Hybrid Implementation of Expectation Maximization Algorithm using CUDA and OpenMP for Gaussian Mixture Models

Ayush Raina
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore, India
ayushraina@iisc.ac.in

***Abstract*—The Expectation-Maximization (EM) algorithm for Gaussian Mixture Models (GMMs) presents substantial computational challenges for large-scale, high-dimensional datasets. We introduce a novel heterogeneous computing approach that synergistically combines GPU acceleration via CUDA with CPU parallelization using OpenMP. Our architecture strategically offloads the computation-intensive E-step to the GPU while executing the more synchronization-dependent M-step on multi-core CPUs, achieving optimal hardware utilization. Numerical stability in high dimensions is ensured through sophisticated log-space computations and adaptive regularization techniques. Performance analysis on synthetic datasets demonstrates remarkable speedups—approximately 73× over sequential implementations and significantly outperforming both pure-CPU (8-9×) and pure-GPU (4×) implementations. The hybrid design effectively balances computational throughput with memory transfer overhead, scales efficiently with increasing dataset size and model complexity, and exhibits superior convergence characteristics. This architecture represents a practical solution for accelerating mixture model estimation in data-intensive applications where both computational efficiency and numerical robustness are critical constraints.**

I. Introduction

Gaussian Mixture Models (GMMs) are powerful probabilistic models used extensively in machine learning for density estimation, clustering, classification, and anomaly detection. GMMs represent complex data distributions as a weighted sum of multivariate Gaussian distributions:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{1}$$

where π_k represents the mixture weights (which sum to 1), $\boldsymbol{\mu}_k$ is the mean vector, and $\boldsymbol{\Sigma}_k$ is the covariance matrix of the k-th component.

The Expectation-Maximization (EM) algorithm is the standard approach for estimating GMM parameters, iteratively refining model parameters to maximize the data likelihood. The EM algorithm consists of two main steps: (1) the Expectation step (E-step), which computes posterior probabilities (responsibilities) for each data point belonging to each Gaussian component, and (2) the Maximization step (M-step), which updates model parameters based on these responsibilities.

For large datasets with high dimensionality, these computations become extremely intensive, with the E-step and M-step both scaling as $O(NKD^2)$ for N samples, K components, and D dimensions per iteration until convergence. The M-step has lower computational complexity at $O(NKD)$ if diagonal covariance matrices are considered and it involves more complex dependencies between data points. This computational burden makes acceleration through parallel processing essential for practical applications. While previous approaches have typically focused on implementing the entire EM algorithm on either CPUs or GPUs or Multi-GPU, we propose a hybrid approach that strategically utilizes a **single GPU** alongside a **multi-core CPU**, mapping different parts of the algorithm to the hardware best suited for that computation.

In this paper, we present a hybrid implementation that:

1) Maps the computationally intensive, data-parallel E-step to the GPU using CUDA
2) Leverages multi-core CPUs with OpenMP for the more synchronization-heavy M-step
3) Addresses critical numerical stability challenges in high-dimensional GMM implementations through log-space computations
4) Minimizes data transfer between CPU and GPU, a common bottleneck in heterogeneous computing
5) Implements dimension-aware regularization techniques for robust covariance estimation

The rest of the paper is organized as follows: Section II details the methodology and hybrid algorithm design, including workload analysis, parallelization strategies, and implementation details. Section **??** presents experimental results and performance analysis. Section IV concludes the paper and discusses directions for future work.

II. Methodology

A. GMM Background and EM Algorithm

The log-likelihood function for a GMM with K components for a dataset $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ is:

$$\log p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{i=1}^{N} \log \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right) \tag{2}$$

The EM algorithm iteratively maximizes this log-likelihood by alternating between two steps:

E-step: Compute the posterior probability (responsibility) that component k takes for explaining data point $\mathbf{x}_i$:

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \tag{3}$$

M-step: Update the model parameters using computed responsibilities:

$$\pi_k^{new} = \frac{N_k}{N} \quad \text{where} \quad N_k = \sum_{i=1}^{N} \gamma_{ik} \tag{4}$$

$$\boldsymbol{\mu}_k^{new} = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} \mathbf{x}_i \tag{5}$$

$$\boldsymbol{\Sigma}_k^{new} = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} (\mathbf{x}_i - \boldsymbol{\mu}_k^{new})(\mathbf{x}_i - \boldsymbol{\mu}_k^{new})^T \tag{6}$$

The algorithm iterates until convergence, typically determined by monitoring changes in log-likelihood between iterations.

B. Algorithm Overview

Our hybrid implementation follows this high-level workflow:

Algorithm 1 Hybrid CUDA-OpenMP EM Algorithm for GMMs

```
1: Initialize model parameters on CPU: $\boldsymbol{\pi}$, $\boldsymbol{\mu}$, $\boldsymbol{\Sigma}$
2: Precompute precision matrices $\boldsymbol{\Sigma}_k^{-1}$ and normalizer terms $c_k = -\frac{D}{2}\ln(2\pi) - \frac{1}{2}\ln|\boldsymbol{\Sigma}_k|$
3: while not converged and iterations < max_iterations do
4:     Transfer model parameters to GPU
5:     Execute E-step on GPU using log-space CUDA kernels:
           $\ln p(x_i|k) = \ln \pi_k + c_k - \frac{1}{2}(x_i - \mu_k)^T \boldsymbol{\Sigma}_k^{-1}(x_i - \mu_k)$   (7)
6:     Calculate responsibilities $\gamma_{ik}$ using the log-sum-exp trick for numerical stability
7:     Transfer responsibilities $\gamma_{ik}$ and log-likelihood $\mathcal{L}$ to CPU
8:     Perform M-step using OpenMP on CPU to update parameters (as described in Section II)
9:     Check convergence: $|\mathcal{L}_{\text{new}} - \mathcal{L}_{\text{old}}| < \epsilon$
10:    if not converged then
11:        Precompute updated precision matrices and normalizers
12:    end if
13: end while
14: return Final model parameters $\boldsymbol{\pi}$, $\boldsymbol{\mu}$, $\boldsymbol{\Sigma}$
```

The algorithm iteratively refines GMM parameters by delegating the computationally intensive E-step to the GPU while performing parameter updates in the M-step on the CPU. This division leverages the strengths of each hardware component.

C. E-step: GPU Implementation with Log-Space Computation

The E-step is the most computationally intensive part of the EM algorithm, especially for high-dimensional data. Our GPU implementation focuses on both performance and numerical stability.

1) Log-Space Computation for Numerical Stability: Computing Gaussian probabilities in high dimensions can quickly lead to numerical underflow due to the exponential term with potentially large negative exponents. We implement log-space computation as follows:

The log of the multivariate Gaussian PDF is:

$$\log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{D}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}| - \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) \tag{8}$$

For efficiency, we precompute the precision matrices $\boldsymbol{\Sigma}^{-1}$ and the normalizer terms $-\frac{D}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}|$ on the CPU, allowing us to reuse these values for all data points.

After computing log probabilities, we use the log-sum-exp trick to compute responsibilities while maintaining numerical stability:

Algorithm 2 CUDA Kernel: Calculate Responsibilities

```
1: $i \leftarrow$ thread index corresponding to data point
2: if $i <$ n_samples then
3:     max_log_prob $\leftarrow -\infty$ {Initialize to negative infinity}
4:     for $k = 0$ to n_components $- 1$ do
5:         max_log_prob $\leftarrow \max($max_log_prob, log_probs$[i, k])$
6:     end for
7:     sum_exp $\leftarrow 0.0$
8:     for $k = 0$ to n_components $- 1$ do
9:         sum_exp $\leftarrow$ sum_exp $+ \exp($log_probs$[i, k] -$ max_log_prob$)$
10:    end for
11:    log_sum_exp $\leftarrow$ max_log_prob $+ \log($sum_exp$)$
12:    log_likelihood$[i] \leftarrow$ log_sum_exp {Store for convergence check}
13:    for $k = 0$ to n_components $- 1$ do
14:        resp$[i, k] \leftarrow \exp($log_probs$[i, k] -$ log_sum_exp$)$
15:    end for
16: end if
```

The log-sum-exp trick involves finding the maximum log probability for numerical stability:

$$\log \sum_k \exp(a_k) = m + \log \sum_k \exp(a_k - m) \tag{9}$$

where $m = \max_k a_k$. This prevents underflow by shifting values to a numerically stable range.

2) GPU Memory Access Optimization: Our CUDA implementation prioritizes efficient memory access patterns in our numerically stable log-space kernels:

- **Coalesced Global Memory Access**: Data is stored in row-major layout to ensure adjacent threads access adjacent memory locations. In our GPU kernels, array

indexing follows patterns like ”log_probs[sample_idx × n_components + k]”, maximizing memory bandwidth utilization.
- **Register Optimization**: Critical variables such as ”max_log_prob”, ”sum_exp”, and intermediate calculation results are kept in local variables (automatically stored in registers) to minimize latency in our calculation-intensive kernels.
- **Memory Layout Conversion**: We implement specialized conversion functions that efficiently transform between Eigen’s column-major format on the CPU and the row-major format optimized for CUDA operations via our `eigenToArray` and `arrayToEigen` utility functions.
- **Precomputation Caching**: We precompute and cache precision matrices and normalizers on the CPU, transferring these values to the GPU only once per iteration to minimize redundant calculations and memory transfers.

Our implementation carefully manages data structures to match the computational patterns of log-space operations, ensuring efficient access to precomputed terms during the calculation of Gaussian probabilities and responsibilities.

3) CUDA Kernel Configuration: Our implementation utilizes two specialized CUDA kernels for the E-step, one for computing log probabilities and other for converting these to responsibilities using the log-sum-exp trick. We choosed standard block size of 256 threads per block to provide the best balance between occupancy and memory usage.

D. M-step: OpenMP Implementation

The M-step updates model parameters using the responsibilities calculated in the E-step. We implement this using OpenMP to leverage CPU parallelism:

Algorithm 3 M-step with OpenMP Parallelization

```
1: $N_k \leftarrow$ sum of responsibilities for each component
2: #pragma omp parallel for
3: for $k = 0$ to $n\_components - 1$ do
4:     $\pi[k] \leftarrow N_k[k]/n\_samples$
5: end for
6: #pragma omp parallel for
7: for $k = 0$ to $n\_components - 1$ do
8:     if $N_k[k] > 0$ then
9:         Initialize $\mu[k]$ and $\Sigma[k]$ to zero
10:        for $i = 0$ to $n\_samples - 1$ do
11:            $\mu[k] \leftarrow \mu[k] + \text{resp}[i,k] \times X[i]$
12:        end for
13:        $\mu[k] \leftarrow \mu[k]/N_k[k]$
14:        for $i = 0$ to $n\_samples - 1$ do
15:            diff $\leftarrow X[i] - \mu[k]$
16:            $\Sigma[k] \leftarrow \Sigma[k] + \text{resp}[i,k] \times \text{diff} \times \text{diff}^T$
17:        end for
18:        $\Sigma[k] \leftarrow \Sigma[k]/N_k[k] + \text{reg\_factor} \times I$
19:    end if
20: end for
```

Our implementation parallelizes component-wise operations, as each component’s parameters can be updated independently. We use OpenMP’s ‘parallel for‘ directive to distribute iterations across CPU threads, with dynamic scheduling for better load balancing when component update times vary.

E. Precomputation of Precision Terms

After each M-step update, we precompute precision matrices and normalizer terms for the next E-step:

Algorithm 4 Precomputation of Precision Terms

```
1: #pragma omp parallel for
2: for $k = 0$ to $n\_components - 1$ do
3:     Apply covariance regularization to $\Sigma[k]$
4:     $\Sigma^{-1}[k] \leftarrow$ inverse of $\Sigma[k]$
5:     $normalizers[k] \leftarrow -\frac{D}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma[k]|$
6: end for
```

This precomputation significantly reduces the computational load in the E-step, as these terms remain constant for all data points within an iteration. In the actual implementation of above function, we have taken care of numerical instabilities.

F. Implementation Details

Our implementation uses C++ with CUDA for GPU acceleration and OpenMP for CPU parallelization, with Eigen for matrix operations. We developed specialized conversion functions between Eigen’s column-major format and CUDA’s row-major layout to ensure optimal memory access patterns.

The implementation was tested on an Intel Core i9-12900K (16 cores, 24 threads) with NVIDIA GeForce GTX 1660 (6GB) GPU. Our OpenMP implementation automatically scales to utilize all available CPU cores, providing optimal performance across different system configurations. This design minimizes transfer overhead between CPU and GPU while optimizing cache utilization on both architectures.

III. Experiments and Results

A. Experiment 1: Synthetic Data and Performance Evaluation

In this experiment, we evaluate the performance of our hybrid implementation on synthetic datasets of varying sizes (2M, 4M, 6M, 8M, and 10M data points) with 5 clusters and 10-dimensional data. We run 5 iterations of each variant

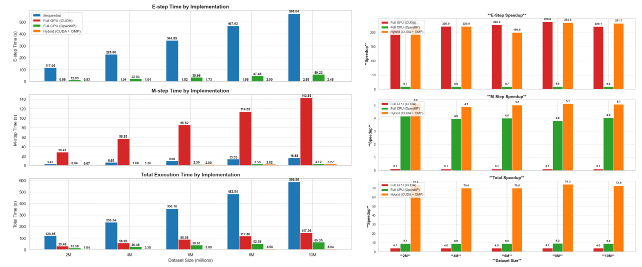


Fig. 1. Execution times and Speedups over sequential implementation are shown.

here and the data reveals impressive performance gains from parallel implementations across different dataset sizes. The sequential version shows poor scalability, with execution time increasing from ∼117s to ∼568s as data grows.

For the E-step, GPU implementation excels with ∼220-234x speedup, while CPU parallelization achieves only ∼9.8x. In contrast, the M-step shows reversed performance patterns with GPU performing poorly (∼0.1x) and CPU showing moderate improvement (∼4.0x).

The hybrid approach effectively combines the strengths of both platforms, using GPU for the computation-heavy E-step and CPU for the more sequential M-step. This strategic combination delivers the best overall performance with total speedups of 70-74x, significantly outperforming either GPU-only (∼4.0x) or CPU-only (∼9.0x) implementations.

These results highlight the importance of matching algorithm components to the most suitable hardware architecture for maximum performance gains in large-scale data processing.

B. *Experiment 2: Comparison with Azizi's Paper*

We generated a synthetic dataset of 1 million points as specified in Azizi's paper and compared our hybrid implementation with the Python, NumPy, Numba, and CuPy implementations provided in the paper. We found that when starting the GMM fitting process with the initial parameters given in the paper, neither their implementations nor our hybrid version converged within 2000+ iterations. However, when initialized with the same random parameters, our hybrid implementation converged in lesser iterations than all other implementations.

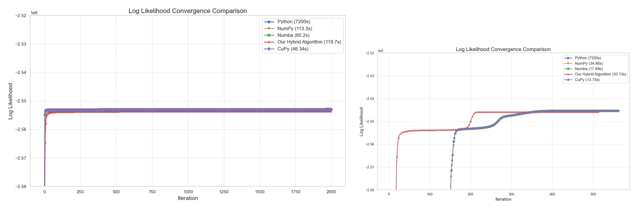
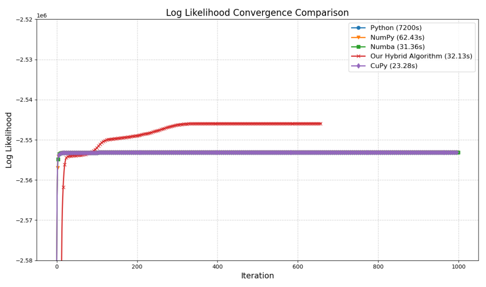


Fig. 2. Left: Log-likelihood when starting with the initial parameters from the paper. Right: Log-likelihood when starting with random parameters.

Since Numba and CuPy are highly optimized libraries, their iteration speed is much faster than our implementation because of vectorization and other optimizations. If we run their code with their initial parameters and our code as usual, we get the following plot, in which we converge much early:



Experiment 3: Testing our Hybrid Algorithm on IRIS Dataset

To validate our hybrid algorithm on real-world data, we tested it on the classic Iris dataset, which contains 150 samples from three iris species with four features each (sepal length, sepal width, petal length, and petal width). The dataset presents an excellent benchmark for clustering algorithms due to its well-defined structure with one linearly separable class and two partially overlapping classes.

Our hybrid GMM implementation achieved excellent clustering results on this dataset, as evidenced by multiple evaluation metrics: Misclassification Rate (MCR) of 0.03333, Adjusted Rand Index (ARI) of 0.90387, and F-measure of 0.96658. These metrics indicate that our algorithm correctly identified the underlying clusters with high accuracy, making only 5 misclassifications out of 150 samples.

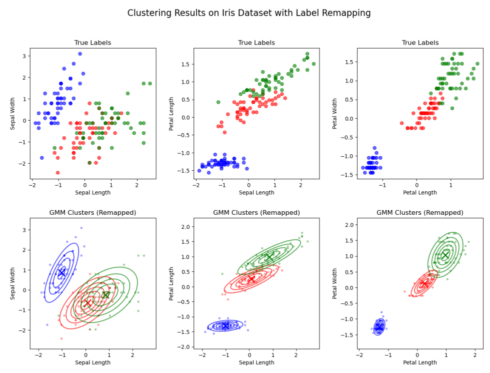


Fig. 3. Top row shows the true class labels across three feature pairs. Bottom row displays the fitted GMM clusters with corresponding density contours. The close match between true labels and model predictions demonstrates the algorithm's effectiveness at recovering the natural clusters in the data.

IV. Conclusion and Future Work

In this paper, we presented a hybrid parallel implementation of the EM algorithm for GMMs that effectively leverages both GPU and CPU resources. The hybrid approach demonstrates very good performance compared to pure CPU and pure GPU implementations, with performance advantages that scale with dataset size and dimensionality. Details of the regularization techniques are not discussed here for brevity but can be found in our implementation.

Future work could explore several directions:

1) **Optimized Linear Algebra Libraries**: Replacing Eigen with CUDA-optimized libraries like cuBLAS and cuSOLVER for matrix operations to further accelerate covariance computations and matrix inversions.
2) **Sparse GMMs**: Implementing sparse covariance matrix representations for very high-dimensional data.
3) **Integration with Deep Learning**: Exploring hybrid deep learning models that incorporate GMMs as components.

Our implementation demonstrates that effectively leveraging heterogeneous computing resources can significantly accelerate machine learning algorithms, enabling the analysis of larger and higher-dimensional datasets than previously practical.

References

[1] I. Azizi, “Parallelization in Python – An Expectation-Maximization Application,” Département des Opérations, Université de Lausanne, Jun. 2023. [Online]. Available: https://iliaazizi.com/projects/em_parallelized/report.pdf

[2] N. S. L. Phani Kumar, S. Satoor, and I. Buck, “Fast parallel expectation maximization for Gaussian mixture models on GPUs using CUDA,” in *11th IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2009, pp. 103–109. [Online]. Available: https://ieeexplore.ieee.org/document/5166982