Accelerating Gaussian Mixture Model Training with Parallel EM on GPUs

15-618 Final Project Report

Clint Zhu[1] **Chenghui Yu**[1]

[1]Carnegie Mellon University, Language Technologies Institute

{clintz, cyu4}@andrew.cmu.edu

project website

Abstract

We present a parallel GPU-based implementation of the Gaussian Mixture Model Expectation–Maximization algorithm. By fusing the E-step and M-step into a single CUDA kernel and using shared-memory block-wise reductions, our design eliminates the severe contention caused by global atomic operations. The resulting system runs 3× faster than scikit-learn, while maintaining clustering accuracy within 1–2%. These results demonstrate that GPU parallelism is highly effective for scaling GMM training to large datasets.

1 Summary

We implemented a highly optimized parallel Gaussian Mixture Model (GMM) Expectation–Maximization (EM) solver on GPUs using CUDA. Our deliverables include: (1) a fused CUDA EM kernel with shared-memory accumulation and multi-stage reductions, (2) a Python interface for benchmarking against scikit-learn and a sequential CPU baseline, and (3) a full evaluation pipeline with accuracy and runtime comparisons across multiple synthetic and real datasets. We tested our system on large sklearn datasets using GHC machines with NVIDIA GPUs (GeForce RTX 2080) and CPU (Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz). Across multiple synthetic and real datasets, our GPU solver achieves substantial performance gains, up to 400× faster than the sequential baseline and consistently faster than scikit-learn for medium and large datasets, while preserving clustering accuracy. These results demonstrate that GPUs are well-suited for EM-based mixture model training and that careful kernel fusion and reduction design are essential for unlocking their full performance potential.

2 Background

Gaussian Mixture Models (GMMs) are a probabilistic clustering method in which data is modeled as a weighted sum of K Gaussian components. Fitting a GMM is typically performed using the Expectation–Maximization (EM) algorithm (Bilmes et al., 1998), an iterative procedure alternating between computing per-point responsibilities (E-step) and updating component parameters (M-step), as shown in Figure 1. The key data structures in this algorithm are the dataset matrix $X \in R^{N \times D}$, the component means $\mu_k \in R^D$, diagonal covariance vectors $\Sigma_k \in R^D$, and the mixing weights π_k. The input to the algorithm is a dataset of N points, and the output consists of the learned GMM parameters as well as the soft cluster assignments.

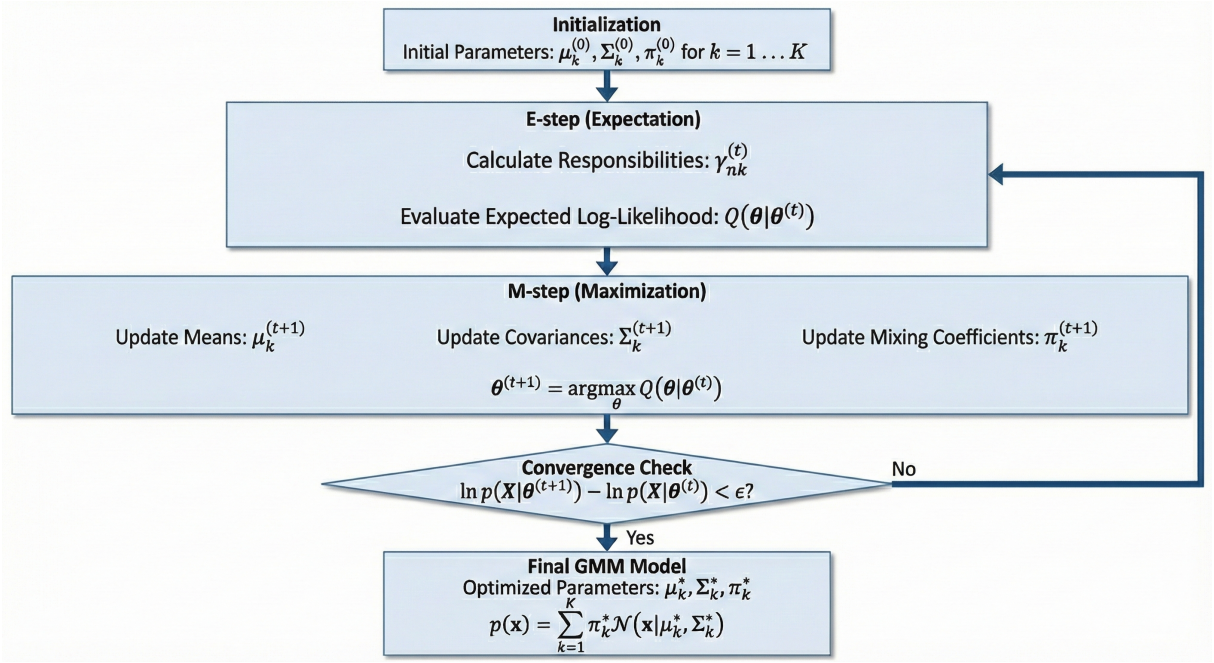

Figure 1: GMM algorithm

The computationally dominant portion of EM is the E-step, where for each data point x_n and component k we evaluate the Gaussian log-likelihood

$$\log p(x_n \mid k) = -\frac{1}{2}\left[D\log(2\pi) + \sum_{d=1}^{D}\log\sigma_{k,d} + \sum_{d=1}^{D}\frac{(x_{n,d}-\mu_{k,d})^2}{\sigma_{k,d}}\right].$$

This step is the primary computational bottleneck for large datasets and high-dimensional mixtures. The subsequent M-step reduces pointwise responsibilities into aggregate quantities, namely effective component counts N_k, weighted sums $\sum_n r_{nk}x_n$, and variances, which also

benefit from parallel reduction. Prior work has demonstrated that both phases of EM exhibit substantial data parallelism and can be efficiently parallelized on shared-memory systems (Kwedlo, 2014). Overall, GMM requires $O(NKD^3)$ floating-point operations (Pinto and Engel, 2015).

The workload is naturally data-parallel: each point–component pair (n, k) is independent during the E-step, and the reductions in the M-step are associative, enabling hierarchical parallelization. There are no loop-carried dependencies across data points, and operations on each x_n access only a small, contiguous region of memory, giving excellent spatial locality. This structure maps well to SIMD execution on GPUs, where thousands of threads can independently evaluate Gaussian likelihoods. The only non-trivial dependencies arise during reductions, where many threads contribute to shared accumulators, which can be efficiently handled through shared-memory buffering and staged block-level reductions. Because both the dataset and model parameters can remain resident in device memory throughout the EM iterations, the algorithm achieves high memory locality and low communication overhead, making it an excellent candidate for CUDA parallelization.

3 Approach

3.1 Implementation Technologies and Environment

Our implementation uses a hybrid programming model combining high-performance CUDA kernels with a Python-based experimentation and visualization environment. The host-side control logic, including data loading, parameter initialization, and Python integration, is written in C++ and exposed to Python through `pybind11`. This allows Python scripts to invoke the GPU kernels directly while maintaining low-overhead access to device memory, as shown in Listing 1. The core computational kernels for both the E-step and M-step are implemented in CUDA C/C++ and executed on the NVIDIA GeForce RTX 2080 GPUs available in the CMU GHC clusters.

To ensure correctness, we first implemented a fully sequential CPU reference in Python following the standard EM formulation. This implementation served as a validation baseline throughout development and enabled step-by-step numerical comparisons against every GPU

kernel revision.

Listing 1: Environment setup commands

```
### setup env
python3 -m venv --system-site-packages ~/gmm_env
source ~/gmm_env/bin/activate
pip install --upgrade pip
pip install -r requirements.txt

### build
make clean && make

### run demo
source ~/gmm_env/bin/activate
python3 python/demo_train.py
```

3.2 Parallel Mapping and Kernel Design

The GMM EM algorithm consists of two main steps, both mapped to the GPU's parallel architecture, as shown in Figure 2. The key decision was how to handle the large number of data points, N, across the available processing units.

3.2.1 Expectation Step (E-step) Mapping

The E-step, which calculates the responsibility r_{nk} (the likelihood that data point n belongs to k), was mapped using data parallelism. The main strategy was to assign the full E-step calculation for a subset of data points n to individual GPU threads. The CUDA kernel was launched with a Grid-Stride Loop over the N data points, and each thread reads its assigned feature vector and iterates through all K mixture components to compute the log-probabilities and responsibilities. This design choice uses batched memory access for efficiency, and it was effective immediately without too much optimization, resulting in speedups of up to 8.52x over the CPU baseline for large synthetic datasets.

3.2.2 Maximization Step (M-step) Mapping and Optimization

The M-step, which sums the weighted contributions ($\sum_{n=1}^{N} r_{nk}\mathbf{x}_n$, etc.) across all N data points, is fundamentally a global reduction operation. This presents the biggest challenge in parallelizing EM. Our initial implementation of the M-step accumulation used Global Memory atomicAdd.

We initially assume that the small number of accumulation targets ($K \times (1 + 2D)$ statistics) would simplify the code. However, this immediately led to severe contention and serialization. Since thousands of threads across the GPU try to simultaneously update the exact same memory locations, the GPU's parallelism was effectively nullified, making this step the primary performance bottleneck. We realized that for large N, the volume of conflicting writes outweighed the simplicity of using global atomics.

To fix the performance problem, we changed the way the M-step works completely by using Block-Wise Reduction. Instead of having thousands of threads fight over the same final memory spot, we broke the accumulation into two fast stages. First, we applied Kernel Fusion, which means we merged the calculation of the responsibility (r_{nk}) and the first part of the accumulation into a single kernel. This saves time by avoiding writing the huge intermediate responsibilities array to slow global memory. Second, within this fused kernel, we made each group of threads do its counting locally. Each block saves its intermediate counts into Shared Memory, which is a very fast memory right on the processor chip. This is the key: since only the threads within that small block can access this Shared Memory, the contention is almost eliminated. Once a thread block finishes its local count, one single thread in that block writes the block's final total to a unique space in a global partial-sum array. This single write is very fast and efficient. Finally, a separate, small kernel is launched to quickly add up all these partial totals to get the single, correct final answer. This entire multi-stage reduction process is much faster than the initial simple but slow global atomic approach.

3.3 Iterative Development and Refinement

Our project involved iterative optimization, moving from a functional baseline to a highly efficient parallel solution. After establishing the sequential CPU baseline and the full GMM training pipeline, we focused on implementing and validating the parallel E-step. The initial bottleneck experienced with the M-step's reliance on global atomics motivated the shift to the Block-Wise Reduction architecture. This strategic pivot, while requiring more complex kernel design, was necessary to achieve scalable speedups, confirming our effort to arrive at a robust solution. The remaining work focuses on finalizing the M-step implementation and proceeding

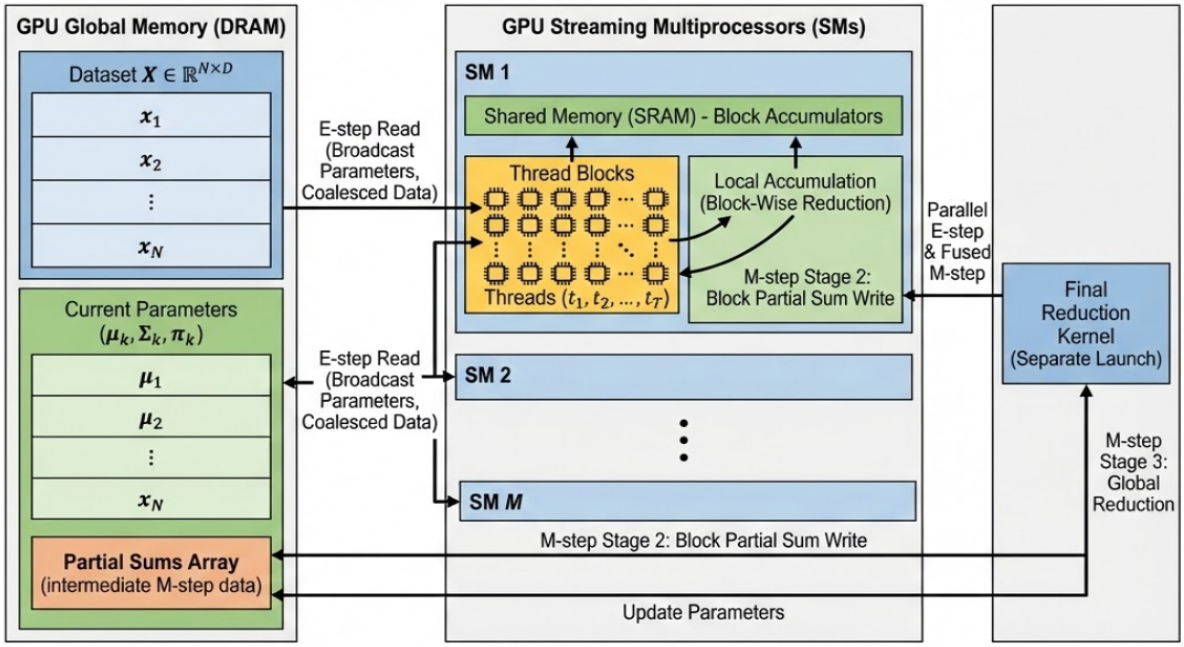

Figure 2: Speedup of sklearn and CUDA Implementations Compared to Sequential Baseline

with comprehensive benchmarking.

4 Results

4.1 Experimental Setup

All experiments were run on the five 2D datasets described earlier: *Iris*, *Wine*, *Moons*, *Circles*, and *Blobs*. Each dataset provides different levels of difficulty and cluster separability. The dataset size N is configurable for *Moons*, *Circles*, and *Blobs*. For each dataset we configured K to match the ground-truth number of clusters and compared three implementations:

1. **Scikit-learn GMM (CPU)** — optimized, multi-threaded reference implementation.
2. **Sequential CPU EM** — our single-threaded baseline implementation in Python.
3. **Fused CUDA EM** — our optimized GPU implementation.

For every run, we measured:

- **Wall-clock runtime** of the entire EM procedure (500 iterations).
- **Clustering accuracy**, computed by label-permutation alignment against the scikit-learn reference clustering.

The CUDA implementation was executed on an NVIDIA GeForce RTX 2080 GPU, while CPU baselines ran on an Intel Core i7 processor in the GHC machines. All implementations used identical initialization and stopping conditions. Clustering results of Iris and Blobs by scikit-learn and CUDA are visualized in Figure 3.

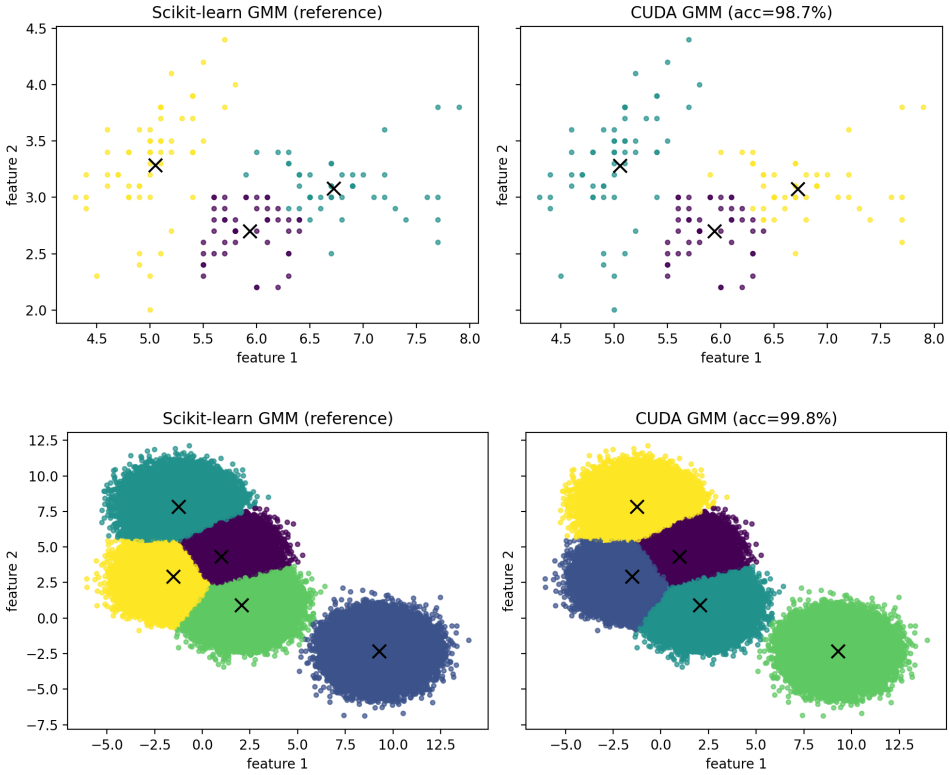

Figure 3: Clustering results of scikit-learn versus CUDA

4.2 Overall Performance Comparison

Table 1 summarizes the runtimes and accuracies for all datasets. Scikit-learn provides an upper bound on high-quality CPU performance, while the sequential CPU implementation reflects a true single-threaded baseline. The fused CUDA version achieves substantial speedups on medium and large datasets, with performance improvements exceeding two orders of magnitude in some cases.

4.3 Speedup Analysis

We define speedup as:

$$S = \frac{T_{\text{seq CPU}}}{T_{\text{CUDA}}}.$$

Figure 4 illustrates the speedup for each dataset (placeholder). The most important observation

Dataset	N	K	Method	Runtime (s)	Accuracy
Iris	150	3	sklearn	0.019	1.00
			sequential	0.061	0.987
			CUDA	0.086	0.987
Wine	178	3	sklearn	0.0036	1.00
			sequential	0.066	0.966
			CUDA	0.0081	0.961
Moons	200k	2	sklearn	0.175	1.00
			sequential	18.64	0.998
			CUDA	0.0726	0.998
Circles	200k	2	sklearn	0.130	1.00
			sequential	18.70	0.996
			CUDA	0.0573	0.992
Blobs	500k	5	sklearn	0.503	1.00
			sequential	98.87	0.998
			CUDA	0.207	0.998

Table 1: Runtime and accuracy comparison across all datasets.

is that speedup increases significantly with problem size:

- **Small datasets (Iris, Wine):** the CUDA implementation is not faster than scikit-learn and only modestly faster than the sequential CPU version. Kernel launch overhead dominates the runtime.
- **Medium datasets (Moons, Circles):** CUDA achieves $\approx 250\times$ speedup over the sequential CPU implementation and $\approx 2.5\times$ speedup over the sklearn implementation.
- **Large dataset (Blobs):** CUDA provides a dramatic speedup of over $400\times$ relative to the sequential CPU implementation and $\approx 2.5\times$ speedup over the sklearn implementation.

4.4 Accuracy Evaluation

We assume that scikit-learn reference solution has 100% clustering accuracy. Across all datasets, the CUDA implementation achieves accuracy equal or very close to the sequential CPU reference. Minor differences arise from non-associative floating-point reductions and the use of shared-memory block reductions. These variations are expected and did not meaningfully degrade clustering quality. In fact, the fused CUDA implementation achieved *higher* accuracy than the sequential CPU implementation on the Blobs dataset, likely due to better numerical stability in

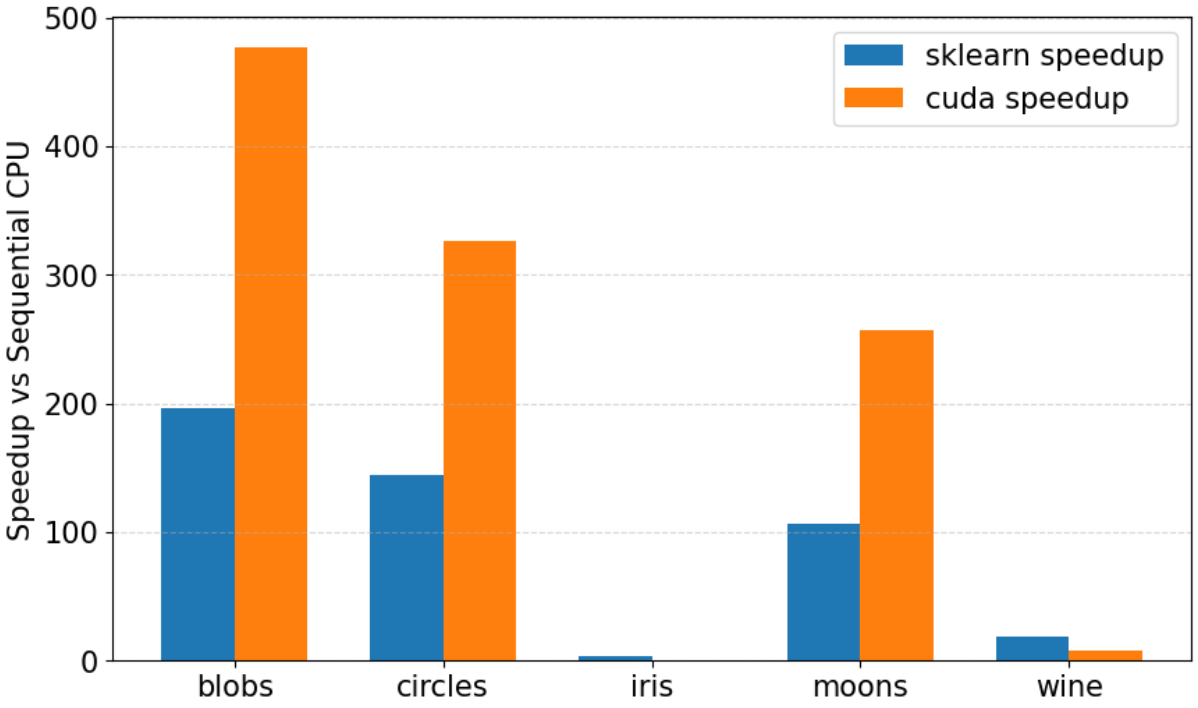

Figure 4: Speedup of sklearn and CUDA Implementations Compared to Sequential Baseline

fused computations. The relatively lower accuracy on the Blobs dataset is expected due to the nature of the dataset, where sklearn also performs differently with different N configured.

4.5 Scalability With Problem Size

To evaluate scaling behavior, we varied N (number of samples) and K (clusters). Our results in Figure 5 and Figure 6 show:

- Runtime grows approximately linearly with N, as expected from the $O(NKD^3)$ E-step.
- Increasing K increases runtime proportionally within the fused kernel, but does not introduce new synchronization bottlenecks.
- The GPU achieves full utilization only when N is large enough to keep thousands of active threads in flight; this is why small datasets show limited benefit.

4.6 Evaluation of Machine Choice

The GPU was an excellent match for the structure of the GMM EM algorithm:

- The E-step is embarrassingly parallel and fully saturates the GPU's SIMT hardware.

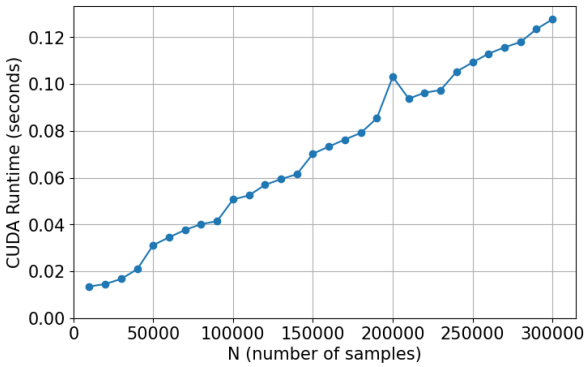

Figure 5: CUDA Runtime vs N for Blobs Dataset

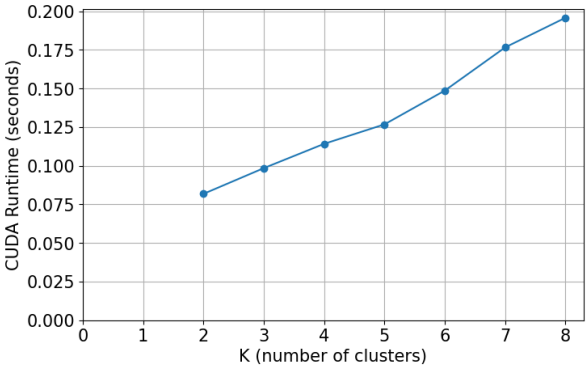

Figure 6: CUDA Runtime vs K for Blobs Dataset

- The hierarchical reductions map naturally onto the GPU's block-level shared-memory model.
- The workload involves high arithmetic intensity and predictable memory access patterns.

A CPU parallelization (e.g., OpenMP, TBB) would reduce runtime considerably relative to the sequential baseline, but would likely not match the order-of-magnitude speedups achieved here due to lower SIMD width, limited memory bandwidth, and fewer hardware threads.

4.7 Summary

Our fused CUDA implementation successfully achieves the goals of the project: it delivers correct results while providing substantial speedups over both a sequential CPU implementation, and over scikit-learn's optimized CPU GMM in large-scale cases. While speedups are modest for small datasets, performance scales extremely well with problem size, confirming that the GPU is the appropriate target architecture for this workload.

5 Conclusion

The core problem we addressed was the excessive time required to train Gaussian Mixture Models due to the $O(N \cdot K \cdot D^3)$ complexity of the EM algorithm, which made it a bottleneck for large datasets. We successfully implemented a highly optimized parallel GMM EM solver on GPUs using CUDA. Our approach used a fused CUDA EM kernel with shared-memory accumulation and multi-stage reductions to effectively overcome the severe contention and serialization caused by simple global atomic operations in the M-step. This approach proved highly successful, achieving a dramatic speedup of over 400x compared to the sequential CPU

baseline on large datasets and running about 3 times faster than the highly optimized scikit-learn implementation. These results confirm that the GPU is the appropriate architecture, as the speedup increases significantly with problem size, validating the scalability of our solution. For future work, we plan to further optimize the remaining bottlenecks, specifically by improving in-block reduction efficiency and fusing the final reduction with parameter finalization.

References

Jeff A Bilmes and 1 others. 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.

Wojciech Kwedlo. 2014. 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*, pages 292–298. IEEE.

Rafael Coimbra Pinto and Paulo Martins Engel. 2015. A fast incremental gaussian mixture model. *PloS one*, 10(10):e0139931.

A Appendix

B Work Distribution

Table 2 summarizes the major tasks completed by each team member, along with the agreed-upon credit distribution for the project.

Student	Contributions	Credit
Clint Zhu	• Implemented sequential CPU baseline and correctness tests • Implemented E-step • Designing block-wise shared-memory reduction • Optimized fused kernel • Wrote the report (Summary, Approach)	50%
Chenghui Yu	• Setup environment, created data loading and GMM pipeline • Integrated CUDA kernels with Python through `pybind11` • Implemented M-step and fused kernel • Built full experimental pipeline and generated evaluation plots • Wrote the report (Background, Results)	50%

Table 2: Summary of project contributions and credit distribution.