Accelerating your Python Code

For GMMs with PyCUDA

Varun Nayyar

27/07/18

Outline

Me = Math Major + Script Kiddy (Manage Expectations)

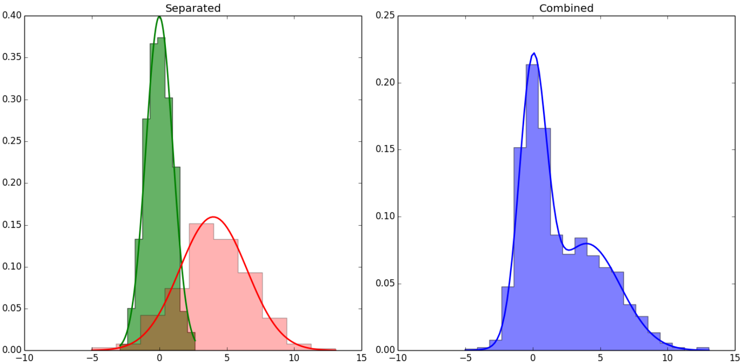
What to Expect

- Some Math
- Thinking with CUDA and Basic Syntax
- How to use PyCUDA to avoid complicated work

What NOT to Expect

- How PyCUDA does it's magic
- Intermediate/Advanced CUDA

Gaussian Mixture Models d=1, K=2



K-means+=1

Gaussian Mixture Models (GMMs)

Density Function

For K mixtures

$$f(\mathbf{x}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)$$

Log Likelihood (function of concern)

$$l(\mu, \Sigma, \mathbf{x}) = \sum_{i=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k) \right)$$

Need for speed

Some post-hoc realizations

- GMM likelihood formula doesn't decompose into a mathematically simple form
- However, note that the GMM likelihood has a parallelizable form in that each point of each mixture is independent (CUDA vibes)

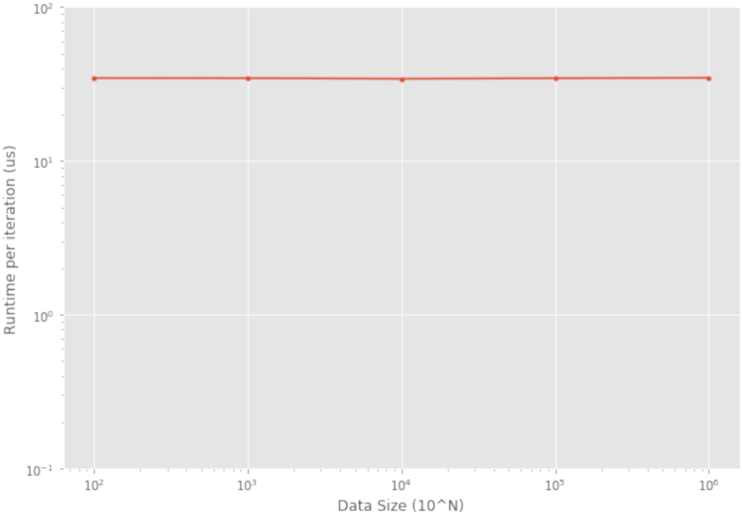
Computational Numbers

- Number of flops are of the order of $O(NKd)$, in my case, $N = 10^6$, $K = 8$, $d = 13$. I.e. $O(10^8)$
- I needed to evaluate the likelihood 10^6 times for a fixed dataset while the parameters were varied. (Markov Chain Monte Carlo)
- i.e 10^{14} floating point operations per run.

First Attempt

- Eh, my computer is fast enough
- Pure Python (numpy)
- Took 36 us per datapoint, or 36s for whole dataset
- 10^6 evaluations would take 1 year

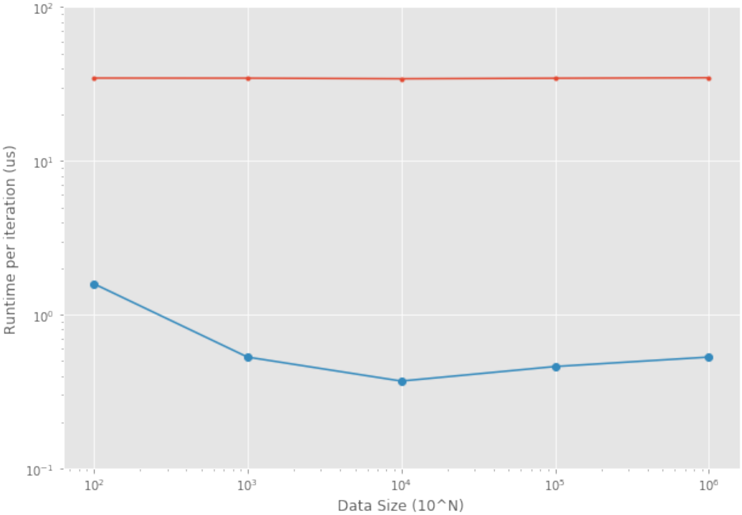
Execution speed per N



Second Attempt

- Stand on the shoulders of giants (scikit-learn)
- Reverse engineered the likelihood evaluator
- 72x improvement!
- Took 0.5 us per datapoint, or 0.5s for whole dataset
- 10^6 evaluations would take 5 days!

Execution speed per N



CUDA

- Stood for Compute Unified Device Architecture but no one cares so this was forgotten
- CUDA devices have Streaming Multiprocessors (SMs) and each has a number of CUDA cores. CUDA runs threads in batches of 32 called Warps. GTX970 has 13 SMs with 128 CUDA cores each.
- in the CUDA paradigm, you need plenty of independent threads to take advantage of the architecture and to minimize memory latency via async scheduling.
- Not many guarantees of all threads running exactly in parallel, so code still needs to be thread safe.
- number of threads is magnitudes greater than in standard multicore programming.

Thinking with CUDA (Matrix Multiplication)

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & & \ddots & \\ a_{n1} & & & a_{nn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ \vdots & & \ddots & \\ b_{n1} & & & b \end{pmatrix}$$

- Each element on output matrix requires n multiplications and additions. n^2 elements.
- Single Core $= O(n^3)$, though tricks allow for $O(n^{2.81})$
- CUDA??

Thinking with CUDA (Matrix Multiplication)

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & & \ddots & \\ a_{n1} & & & a_{nn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ \vdots & & \ddots & \\ b_{n1} & & & b \end{pmatrix}$$

- Each element on output matrix requires n multiplications and additions. n^2 elements.
- Single Core $= O(n^3)$, though tricks allow for $O(n^{2.81})$
- Theoretical CUDA $= O(n)$

Thinking with CUDA (Matrix Multiplication)

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & & \ddots & \\ a_{n1} & & & a_{nn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ \vdots & & \ddots & \\ b_{n1} & & & b \end{pmatrix}$$

- Each element on output matrix requires n multiplications and additions. n^2 elements.
- Single Core $= O(n^3)$, though tricks allow for $O(n^{2.81})$

- Theoretical CUDA $= O(n)$
- Create n^2 threads. Each takes $O(n)$ time and can be run in parallel

Thinking with CUDA (Summing an Array)

Sum an array $\begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix}$

- Single Core $= O(n)$
- CUDA ??

Thinking with CUDA (Summing an Array)

Sum an array $\begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix}$

- Single Core $= O(n)$

- theoretical CUDA $= O(lg(n))$

Thinking with CUDA (Summing an Array)

Sum an array $\begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix}$

- Single Core $= O(n)$

- theoretical CUDA $= O(lg(n))$
- First pass, have $N/2$ threads add pos i and pos $i + N/2$
- Second pass, have $N/4$ threads add pos i and pos $i + N/4$