

Accelerating Gaussian Mixture Model Training with Parallel EM on GPUs

Clint Zhu (clintz)

Chenghui Yu (cyu4)

<https://seris370.github.io/15618-project-web/>

Summary:

We will implement a CUDA-accelerated version of the Expectation-Maximization (EM) algorithm used to fit Gaussian Mixture Models (GMMs). By parallelizing the responsibility calculation and parameter updates in EM, we aim to achieve significant speedups over a CPU implementation, especially for high-dimensional and large-scale datasets. Finally, as comparison metrics, we plan to compare our results against scikit-learn and PyTorch baselines on GPUs and CPUs to evaluate scalability and performance.

Background:

Gaussian Mixture Models (GMMs) are commonly used to model data as a mixture of multivariate Gaussian distributions. Training a GMM using the EM algorithm involves two main steps:

- Expectation-Step: For each data point and Gaussian component, we should compute the responsibility, which is the probability that the component generated the data point.
- Maximization-Step: Update the parameters, such as mean, covariances, and mixture weights, using the responsibilities.

Since both steps involve dense looping over the large number of index pairs (depending on the data size), this setup makes the EM algorithm ideal for parallel computation. However, current implementations of GMM in scikit-learn are CPU-based, and it's slow when the dataset is large.

The Challenge:

For the E-step, even though the responsibility computation for each index pair is independent of each other, the key challenge for this step would be that we need to fetch point data and component parameters, and they will potentially result in scattered memory access with limited spatial locality if not laid out properly. Additionally, for the M-step which requires summing up all data points for each component, due to the intrinsic reduction dependencies, incorrect grouping or synchronization can lead to race

conditions or inaccurate statistics. Furthermore, we need a smart design of memory access pattern to ensure locality, since if we just let each thread process one data point or one Gaussian distribution, the feature vector is locally reused, but for each iteration over k, model parameters are scattered in memory. Thus, it's important to find the concession between these two localities.

Resources:

The main hardware resource we will depend on would be GHC and PSC machines. We will write our cuda version of EM from scratch, but the rest of GMM will be written in python using NumPy or PyTorch functions if necessary for a full working classification pipeline. As primary technical references, we will use [1] for EM's mathematical foundations clearly for GMMs, and [2] which discusses design choices and lessons learned in a shared-memory parallel context. Although their work was on OpenMP and not CUDA, their framework offers useful insight for parallel structure and memory layout considerations.

Additionally, to the best of our knowledge, existing GPU libraries like NVIDIA cuML do not provide GMM support, which makes our project possibly meaningful and impactful.

Goals & Deliverables:

Minimum Plan to Achieve:

- Implement a correct parallel GMM training loop with:
 - Parallel E-step: compute log-probabilities and responsibilities on GPU
 - Parallel M-step: compute new means, covariance matrices, and mixture weights
- Benchmark vs. single-threaded and multi-threaded CPU baselines on synthetic Gaussian datasets
- Provide a speedup analysis on varying N, K, and feature dimension D

Hope to Achieve:

- Optimize GPU kernels using:
 - Shared memory tiling for repeated matrix access
 - Warp-level reductions for numeric stability and speed
 - Per-Gaussian stream parallelism for high K
- Support full covariance matrices in addition to diagonal ones
- Achieve 3x speedup over CPU baseline for moderate-to-large datasets on GHC Machine

Fallback Goals:

- Implement only diagonal covariance version and avoid shared-memory optimizations.
- Limit tests to moderate sizes

Demo:

- A classification pipeline that has the following stages
 - Accepts input dataset
 - Partitions into training dataset and test dataset
 - Performs EM on the training dataset
 - [Optional] Outputs visual training classification results (depending on dataset dimensions)
 - Performs inference on the test dataset
 - [Optional] Outputs visual test classification results (depending on dataset dimensions)

Platform Choice:

We think CUDA is ideal because the workload is fairly parallel across data-point versus Gaussian-component pairs, making it suitable for mapping to thread/block patterns. Additionally, GPUs can provide an order-of-magnitude higher memory bandwidth than CPUs, which is ideal for our case to handle GMM's data-heavy loops.

Schedule:

Now - Nov 19: Build baseline CPU implementation; set up synthetic data and correctness tests.

Nov 19 - Nov 24: Implement parallel E-step kernel (compute log-likelihoods and responsibilities).

Nov 24 - Nov 28: Implement M-step parallel reductions for means and covariances.

Nov 28 - Dec 2: Optimization

Dec 2 - Dec 4: Benchmarking and experimenting

Dec 4 - Dec 8: Finish final report

References:

- [1] Bilmes, J. A. (1998). A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models. *International computer science institute*, 4(510), 126.
- [2] Kwedlo, W. (2014, February). A parallel EM algorithm for Gaussian mixture models implemented on a NUMA system using OpenMP. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (pp. 292-298). IEEE.