GPGPU Final Project: GPGPU Implementation of Expectation-Maximization Algorithm for Gaussian Mixture Models

Bohdan Forostianyi

February 2024

1 Abstract

The project goal was to develop a GPGPU implementation of the Expectation Maximization algorithm for Gaussian Mixture Models. Gaussian Mixture Models can be viewed as a simple linear superposition of Gaussian components, aimed at providing a richer class of density models than the single Gaussian [1]. These models are used for modelling datasets based on observations and have potential applications for more robust clustering, where instead of receiving a binary label of the belonging class, one can expect to have an insight into the probabilities of belonging to the particular class. A simple, yet powerful algorithm for learning underlying Gaussians is called Expectation-Maximization. An algorithm contains a lot of matrix computations and therefore has a big potential for parallelization. This report discusses the design and aims to speed up computations for higher volumes of data. To demonstrate it, the report compares two implementations:

- sequential C++ with *linalg* library [2] used for matrix operations,
- parallelized CUDA implementation.

Both implementations are evaluated for larger and larger problem sizes. Code for the project is available in the repository.

2 Design Methodology

For the General EM Algorithm, the goal is given a joint distribution $p(X, Z|\theta)$ over observed variables X and latent variables Z, governed by parameters θ, the goal is to maximize the likelihood function $p(X|\theta)$ concerning θ [1].

1. Choose an initial setting for the parameters θ
2. E-step: Evaluate $p(Z|X, \theta_{old})$

3. M-step: Evaluate θ_{new} given by:

$$\theta_{new} = argmax\, Q\left(\theta, \theta_{old}\right) \cdot \theta$$

where

$$Q\left(\theta, \theta_{old}\right) = p(Z|X, \theta_{old}) \cdot ln\, p(X, Z|\theta).$$

4. Check for convergence of either the likelihood or the parameter values. If the convergence criterion is not satisfied, then let $\theta_{old} \leftarrow \theta_{new}$ and return to step 2.

In general, this algorithm is rich in three types of operations:

- element-wise summation, power, multiplication or division,
- dot product,
- column or row reduction of the matrix.

Therefore an algorithm parallelization was performed by writing a kernel for each type of operation and replacing sequential operation with its parallel implementation. For simplicity of description, parallelization was done only in the function for the calculation of log probability used in E-step. This function performs a lot of computations and performs each of the aforementioned kernels. Its implementation is introduced in figure 1. However, before proceeding to the analysis it is worth noticing that these kernels can be also used in other parts of the algorithm.

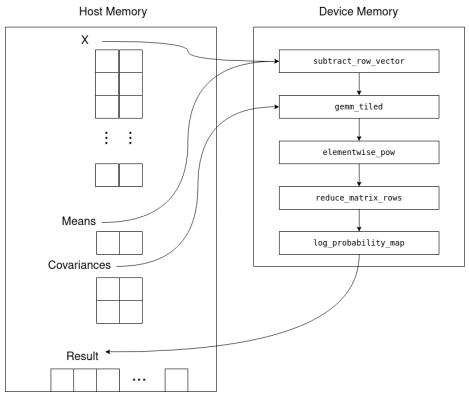

Figure 1: Parallelized log probability calculation

In the beginning, three arrays are stored in host memory, namely

- X of dimensions $N \times D$, where N is the number of rows and D is the number of cols,
- $Means$ of dimensions $1 \times D$,

- *Covariances* of dimensions $D \times D$.

These arrays are consequently transferred to the device memory to perform required calculations. After the first two steps, when all arrays are transferred calculations are performed on outputs from previous steps. In the end, the result is transferred back to the host memory.

Kernels *subtract_row_vector*, *elementwise_pow*, *log_probability_map* are simple elementwise kernels one read (two in case of *subtract_row_vector* kernel) from global device memory and store the result of the operation.

Kernel *gemm_tiled* is used for computation of dot product using tiled matrix multiplication algorithm with *TILE_DIM* equal to 32.

Kernel *reduce_matrix_rows* is the kernel which uses shared memory to reduce all rows along columns in the matrix to a single number. It is presented in the figure 2. Kernel works by loading array elements from global memory to 2D *TILE*. After that, every thread sums the row, based on its *threadIdx.y*, to the single value by iterating over the column in shared memory. In the end, thread with *threadIdx.x == 0* writes output to global memory.

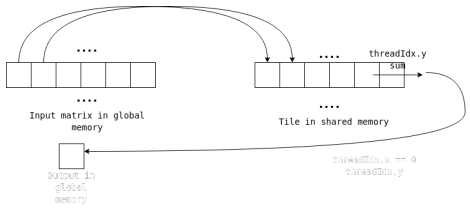

Figure 2: Reduce kernel

3 Results and Data Analysis

	GPU	CPU
OS	Ubuntu 22.04.2 LTS	
CPU	Intel Core Processor (Skylake) @ 2194 MHz	
GPGPU	RTX6000P-8C	N/A
Compiler	Cuda compilation tools, release 11.6, V11.6.124	

Table 1: Evaluation environment

All algorithms were evaluated using the following scenario:

1. set up a known number of iterations that EM will take,
2. generates synthetic data of 2D points with a known number of clusters and the known number of points per cluster. Example data is shown in the figure 3,

3. estimate Gaussian Mixtures parameters and measures time needed for an operation multiple times,

4. increase the number of points per cluster by a power of 2 and return to step 2.

Tests were performed on the university machines. Specification of the environment is provided in table 1.

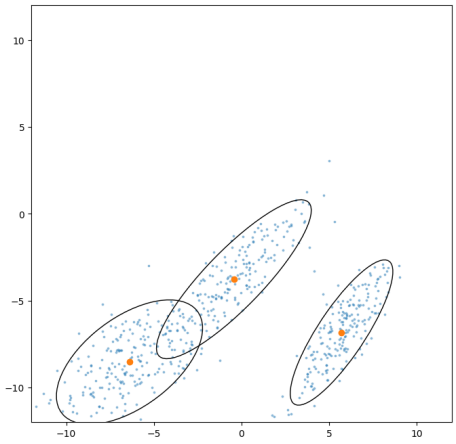

Figure 3: Syntetic data with 200 points per cluster and three clusters together with generated mixture models

Figure 4 presents speedup obtained by dividing the time needed by the sequential implementation for calculating a mixture model for a given data size by the time needed for parallel implementation. Figure 5 shows execution times depending on the problem size. Both figures use box plots which were built based on 5 measurements.

From the figure 4, it is clear, that with these parallelization potential gains will appear only for very large data sizes and from the figure 5 we can see that the obtained gain is equal to multiple seconds (approximately 6%). Therefore more work can be done to improve this result, but at the same time, we are also able to see that even this limited parallelization provides us with gains.

4 Conclusion

Speedup gains are visible only while approaching big volumes of data, so the overhead related to the thread initialization and data transfer to the GPU will be smaller than the potential benefits. In the case of the presented implementation, this data size equals 2^{17} 2D points which should be clustered. However, it should be possible to obtain better results by parallelising other computationally intense parts of the algorithm. An example of such implementation is

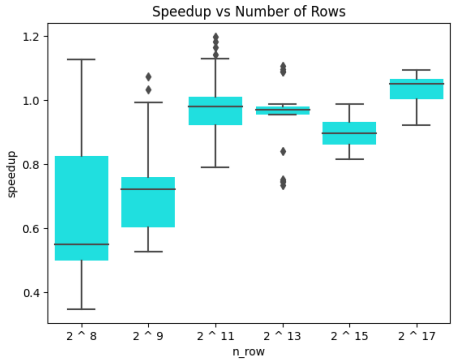

Figure 4: Speedup depending on the problem size

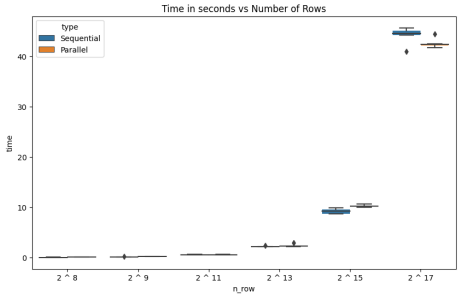

Figure 5: Execution time in seconds depending on the problem size

described in the paper which provided another parallel implementation of the EM algorithm [3].

During the implementation of the project, I learned how to identify parts of the problem which have the potential to be parallelized and how to implement kernels for these parts. I also improved my skills in benchmarking and developed a clear view of the parallelization pipeline which consists of four steps:

- implement proof of concept algorithm in a language such as Python, which will allow you to better understand the algorithm,
- implement sequential version in either C or C++,
- parallelize sequential version by slowly replacing parts of sequential code with kernels and checking algorithm results after every step.

References

[1] Christopher Bishop. Pattern recognition and machine learning. *Springer google schola*, 2:5–43, 2006.

[2] Matthew Drury. linalg: Linear algebra and regression in c. `https://github.com/madrury/linalg`, December, 2023.

[3] Hiroki Nishimoto, Renyuan Zhang, and Yasuhiko Nakashima. Gpgpu implementation of variational bayesian gaussian mixture models. *IEICE TRANSACTIONS on Information and Systems*, 105(3):611–622, 2022.