Accelerating your Python Code

For GMMs with PyCUDA

Varun Nayyar

27/07/18

Outline

Me = Math Major + Script Kiddy (Manage Expectations)

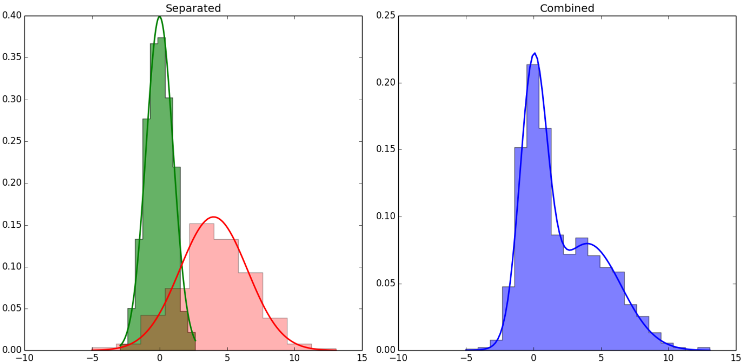
What to Expect

- 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)

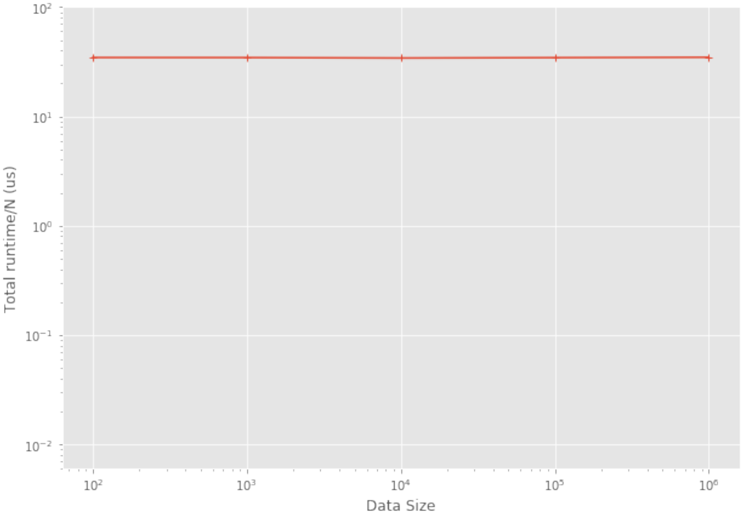
Computational Numbers

- 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)
- Averaged 36 us per datapoint, or 36s for whole dataset
- 10^6 evaluations would take 1 year

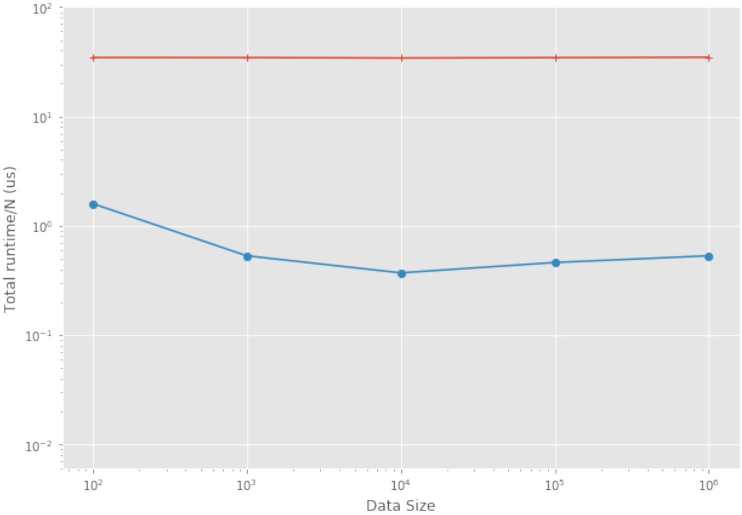
Execution speed per N



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



Aside: Timing Methodology and Testing

Ran on Ubuntu 18.04 - i3, 8GB RAM, >3 years old

- fixed $K = 8$ and $d = 13$ as per live behaviour. Varied N from 10^2 to 10^6 for completeness. 10^6 is intended data size.
- Used `timeit` module with 100 iterations and report $\text{totaltime}/100 * N$

Testing

- Manually compared my hand written version to R implementation using output
- Used `pytest` to create random tests to check that random data and parameters matched values.

It's good to be rigorous - Math Major

CUDA

- Stood for Compute Unified Device Architecture but no one cared 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 $i + N/2$

Thinking with CUDA (Summing an Array)

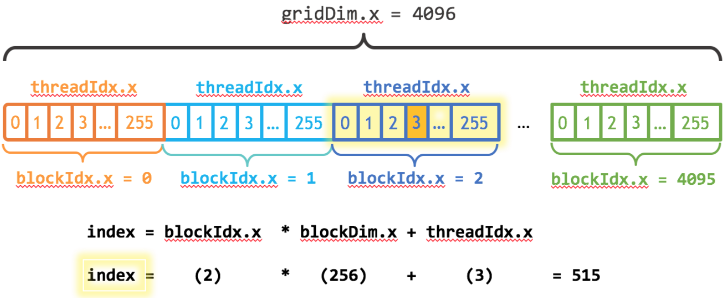
Sum an array $\begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix}$

- 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 $i + N/4$
- Etcetera..

CUDA Architecture

A grid of blocks of threads.



```
index = blockIdx.x   * blockDim.x + threadIdx.x

index =     (2)       *   (256)    +     (3)        = 515
```

Could be 3d if necessary (We're gonna forget that fact)

CUDA Architecture

- Main function is called a kernel
- Each block runs 32 threads at a time (called warps) and shares a limited amount of memory available to all threads. Keep Blocks in multiples of 32 threads.
- GTX 970 = 128 cores * 13 SMs = 52 concurrent warps. Most GPUs have more 32 bit processing power (float32)
- Generally, Blocks are completed as quickly as possible
- Threads act asynchronously - when they access data, threads give up resources while waiting for data.
- Memory access and data transfers (especially between host and device) have a lot of tricks and patterns to optimize (none of which I'm going to investigate)

Some CUDA Syntax (C++)

- Specify the kernel function with the `__global__` keyword.
- Specify GPU function `__device__` keyword. (As opposed to CPU function, .cu files can be compiled with `nvcc`)
- share memory/data across with `__shared__` keyword. Only available within block
- pause thread execution till all threads have reached same point `__syncthreads()`. Async-esque
- `int index = blockIdx.x * blockDim.x + threadIdx.x` is considered idiomatic cuda for indexing global arrays
- `int stride = blockDim.x * gridDim.x;` is used in situations there are fewer threads than data points.

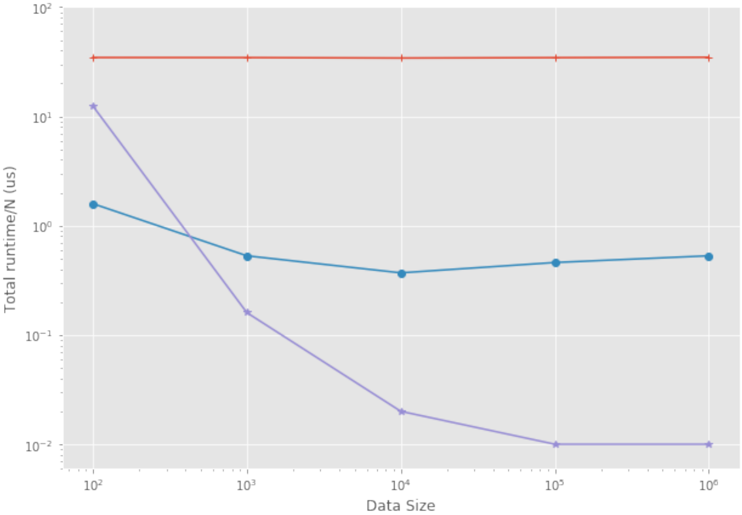
PyCUDA

- CUDA is complex. C++ is messy. I'm a maths major
- The mathematics (parameter variation) needed to be done in Python for quick implementation and easy refinement.
- Allows for easier kernel writing since you can avoid mallocs for memory management and use static sized arrays
- PyCUDA abstracts a lot of the process away, at the cost of fine tuned optimization
- Some investigation needed for example 2 dim numpy arrays were flattened to 1d on the device
- fast enough was what I needed.

CUDA Results

- CUDA Impletation
- Minimal optimization
- 50x improvement over sklearn
- Took 0.01 us per datapoint, or 0.01s for whole dataset
- 10^6 evaluations would take 2.5 hours!

Execution speed per N



Thanks

Code available at `https://github.com/nayyarv/PyCudaIntro`

Timing Data (100 iterations)

N	Simple	Sklearn	GPU
100	0.35	0.02	0.12
1000	3.45	0.05	0.02
10000	34.22	0.37	0.02
100000	344.93	4.6	0.13
1000000	3474.57	53.39	1.07

CUDA and PyCUDA easily installed via apt on Ubuntu 18.04.
Project was in Speech Recognition and Bayesian Statistics (MCMC)