CME 253 Project:
Estimating Latent Parameters with Cuda

Jacob Perricone
Stanford University
jacobp2@stanford.edu

Introduction

My project is a continuation of work I conducted in MS&E448, during which I analyzed the Consumer Financial Protection Bureaus (CFPB) consumer complaint database, leveraging both the data's structured fields and unstructured complaint narratives. In order to gather more insight from the raw complaints, we developed a probabilistic defect model to extract latent defects from the complaint data. The motivation for using a probabilistic latent defect model is that it is able to extract underlying defects that span across both the unstructured data, the complaint narrative, and the structured fields. The primary limitation of the project was that it took an inordinate amount of time for the expectation-maximization algorithm to estimate the latent parameters of the model. My project is thus is to utilize GPU's parallel processing capabilities to speed up the algorithm. Although there is much more work I would like to do on this, the speed up from cuda is awesome, processing a complaint set of $\approx$ 13000, with $\approx$ 7000 words on the order of seconds, rather than hours.

The technical details of the model as well as what it aims to achieve can be investigated further in the document poster.pdf. For our purposes, it suffices to say my project an application of the expectation-maximization algorithm. A generative model is assumed to generate the entities within a complaint (company, product, issue, and narrative), and are modeled as independent multinomials. Specifically: We assume that each complaint record is generated from a latent distribution of defects, and we set out to model the joint distribution of complaints and defects. For each complaint record, $x^{(i)} : x \in \mathbf{C}$, we aim to model the joint distribution $p(x^{(i)}, d^{(i)}) = p(x^{(i)}|d^{(i)})p(d^{(i)})$ where $d^{(\mathrm{i})} \sim$ Multinomial(ϕ) (where $\phi_j > 0, \sum_{j=1}^{|\mathbf{D}|} \phi_j = 1$) with the parameter ϕ_j specifying $p(d^{(\mathrm{i})} = j)$. The E-M algorithm proceeds as follows to maximize the lower bound of the log-likelihood:

$$\mathcal{L}(X; \Phi, \phi) = \sum_{i=1}^{m} \log \big[\sum_{j=1}^{k} p(x^{(\mathrm{i})}|d^{(\mathrm{i})}; \Phi)p(d^{(\mathrm{i})}; \phi) \big]$$

1: Complaint Generation

1: Initialize the complaint entity posteriors randomly
2: **Until Convergence**
3: For all complaints i and defects j
   (i) Expectation Step (E-STEP): Calculate the posterior $p(d^{(\mathrm{i})} = j|x^{(\mathrm{i})})$ using Baye's rule and smoothing techniques to prevent numerical underflow.
4: Calculate the lower bound of the log likelihood using the defect posterior estimate obtained from the E-Step
5: For all complaints i and defects j
   (i) Maximization Step (M-STEP): re-estimate the posteriors density using the defect posteriors obtained in the E-Step

The E-STEP and M-STEP were parallelized and executed on a GPU. The M-STEP was implemented with both a CUDA Kernel and using the cuBLAS library. The highly optimized cuBLAS library is roughly twice as fast as the kernel implementation. I will go through each

1. `update_entities` This function has the highest priority of the kernels employed throughout the algorithm with a rank of 100. The main functionality of the update entities is to perform the M-Step of the algorithm for each of the different entities. To simplify the explanation, let's say there are d defects,N complaints and V words in the vocabulary. The defect posterior matrix P_d are a $d \times N$ matrix corresponding to the $P(d_j|\text{complaint i})$, For each entity E, let $\bar{e}$ be the number of distinct values an entity can take on. Let E_D be $N \times \bar{e}$ matrix that has a one in (i, j) if complaint i has entity realization e_i and zero otherwise (very sparse). Furthermore let $\div$ be the element wise row division operator— that is, if A is $m \times n$ and x is $m \times 1$, $B = A \div x$. Is $m \times n$ where each element in row i of B has been divided by x_i To update the $d \times \bar{e}$ matrix corresponding to $P[e|d_j]$ we must perform the operation $P_{ej} = (\mathbf{1} + P_d E_D) \div (\bar{e} + \sum_{i=1}^{n} P_d^{ji})$.
   My first approach was to do the matrix operations separately using the tiled block algorithm discussed in class but expanding for non-square arrays. This function is implemented in `mat_mul`. Then a separate kernel `elementwise_division` would perform the division operations. This seemed inefficient so I combine the two steps in `update_entities` is very similar except that it also adds the numerator and denominator constants $(1, \bar{e})$ in the example above. Examining the profiler, it appears that the performance is limited by compute resources. I have 12.6% divergence for updating the word posteriors and 15.6% for the others, which is bad. On average it took around $.008$ seconds, which is a huge improvement over the CPU. It achieves an occupancy of 94% for the word posteriors, indicating that the kernel has enough work to execute and enough warps active on the GPU. I believe that the warp divergence occurs since the matrix dimensions are so vastly difference. The number of defects is typically around $3 - 5$, with the the vocab size and complaint number being in the thousands. Thus, many threads are outside the bounds of one of the first matrix and do nothing. To improve this, in theory one could create non-square tile's, and slide them along the matrices, but I did not know exactly how one would do this.
2. `eStep` eStep was my first take at a kernel that completes the expectation segment of the algorithm by launching blocks of size (number of defects, 256).I believe this first design decision was my biggest flaw and given the time constraints, I was unable to take a step back and reformulate to a block, grid schema that yielded better results. Nevertheless, on average it takes 4 milliseconds to complete, which two orders of magnitudes times faster than the CPU. It achieves an occupancy of 50% out of a theoretical 75%. Profiling the function it appears that I may increase the number of warps on each SM by increasing the number of thread between each block. However since I am limited by 1026 threads, increasing the threads per blocks didn't yield significant improvements (this could be ameliorated with a different block grid construction). The kernel also contains a gross amount of logic for the smoothing steps, which I tried to get rid of, but my modifications often yielded incorrect posteriors, and so I gave up. Although I store many variables to shared memory, I believe global memory access is also an issue. Particularly in calculating the product of defect posteriors and term frequency where I have to loop through all words in the vocabulary. The shared memory bandwidth is $20GB/s$ and the global memory bandwidth is $8GB/s$, which can be improved.
3. `eStep2` An iteration of eStep where I try to increase shared memory usage. The time of execution drops nearly 50%, with an average duration of 2.9 ms. Doing so, however, makes results in terms occupancy, with a percentage 47.75% out of 75% theoretical. The memory bandwidth improves, though, with global memory bandwidth $11.1GB/s$ and shared memory bandwidth $30.5GB/s$.
4. `reduce_columns` This function sums across columns of an array returning a column vector of the sums. It uses a block-stride approach to load the data in shared memory and is a two dimensional version of the reduction learned in class. I launch a block per row, which limits the amount of complaints to ≈ 65000. The occupancy is 98%, which is very nearly optimal. However, since it does not load into shared memory, load/store operations are the main limitations.

Ideally I would make the code more flexible, but now there are three files, headers.h main.cu and auxiliary.cu. The headers file outlines the functions. I am restricting the dataset I send over to the small files, since I did not allow for command line arguments for altering those parameters and thus changing the data requires manually tuning of the code. The complete dataset is around 100,000 complaints. I aim to keep working on this to be able to process the whole dataset the future. [1]

[1] Thank you for everything this quarter. The compute capabilities of GPUs are remarkable. I hope to get better at this