these of point cloud data. Thus, we store the model parameters in on-chip BRAM and pipeline the computation at point level.

B. GetData & UpdateParam

The *GetData* module is designed to read data from DDR and push it to FIFOs. Every time it gets a set of 3D coordinates of one point, the module puts the data to two streams: one to the *CalProb* module and the other to the *CollectStat* module.

After one iteration, *UpdateParam* module receives the output of PL and use it to update the parameters of Gaussian models, corresponding to Algorithm 3 Line 13.

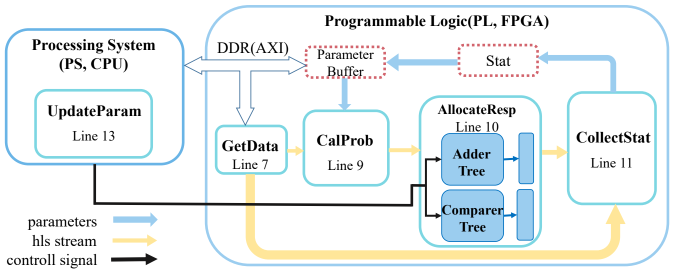


Fig. 2. The overview of hardware architecture. The "Line X" in each module corresponds to Line X in Algorithm 3.

C. CalProb

The *CalProb* module takes a point data from the data stream and calculates the probabilities of the point on all Gaussian models. In order to reduce the module's latency, we expand the computation on the GMM model number (K-dimension). Each Gaussian model is assigned with a processing element by unrolling the loop in Algorithm 1 Line 6. In the final implementation, the number of GMM models is 128 (K=128), and each GMM model corresponds to 16 multiplications and 3 additions. For multiplication, there are 15 multiplications to calculate GMM probability function $\mathcal{N}(\boldsymbol{z_n}|\boldsymbol{\mu_k}, \boldsymbol{\Sigma_k})$ and 1 multiplication with the weight (π_k). For addition, *CalProb* needs to subtract the corresponding mean value from the input data ($\boldsymbol{z_n} - \boldsymbol{\mu_k}$) at 3 dimensions. The parallelism of the this module is 128 $\times$ (16 + 3) = 2432.

D. AllocateResp

This module contains two different dataflows, corresponding to different hardware designs. The module receives the control signal to determine which one to use.

The adder tree is used if the IP is performing EM algorithm. It computes the sum of the probabilities that current data point on all the Gaussian models. After that, a divider divides all the probabilities by the sum, corresponding to Line 10 in Algorithm 1.

The comparer tree is used if K-means is performed. It determines the maximal probability, and set it to 1 while the others to 0, corresponding to Line 6 in Algorithm 2.

Each GMM model processes 1 addition and 1 division (replaced by multiplying the reciprocal of s) in this step, corresponding to Line 8 and Line 10 in Algorithm 1. The parallelism of this module is 128 $\times$ 2 = 256.

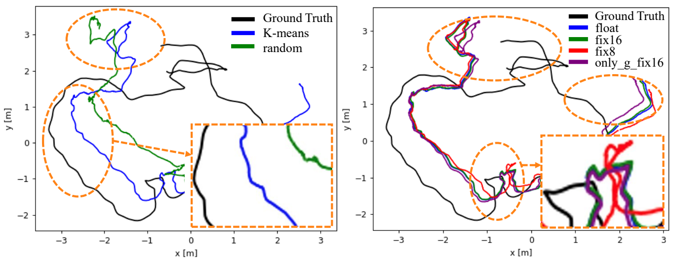
E. CollectStat

This module receives the responsibilities and corresponding data point to calculate the statistics, which will be used for *UpdateParam*. It receives the data streams from the *GetData* module and *AllocateResp* module. We also expand the computation on the GMM model dimension.

Each GMM model processes 9 multiplications and 8 additions in this step, corresponding to Algorithm 1 Line 11-14. The parallelism of this module is 128 $\times$ (9 + 8) = 2176.

(a) Stanford Bunny (b) TUM dataset (c) Gazebo simulation

Fig. 3. The three datasets we use to evaluate GMM mapping.



(a) different initialization methods (b) different quantization strategies

Fig. 4. (a) Top-down view of trajectories generated using different initialization methods for the TUM dataset. The blue trajectory (K-means initialization) is closer to the black trajectory (Ground Truth) than the green trajectory (random initialization). (b) Top-down view of trajectories generated using different quantization strategies for the TUM dataset. The purple trajectory (only_g_fix16 as Exp5 in Table I) matches the ground truth trajectory best and the red trajectory (fix8 as Exp3 in Table I) has the worst performance.

F. *Parallelism Analysis*

Because all modules run in parallel (each cycle can process one point), our hardware's overall parallelism is the sum of the parallelism of each submodule on PL. Thus the total parallelism is 4864, which is much higher than that of embedded GPU TX2 (256) or even higher than desktop GPU 1080Ti (3584). All the parallelism above refers to the number of computations the hardware can perform in each clock cycle. GPU cannot support fine-grained pipeline optimization, which leads to far lower hardware utilization rate than FPGA, further reducing the performance on GPU.

V. Experiment

A. *Experiment Setup*

1) Data Set: We use the results of map registration and the trajectory length of navigation to evaluate the performance of GMM mapping as Keselman *et al.* do in [21]. For registration, we replicate the experimental setup of [22] on Stanford Bunny and perform experiments on an RGB-D dataset sequence from the TUM dataset (*freiburg3 long office household*) [23]. We use the same GMM distribution-to-distribution (D2D) registration method from [5]. For the path length, we develop a simulation environment based on Gazebo [24] as a test scene for navigation. We finally test the time costing of GAME with the point cloud filtered from the Intel RealSence D435 RGB-D camera [25]. The datasets that we use are shown in Figure 3.

2) Computation Platform: Our whole design is described in C and synthesized into Verilog RTL using Xilinx Vitis HLS toolkit version 2020.2. We deploy our implementation in a Xilinx ZCU102 running at 250 MHz. For the GPU implementation, we use open-source codes with best performance described in [26] and [27] to accelerate K-means and EM respectively, achieving almost the same results in previous work [6][14]. The target GPU computing platforms are an GeForce GTX 1080Ti and an embedded Jetson TX2 board.

B. *The Influence of Different Initialization Methods*

There are two major initialization methods for EM algorithm, random initialization and K-means initialization. We show registration results on the trajectories generated using registration methods for visual odometry on the TUM dataset. The only difference is that GMMs are built with random or K-means initialization. For the TUM dataset, the objects are scattered and overlapped with each other in a small amount. At this time, the advantage of K-means initialization compared to random initialization is obvious, as shown in Figure 4(a).

C. *Data Quantization*

We investigate comprehensively on different quantization strategies and the results are shown in Table I on Stanford Bunny Dataset. All the GMMs in this experiment have $K = 64$ components. The rotation error is $0.4219°$ and the translation error is 0.526 mm when 32-bit floating-point numbers are used (Exp1). When employing 16-bit dynamic-precision quantization, only $0.02°$ rotation loss and 0.002 mm translation loss are introduced (Exp2). However, when using 8-bit quantization configuration, the mapping performance degrades significantly, resulting in $0.2°$ rotation loss and 0.15 mm translation loss for registration (Exp2). As mentioned in Section III-B, the bit width and fixed-point position of the original input point cloud z and the intermediate variable g representing the probability may have a major impact on the iterative effect of EM algorithm. Therefore, Exp 4 uses 16-bit quantization for both of them and improves the mapping performance a lot, achieving almost the same result as Exp 1. Then Exp 5 only employs 16-bit quantization for g and introduces less than $0.01°$ loss. However, if only the bit width of input data z is 16 bit, the rotation loss is more than $0.2°$, losing lots of information. We hope the 8-bit fixed-point data is more when the mapping accuracy loss is negligible, so we choose the mixed-bit fixed-point data representation in Exp5 (only_g_fix16) as our final quantization strategy for GAME.

We also test GMM mapping with different quantization strategies on the TUM dataset and get a similar conclusion. Since the real environment is more complex, we use $K = 128$ components for all GMM maps. As shown in Figure 4(b), low-bit quantization has almost no effect on the quality of GMM maps constructed from real environments, and even achieves better performance than 32-bit floating-point numbers. Furthermore, the bit width of probability g is of significant influence, where 16-bit data representation improves mapping obviously compared to 8-bit.

TABLE I
REGISTRATION RESULTS ON THE STANFORD BUNNY (K=64)

Experiment	Exp1	Exp2	Exp3	Exp4	**Exp5**	Exp6
Data bw $[\boldsymbol{z}]$	float	16	8	16	**8**	16
Weight bw $[\boldsymbol{\pi}]$	float	16	8	8	**8**	8
Mean bw $[\boldsymbol{\mu}]$	float	16	8	8	**8**	8
RStd bw $[\frac{1}{\sigma}]$	float	16	8	8	**8**	8
Prob bw $[g]$	float	16	8	16	**16**	8
Resp bw $[r]$	float	16	8	8	**8**	8
Rotation Error (°)	0.432	0.434	0.641	0.453	**0.462**	0.61
Translation Error (mm)	0.526	0.528	0.67	0.569	**0.587**	0.655
BRAM (%)	12	6	3	5	5	3
DSP (%)	204*1	25	0	25*2	10*2	15
FF (%)	105*1	8	5	4	4	5
LUT (%)	189*1	36	36	28*2	33*2	30

*1 These data is from HLS synthesis. It cannot be implemented on FPGA.
*2 Exp5 uses much less DSPs by using more LUT than Exp4. The saved DSPs can be used to for other computation-intensive robotics algorithms, such as CNN.

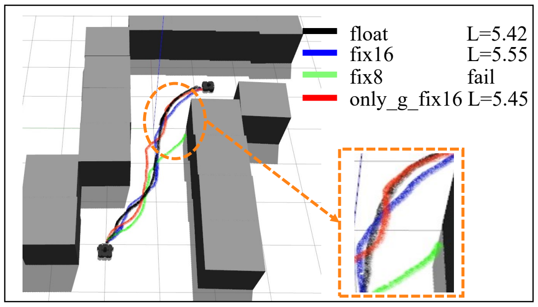


Fig. 5. The navigation path using GMM maps built with different quantization strategies. L corresponds to the navigation path length. The shorter path length means the navigation based on the map is more successful. The black trajectory (float) is the smoothest and shortest. Our proposed quantization strategy (only_g_fix16 as Exp5 in Table I), corresponding to the red trajectory, achieves almost the same result as floating-point numbers.

Finally, we verify the functions of GAME's mapping and navigation in the simulation environment, and at the same time, study the influence of different quantization strategies on the navigation path. The results are shown in Figure 5. Because low-bit quantization will ignore some information in the scene, for example, an object with too few points will not be constructed, the smoothness of the navigation path will be affected to a certain extent, but the total path length is basically unchanged. But the full 8-bit quantization strategy, corresponding to the green path in Figure 5, will cause excessive loss of scene information, and the robot will get stuck at a corner with less scene information, leading to navigation failure. Therefore, we believe that our mixed-bit quantization is very important for GMM mapping.

The results show that our proposed quantization strategy will introduce negligible loss to the GMM mapping performance. With the mixed-bit quantization, we can use much shorter bit width for data with a small dynamic range while still achieving comparable results. For example, compared with all 16-bit quantization, 8/16-bit quantization (Exp5 in Table I) almost halves the storage space for input and intermediate data and reduces the complexity of multiplication greatly.

TABLE II
RESOURCE USAGE ON FPGA (K=128)

		BRAM	DSP	FF	LUT
Available		1824	2520	548160	274080
Usage	K-means	9%	20%	7%	62%
	EM	9%	20%	8%	63%
	GAME	9%	20%	8%	64%

TABLE III
PERFORMANCE COMPARISON BETWEEN GPU AND FPGA ($N = 90k$, MILLISECOND FOR EACH ITERATION)

K	GPU				FPGA			
	1080Ti		TX2		Virtex6 @ 150 MHz [14]		ZCU102 @ 250 MHz	
	EM [1]	KM [2]	EM	KM	EM	KM [3]	EM	KM
32	14.43	4.06	78.07	46.3	2.416	N/A	0.48	0.48
64	22	7.65	142.2	69.1	4.832	N/A	0.48	0.48
128	39.2	13.59	273.7	124.3	9.664	N/A	0.48	0.48

[1] It stands for the running time for one EM iteration.
[2] It is the running time for one K-means iteration.
[3] The K-means algorithm is not supported by [14].

D. Accelerator Performance

We test the GMM mapping algorithm on different GPU and FPGA platforms in the real environment. In addition to the GPU implementations described in Section V-A2, we also compare our systems with an FPGA implementation described in [14]. As we do not have the same experimental settings and datasets with the ones in [14], we take the best performance record in [14] with similar N and K. Their FPGA implementation is deployed in a Xilinx Virtex-6 FPGA running at 150 MHz.

The resources usage for different quantization strategies on ZCU102 is displayed in Table I. It is clear that quantization reduces the resource utilization significantly.

Table II shows that our unified dataflow and architecture enable efficient use of hardware resources. Integrating the functions of both EM and K-means, the GAME IP consumes almost the same BRAMs, DSPs, FFs and LUTs as a single K-means or EM IP. Therefore, we can save half of the hardware resources when deploying GAME for GMM mapping.

We demonstrate the performance of different methods in a variety of test cases. We use the D435 camera to obtain the depth image in the scene, then convert it into a point cloud. The point cloud will be processed and filtered to a fixed voxel size using PCL Library [28]. The average number of points is 90K and we employ 128 components of GMM to model the scene ($N = 90K$, $K = 128$). To evaluate the generalization and performance of GAME, we demonstrate the speed of each iteration in a variety of test cases in Table III. GAME achieves 81$\times$ faster than 1080Ti and 570$\times$ faster than TX2. GAME is also 20$\times$ faster than previous accelerator on Virtex6 FPGA [14].

As our design is fully-pipelined and the optimized hardware resource usage on ZCU102 support expansion of all computing units in an iteration, only one cycle is required to process one point. This iteration speed keeps constant when K changes. As we analyze in Section IV-F, our parallelism is much higher than GPU. Compared with [14], GAME is fully pipelined and makes full use of the computing resources. On the basis of increasing the operating frequency by $1.6\times$, GAME achieve an additional $10\times$ speedup.

It takes an average of 10 K-means and 25 EM iterations to construct a GMM map from a point cloud when $N = 90K$. Our ZCU102 can achieves 59 fps when $K = 128$, which is **over 60× faster than a 1080Ti** (0.9 fps) and **over 490× faster than a TX2** (0.12 fps).

VI. Related Work

A. GMM Mapping in Robot Application

GMM map is a probabilistic map representation that models the environment as a continuous distribution rather than a set of discrete cells. Such map representation can capture spatial dependencies and obtain compact features from sensor observations. Meadhra *et al.* [29] and Srivastava *et al.* [30] use GMM mapping as a memory-efficient method to derive occupancy at arbitrary resolutions without storing an occupancy grid map of the entire environment. Tabib *et al.* [5] present a robust distribution-to-distribution registration method to enable GMM mapping and navigation in subterranean environments. Dhawale *et al.* [31] leverage the geometric properties of GMM maps to achieve real-time collision checking and safe flight of UAVs given a time-parameterized trajectory. Corah *et al.* [32] and Tabib *et al.* [4] propose a framework based on GMM maps for multi-robot perception and exploration in large and cluttered 3D environments.

Although GMM map is superior to traditional discrete map representations in many robot applications, the computationally expensive and time-consuming map building process limits its wide range of use on embedded robot systems. In the above studies, all the programs for mapping run on GPU-accelerated desktops or mini-PC. As far as we know, we are the first to achieve a real-time GMM mapping and navigation system fully running on embedded FPGA platforms.

B. Acceleration for GMM

Kumar *et al.* [6] design a parallel implementation of EM for GMM for many-core architecture of GPU, following the single instruction multiple threads model. The method of data parallelism with CUDA has good scalability across cores. Chen *et al.* [33] propose an FPGA implementation of EM towards computerized tomography (CT) application, which modifies the algorithm to match the hardware platform and pipelines basic modules on the task level. Eckart *et al.* [34] construct a top-down hierarchy of GMMs to deal with the sparsification of r_{nk} and implement a corresponding hierarchical EM algorithm for compact representation of 3D point cloud data. The high parallelism of GPU allows for point-level parallelism for each step. He *et al.* [14] propose a pipeline-friendly EM-GMM algorithm that makes the data stream only once in each EM iteration. However, they use random initialization, resulting in poor GMM mapping performance. In addition, They do not consider lower data bit width to enhance parallelism. We extend this streaming data flow to K-means clustering as well as the EM iterations with a unified hardware architecture.

C. Fixed-point Data Representation

Although FPGA and GPU can be used for accelerating machine learning algorithms, including CNN [35] and GMM [36], the data-driven machine learning algorithms still suffer from prohibitive computation, storage, and bandwidth. Since floating-point arithmetic consumes a lot of computing and storage resources, recent works try to replace this representation with low-bit fixed-point data. For one thing, shorter bit width of each input data helps reduce the bandwidth and storage requirement. For another, a simplified data representation reduces the hardware resource cost for each operation to improve parallelism.

Previous work [19] proposes a fixed-point GMM accelerator on FPGA for video coding. They adopt a unified data conversion method to replace each floating-point variable with a 32-bit integer. This static integer and fraction word-length fail to consider the difference in the influence of different inputs on the final result. Moreover, to avoid overflow, the bit width of the intermediate data in this work is very high, which requires additional computing resources.

Qiu *et al.* [20] find that the dynamic range of input and intermediate data in a CNN layer differs across different layers. Thus it proposes an adaptive quantization method for CNN. Inspired by this previous work, we analyze the dynamic range of the input and intermediate data for GMM mapping and propose a mixed-bit fixed-point GMM mapping method. Thus, the hardware resource consumption is significantly reduced and the parallelism is increased.

VII. Conclusion

This paper presents a real-time and high-performance GMM mapping and navigation engine named GAME on embedded FPGA. By discovering the algorithmic similarity between K-means and EM, which are two important steps for GMM mapping, we design a unified dataflow and hardware architecture to minimize the hardware resource usage. Since the EM algorithm is computation-constrained, we explore different quantization strategies and propose a mixed-bit fixed-point GMM map building method to increase hardware parallelism. The experimental results show that our 8/16-bit quantization introduces negligible accuracy loss to the GMM mapping. When tested in the real scene, GAME on the ZCU102 runs at the speed of 59 fps, meeting the real-time 30 fps mapping requirements and achieving $490\times$ speedup over an embedded GPU solution.

Acknowledgment

This work is supported by National Natural Science Foundation of China (No. U20A20334, U19B2019, 61832007), National Key R&D Program of China (No. 2019YFF0301505). This work is also supported by Tsinghua EE Xilinx AI Research Fund, Tsinghua EE Independent Research Project, and Beijing Innovation Center for Future Chips.

References

[1] S. Thrun, "Learning occupancy grid maps with forward sensor models," *Autonomous robots*, vol. 15, no. 2, pp. 111–127, 2003.

[2] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: An efficient probabilistic 3d mapping framework based on octrees," *Autonomous robots*, vol. 34, no. 3, pp. 189–206, 2013.

[3] "Mars science laboratory data rates/returns," 2020. [Online]. Available: https://marsmobile.jpl.nasa.gov/msl/mission/communicationwithearth/data/

[4] W. Tabib, K. Goel, J. Yao, M. Dabhi, C. Boirum, and N. Michael, "Real-time information-theoretic exploration with gaussian mixture model maps." in *Robotics: Science and Systems*, 2019.

[5] W. Tabib, C. O'Meadhra, and N. Michael, "On-manifold gmm registration," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3805–3812, 2018.

[6] N. P. Kumar, S. Satoor, and I. Buck, "Fast parallel expectation maximization for gaussian mixture models on gpus using cuda," in *2009 11th IEEE International Conference on High Performance Computing and Communications*. IEEE, 2009, pp. 103–109.

[7] G. J. McLachlan and T. Krishnan, *The EM algorithm and extensions*. John Wiley & Sons, 2007, vol. 382.

[8] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[dl] a survey of fpga-based neural network inference accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 1, pp. 1–26, 2019.

[9] V. Melnykov and I. Melnykov, "Initializing the em algorithm in gaussian mixture models with an unknown number of components," *Computational Statistics & Data Analysis*, vol. 56, no. 6, pp. 1381–1395, 2012.

[10] C. Biernacki, G. Celeux, and G. Govaert, "Choosing starting values for the em algorithm for getting the highest likelihood in multivariate gaussian mixture models," *Computational Statistics & Data Analysis*, vol. 41, no. 3-4, pp. 561–575, 2003.

[11] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA, 1967, pp. 281–297.

[12] R. Maitra, "Initializing partition-optimization algorithms," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 6, no. 1, pp. 144–157, 2009.

[13] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[14] C. He, H. Fu, C. Guo, W. Luk, and G. Yang, "A fully-pipelined hardware design for gaussian mixture models," *IEEE Transactions on Computers*, vol. 66, no. 11, pp. 1837–1850, 2017.

[15] Z. Zivkovic, "Improved adaptive gaussian mixture model for background subtraction," in *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, vol. 2. IEEE, 2004, pp. 28–31.

[16] B. Zhou, J. Pan, F. Gao, and S. Shen, "Raptor: Robust and perception-aware trajectory replanning for quadrotor fast flight," *arXiv preprint arXiv:2007.03465*, 2020.

[17] L. Han, F. Gao, B. Zhou, and S. Shen, "Fiesta: Fast incremental euclidean distance fields for online motion planning of aerial robots," *arXiv preprint arXiv:1903.02144*, 2019.

[18] C. Guo, H. Fu, and W. Luk, "A fully-pipelined expectation-maximization engine for gaussian mixture models," in *2012 International Conference on Field-Programmable Technology*. IEEE, 2012, pp. 182–189.

[19] W. Chen, Y. Tian, Y. Wang, and T. Huang, "Fixed-point gaussian mixture model for analysis-friendly surveillance video coding," *Computer Vision and Image Understanding*, vol. 142, pp. 65–79, 2016.

[20] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[21] L. Keselman and M. Hebert, "Direct fitting of gaussian mixture models," in *2019 16th Conference on Computer and Robot Vision (CRV)*. IEEE, 2019, pp. 25–32.

[22] B. Eckart, K. Kim, and J. Kautz, "Hgmr: Hierarchical gaussian mixtures for adaptive 3d registration," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 705–721.

[23] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A benchmark for the evaluation of rgb-d slam systems," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 573–580.

[24] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. IEEE, 2004, pp. 2149–2154.

[25] "Intel RealSense^{TM} D435," 2018. [Online]. Available: https://www.intelrealsense.com/depth-camera-d435

[26] "Kmeans using PyTorch." 2021. [Online]. Available: https://github.com/subhadarship/kmeans_pytorch

[27] "Gaussian mixture models in PyTorch." 2021. [Online]. Available: https://github.com/ldeecke/gmm-torch

[28] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE international conference on robotics and automation*. IEEE, 2011, pp. 1–4.

[29] C. O'Meadhra, W. Tabib, and N. Michael, "Variable resolution occupancy mapping using gaussian mixture models," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 2015–2022, 2018.

[30] S. Srivastava and N. Michael, "Efficient, multifidelity perceptual representations via hierarchical gaussian mixture models," *IEEE Transactions on Robotics*, vol. 35, no. 1, pp. 248–260, 2018.

[31] A. Dhawale, X. Yang, and N. Michael, "Reactive collision avoidance using real-time local gaussian mixture model maps," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 3545–3550.

[32] M. Corah, C. O'Meadhra, K. Goel, and N. Michael, "Communication-efficient planning and mapping for multi-robot exploration in large environments," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 1715–1721, 2019.

[33] J. Chen, J. Cong, M. Yan, and Y. Zou, "Fpga-accelerated 3d reconstruction using compressive sensing," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, 2012, pp. 163–166.

[34] B. Eckart, K. Kim, A. Troccoli, A. Kelly, and J. Kautz, "Accelerated generative models for 3d point cloud data," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 5497–5505.

[35] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170. [Online]. Available: https://doi.org/10.1145/2684746.2689060

[36] M. A. Momen, M. A. Khalid, and M. A. M. Oninda, "Fpga-based acceleration of expectation maximization algorithm using high-level synthesis," in *2019 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, 2019, pp. 41–46.