Efficient Training of LDA on a GPU by Mean-for-Mode Estimation

Jean-Baptiste Tristan JEAN.BAPTISTE.TRISTAN@ORACLE.COM
Oracle Labs, USA

Joseph Tassarotti JTASSARO@CS.CMU.EDU
Department of Computer Science, Carnegie Mellon University, USA

Guy L. Steele Jr. GUY.STEELE@ORACLE.COM
Oracle Labs, USA

Abstract

We introduce Mean-for-Mode estimation, a variant of an uncollapsed Gibbs sampler that we use to train LDA on a GPU. The algorithm combines benefits of both uncollapsed and collapsed Gibbs samplers. Like a collapsed Gibbs sampler — and unlike an uncollapsed Gibbs sampler — it has good statistical performance, and can use sampling complexity reduction techniques such as sparsity. Meanwhile, like an uncollapsed Gibbs sampler — and unlike a collapsed Gibbs sampler — it is embarrassingly parallel, and can use approximate counters.

1. Introduction

The performance of GPUs makes them an appealing choice for machine learning. However, there are several challenges in using them efficiently. First, there must be enough parallelism to make use of all the many thousands of GPU cores. Second, GPUs have small amounts of memory, and with enough parallelism, memory bandwidth becomes a bottleneck. Finally, it is often unclear how to translate techniques that have been developed for CPUs into improvements for GPUs.

Topic models, such as LDA (Blei et al., 2003), are a good example of these challenges of using GPUs. Although there are several methods that can be used to train LDA (Asuncion et al., 2009), earlier work targeting the GPU (Yan et al., 2009; Lu et al., 2013) has primarily used collapsed Gibbs sampling (Griffiths & Steyvers, 2004) and adapted techniques from distributed implementations of LDA (Newman et al., 2009).

Proceedings of the 32^{nd} International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37.

Since collapsed Gibbs sampling is inherently sequential, this earlier work has tried to adapt collapsed sampling to recover some parallelism. However, as the number of cores on GPUs continues to grow, it becomes more and more difficult to extract enough parallelism to scale effectively. In addition, the amount of memory required for collapsed Gibbs sampling and its memory access patterns can limit the amount of data that can be processed at a time. Furthermore, techniques that have been developed to reduce sampling complexity, such as working with sparse matrices (Yao et al., 2009), are difficult to adapt to GPUs.

In contrast, a standard uncollapsed Gibbs sampler is embarrassingly parallel. Tristan et al. (2014) show that a GPU implementation of uncollapsed Gibbs sampling for LDA can scale to a large number of GPU cores. However, Newman et al. (2009) have shown that the statistical performance of an uncollapsed Gibbs sampler is not as good as that of the collapsed Gibbs sampler: it requires more iterations to converge, and it generates estimates that are not as good. Moreover, as pointed out by Smola & Narayanamurthy (2010), the standard uncollapsed sampler cannot use the sparse representations that have been developed for collapsed samplers.

In this paper, we present Mean-for-Mode estimation, a modification of the uncollapsed Gibbs sampler that is related to stochastic expectation maximization as presented by Celeux et al. (1995). In §2 we describe the algorithm and empirically demonstrate that its convergence rate is similar to that of the collapsed Gibbs sampler. Next, we describe a simple GPU implementation in §3 and show that there is more than enough parallelism to scale effectively. One important difference between our work and previous work here is that instead of adapting popular inference methods for LDA that were implicitly designed for a CPU architecture, *we rethink the inference method in light of the needs of the GPU architecture*.

A key benefit of Mean-for-Mode estimation is that *it enables the use of several techniques that are key to scaling to larger datasets and managing memory issues*. First, Mean-for-Mode estimation can use single-precision floating points to store parameter values, and does not need to store the latent topic-assignment variables. Since there are as many latent variables in the model as there are tokens in the corpus, this is a very significant reduction in memory usage. In addition, Mean-for-Mode estimation can use approximate counters (Morris, 1978) to decrease memory use (§4.1). Next, we show that unlike traditional uncollapsed Gibbs samplers, it can also make use of sparsity (§5). In comparison to earlier work that tried to use sparsity with collapsed Gibbs samplers on GPUs, the implementation is straightforward. We believe these techniques are applicable to any large mixture models.

2. Mean-for-Mode Estimation for LDA

Before describing Mean-for-Mode estimation, we briefly review the LDA model. LDA is a categorical mixture model for text documents with one set of mixing coefficients per document, with the components shared across the corpus. It is a Bayesian model in which both the mixing coefficients and the components are given a Dirichlet prior. The distribution of LDA is described in Figure 1, and the reader can refer to the introduction written by Blei (2012) for more details about topic modeling and LDA. Note that we reserve the word "latent" only for the topic-assignment variables (z_{ij}), not the parameters of the model.

In order to effectively train LDA on the GPU, we need an inference method that is embarrassingly parallel. For example, to make full use of a modern NVIDIA GPU, it is desirable to have an application with tens of thousands of threads, perhaps even millions. Unfortunately, the collapsed Gibbs sampler is a sequential algorithm. Indeed, what makes it a good algorithm is that integrating the parameters of LDA makes the latent variables directly dependent on each other. An obvious way to devise a highly parallel sampler is to not integrate the parameters before deriving the Gibbs sampler, thereby using an uncollapsed Gibbs sampler. In this case, the algorithm will sample not only the latent variables, but also the parameters of the model ($\boldsymbol{\phi}$ and $\boldsymbol{\theta}$). However, as noted by others (Newman et al., 2009), using such an uncollapsed Gibbs sampler for LDA requires more iterations to converge.

To address this problem, Mean-for-Mode estimation uses a point estimate of the ϕ and θ parameters instead of sampling them. The algorithm is shown in Figure 2. First, we draw the parameters from the prior.[1] Inside the main loop that generates samples, we first draw all of the latent variables given the parameters, as we would with an uncollapsed Gibbs sampler for LDA. Then, we "simulate" the parameters by assigning to them the mean of their distribution conditioned on all other variables of the model.

Why use the mean? A different choice for a point estimate would be to use the mode of the conditional distribution. However, these conditional distributions are Dirichlet distributions, which do not necessarily have a mode. In contrast, the mean of a Dirichlet distribution is always defined, and in practice it results in good statistical performance. Moreover, note that if $X \sim \text{dir}(\alpha)$ where X is a random vector of size K, then $\mathbb{E}[X_i] = \frac{\alpha_i}{\sum_k \alpha_i}$, which is equal to $\frac{(\alpha_i+1)-1}{(\sum_k \alpha_i+1)-K}$ which is the mode of the Dirichlet distribution $\text{dir}(\alpha+1)$. This means that Mean-for-Mode estimation is an instance of stochastic expectation maximization (Celeux et al., 1995), and consequently it has an equilibrium.

A different way to think about the algorithm is with respect to the collapsed Gibbs sampler. In the collapsed Gibbs sampler, we draw the latent variables with the parameters integrated out:

$$p(\boldsymbol{z}|\boldsymbol{w}) = \int_{\boldsymbol{\theta}} \int_{\boldsymbol{\phi}} p(\boldsymbol{z}|\boldsymbol{\theta}, \boldsymbol{\phi}, \boldsymbol{w}) p(\boldsymbol{\theta}) p(\boldsymbol{\phi}) d\boldsymbol{\phi} d\boldsymbol{\theta}$$

The Mean-for-Mode algorithm corresponds to a plug-in approximation (Murphy, 2012) of the above equation using the estimates $\mathbb{E}[\boldsymbol{\theta}]$ and $\mathbb{E}[\boldsymbol{\phi}]$ instead of the MAP.

$$p(\boldsymbol{z}|\boldsymbol{w}) \approx p(\boldsymbol{z}|\mathbb{E}[\boldsymbol{\theta}], \mathbb{E}[\boldsymbol{\phi}], \boldsymbol{w})$$

Using the posterior mean as opposed to the MAP is common to avoid overfitting.

We do not claim that this algorithm is sophisticated or very original: many inference algorithms are modifications of expectation-maximization or Gibbs sampling that use both stochastic simulation and point-estimation (*e.g.*, Monte-Carlo Expectation maximization, iterated conditional modes, greedy Gibbs sampling (Bishop, 2006)). Rather, our contribution is in noting that this point in the design space is a sweet spot for GPU implementations (and possibly distributed implementations as well): it is embarrassingly parallel, while still allowing the use of space-saving optimizations such as sparsity and approximate counters. Although our focus in this paper is on LDA, these optimizations could be used for other mixture models by carefully using point estimates for the parameters.

We have run a series of experiments which show that in practice, Mean-for-Mode estimation converges in fewer samples than standard uncollapsed Gibbs sampling. In these experiments, we observed how the log-likelihood of LDA evolves with the number of samples. Figure 3 presents the results of one of our experiments, run on a subset of Wikipedia

[1] For this presentation, we use this simple initialization. There are other, possibly better, initializations we could use (Wallach et al., 2009), but this is independent of the key ideas of our algorithm.

$$p(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{\theta}, \boldsymbol{\phi}) = \left[\prod_{i=1}^{M}\prod_{j=1}^{N_i} \text{cat}(w_{ij}|\phi_{z_{ij}})\ \text{cat}(z_{ij}|\theta_i)\right]\left[\prod_{i=1}^{M} \text{dir}(\theta_i|\alpha)\right]\left[\prod_{i=1}^{K} \text{dir}(\phi_i|\beta)\right]$$

Figure 1. Mixed density for the LDA model. M is the number of documents, N_i is the size of document i, K is the number of topics, w_{ij} is the j^{th} word in document i, z_{ij} is the topic associated to w_{ij}, θ_i is the distribution of topics in document i, ϕ_k is the distribution of vocabulary words in topic k . cat and dir refer respectively to the probability mass function of the Categorical distribution and the probability density function of the Dirichlet distribution. The variables $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ can be integrated analytically; doing so leads to the collapsed form of LDA, from which we can derive the effective collapsed Gibbs sampler.

1. Initialize
   - Sample $\boldsymbol{\theta}^{(0)} \sim \text{Dir}(\alpha)$
   - Sample $\boldsymbol{\phi}^{(0)} \sim \text{Dir}(\beta)$
2. For $\tau = 1, ..., T$
   - Sample $\boldsymbol{z}^{(\tau+1)} \sim p(\boldsymbol{z}|\boldsymbol{\theta}^{(\tau)}, \boldsymbol{\phi}^{(\tau)}, \boldsymbol{w})$
   - Evaluate $\boldsymbol{\theta}^{(\tau+1)}$ and $\boldsymbol{\phi}^{(\tau+1)}$ given by

$$\boldsymbol{\theta}_i^{(\tau+1)} = \mathbb{E}[\boldsymbol{\theta}_i|\boldsymbol{z}^{(\tau+1)}]$$
$$\boldsymbol{\phi}_i^{(\tau+1)} = \mathbb{E}[\boldsymbol{\phi}_i|\boldsymbol{z}^{(\tau+1)}]$$

Figure 2. Mean-for-Mode estimation for LDA producing $T + 1$ samples. It is similar to an uncollapsed Gibbs sampler, but the stochastic steps that sample the parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are replaced with point estimates. If we were to use the mode as a point estimate, this algorithm would be a stochastic expectation maximization. However, because the mode of each distribution of interest may not exist, we use the conditional mean as a point estimate. Note that for simplicity, the equations are presented modulo conditional independence.

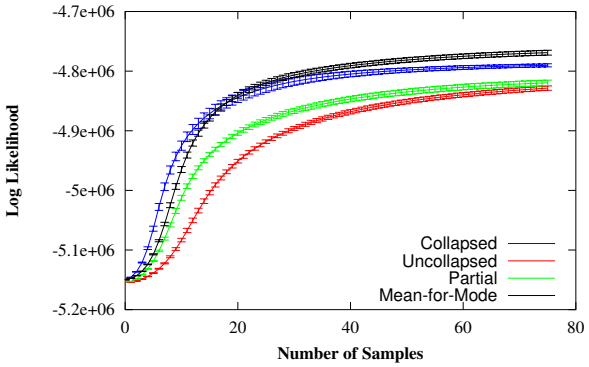

Figure 3. Comparison of the statistical performance of four samplers for LDA: collapsed Gibbs sampler, uncollapsed Gibbs sampler, Gibbs sampler with only the $\boldsymbol{\theta}$ parameters collapsed, and the Mean-for-Mode estimation. This graph shows the result of 10 runs on a Wikipedia subset (50,000 documents, 3,000,000 tokens, 40,000 vocabulary words) using 20 topics.

(50,000 documents, 3,000,000 tokens, 40,000 vocabulary words) with 20 topics, and both α and β equal to 0.1. Unless otherwise noted, this is the dataset and configuration we use in the remainder of the paper. The experiment was run 10 times with a varying seed for the random number generator. We drew over 70 samples with each method. Note that we have also run many more experiments, by varying the values of the hyperparameters (36 combinations taken from $\{0.01, 0.1, 0.25, 0.5, 0.75, 1\}$ for both α and β), and by varying the number of documents (from 5,000 to 50,000) and the number of topics (from 20 to 1000), but we present only one here. The result shown here is consistent with our other experiments and the conclusions we draw in the next paragraph.

The graph confirms the experiments presented by Newman et al. (2009). The uncollapsed Gibbs sampler does indeed converge significantly more slowly than the collapsed Gibbs sampler. Also, as they noted, trying to improve the convergence while keeping a lot of the parallelism by integrating only $\boldsymbol{\theta}$ does not improve the situation much. However, the Mean-for-Mode estimation seems to work well, with a convergence rate closer to that of the collapsed Gibbs sampler. In this specific experiment its log likelihood exceeds that of the collapsed Gibbs sampler after 20 iterations. Note that we have consistently examined the quality of the topics, and the topics seemed to be of the same quality as the ones of the collapsed Gibbs sampler (and most often, the topics are very similar).

3. Parallel GPU Implementation

The Mean-for-Mode estimation applied to LDA exposes a lot of fine-grained parallelism and enables effective GPU implementations. In this section, we describe a basic implementation, which is mostly straightforward. We also show how to refine this with space-saving optimizations such as approximate counters and sparse representations.

We store the data and state of the algorithm using the fol-

Algorithm 1 Drawing the latent variables

memclear `wpt`
memclear `tpd`
memclear `wt`
for $m = 0$ **to** $M - 1$ **in parallel do**
  float p[K]
  for $i = 0$ **to** $N - 1$ **do**
    float sum = 0;
    int c_word = words[m][i]
    for $j = 0$ **to** $K - 1$ **do**
      sum += phis[j][c_word] * thetas[m][j]
      p[j] = sum
    end for
    float stop = uniform() * sum;
    for $j = 0$ **to** $K - 1$ **do**
      if $stop < p[j]$ **then**
        break
      end if
    end for
    atomic increment `wpt`$[j][c\_word]$
    atomic increment `wt`$[j]$
    increment `tpd`$[m][j]$
  end for
end for

Algorithm 2 Estimation of the ϕ variables

for $v = 0$ **to** $V - 1$ **in parallel do**
  for $k = 0$ **to** $K - 1$ **in parallel do**
    `phis`$[k][v]$ = (`wpt`$[k][v]$ + β) / (`wt`$[k]$ + βV)
  end for
end for

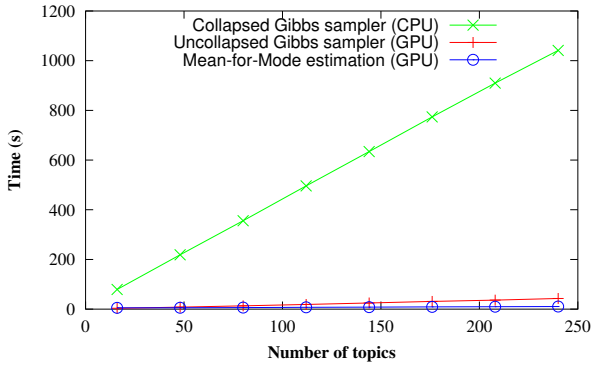

Figure 4. Time to draw 100 samples for 50,000 documents as a function of the number of topics. The uncollapsed Gibbs sampler and the Mean-for-Mode estimation benefit from the large number of parallel units of the GPU when we increase the number of topics. They have similar runtime performance, with the Mean-for-Mode estimation being faster.

lowing arrays and matrices (where $\mathbb{I}$ denotes integers and $\mathbb{F}$ denotes floating-points): `words` $\in \mathbb{I}^{M \times N}$; `phis` $\in \mathbb{F}^{K \times V}$; `thetas` $\in \mathbb{F}^{M \times K}$; `tpd` $\in \mathbb{I}^{M \times K}$ (topics per document); `wpt` $\in \mathbb{I}^{K \times V}$ (words per topic); `wt` $\in \mathbb{I}^{K}$ (total words per topic).

Algorithm 1 shows how we sample the latent variables. Note that for each document, we launch a separate thread that samples topics for every word in that document. For the corpuses we are interested in, the number of documents ranges from tens of thousands to millions, so there is more than enough parallelism.

As the threads select topics for each token in their documents, they update the `wpt`, `wt`, and `tpd` matrices. These keep count of how many times each word has been assigned to a particular topic, how many times each topic has been assigned throughout the corpus, and how many times each topic appears in each document. These counts are used to estimate ϕ and θ. Algorithm 2 shows how $\mathbb{E}[\boldsymbol{\phi}]$ is computed by launching a thread for each entry ($\mathbb{E}[\boldsymbol{\theta}]$ is computed similarly).

We implemented this algorithm on an NVIDIA Titan Black, as well as the uncollapsed Gibbs sampler. We also implemented a collapsed Gibbs sampler for comparison, on an Intel i7-4820K CPU. We present the resulting benchmarks in Figures 4 and 5 to show how the gap between the GPU algorithms' runtimes and that of a collapsed Gibbs sampler scales. Moreover, note that the GPU algorithms scale with increased number of cores. As an example, while switching from an NVIDIA GeForce 640 (384 cores) to a Titan (2688 cores), we observed a $10\times$ speed-up on the uncollapsed Gibbs sampler.[2] Note that the sequential sampler here is not the most efficient CPU implementation of collapsed Gibbs sampling, such as Fast-LDA (Porteous et al., 2008) or SparseLDA (Yao et al., 2009). However, our point is that an implementation of Mean-for-Mode estimation for GPUs that has not been overly tailored to LDA has good complexity and scales well, and the gap in runtime compared with that of a collapsed Gibbs sampler increases linearly.

4. Improving Memory Usage

As we suggested in the introduction, after exposing enough parallelism, the next barriers to performance are related to memory use. Given the degree of parallelism of the algorithm and the hardware, it is best to process as much data as possible on one GPU. In addition to reducing memory footprint, we can improve performance by reducing the amount of memory accesses, thereby improving memory bandwidth

[2]Not all of this performance gap is due to number of cores. The GeForce 640 is based on the earlier Kepler chipset and has different memory clockrate and bandwdith.

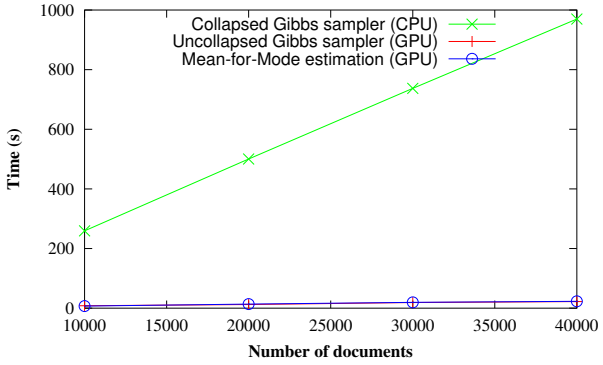

Figure 5. Time to draw 100 samples for 20 topics as a function of the number of documents. The uncollapsed Gibbs sampler and the Mean-for-Mode estimation benefit from the large number of parallel units of the GPU when we increase the number of documents.

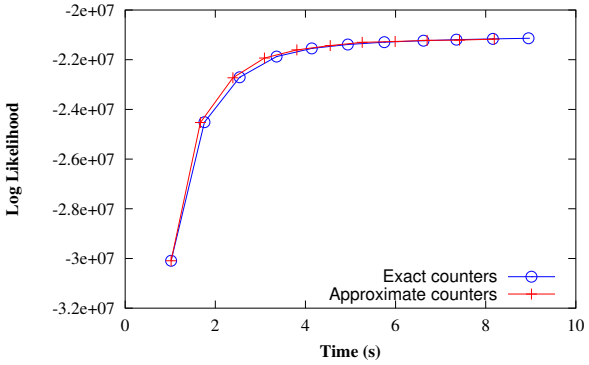

Figure 6. Evolution of log likelihood over time for the Mean-for-Mode estimator with exact counters and approximate counters. The experiment is plotted every 10 iterations for 20 topics and 50,000 documents. The sampler with approximate counters is designed to handle large data loads, but also happens to have a better runtime. This runtime advantage increases with larger number of topics.

usage within the GPU.

Unfortunately, the matrices used in Algorithms 1 and 2 are very large. However, note that even the basic version of Mean-for-Mode estimation described in the previous section has some advantages in these respects over both uncollapsed and collapsed Gibbs samplers.

Unlike the uncollapsed Gibbs sampler, Mean-for-Mode estimation does not sample the parameters from Dirichlet distributions. Dirichlet variates are often generated by sampling from Gamma distributions and then normalizing. The shape parameters of these Gamma distributions are computed by adding the priors to the appropriate count from `wpt` or `tpd`. If α or β are small (for instance, 0.01) and the current count from `wpt` or `tpd` is very small, it is quite likely that the unnormalized Gamma variate is not representable with single-precision floating-points.[3] This means uncollapsed Gibbs samplers must store these samples as double-precision floating-point values to prevent rounding off to 0. In contrast, because it doesn't need to draw from a Gamma distribution, the Mean-for-Mode estimator can store these parameters as single-precision floats, which are smaller and faster to process. Moreover, some GPU architectures, including the ones used in our experiments, have many more single-precision cores than double-precision cores, so using single-precision parameters achieves greater parallelism on such GPUs.

Although the collapsed Gibbs sampler does not have to store these parameters at all, the trade-off is that it must store the latent variables. This requires space on the order of the total number of tokens in the corpus, and is typically much larger than the parameter matrices when analyzing many documents.

[3]It is common to use β as small as 0.01, and the smallest positive single precision float is 2^{-149}. The CDF of the Gamma distribution with shape 0.01 and scale of 1 at 2^{-149} is ≈ 0.358.

4.1. Approximating the Counters

In Algorithm 1, note that the matrices `wpt` and `tpd` are cleared to zero at the very beginning of the algorithm, and that the only updates to their entries are increments. (In contrast, with the collapsed Gibbs sampler, the corresponding matrices are never cleared to zero, and updates can involve both increments and decrements.) This feature of the Mean-for-Mode algorithm (and in general of uncollapsed algorithms) makes it possible to use approximate counters for the counts in `wpt` and `tpd`.

The intuitive idea of approximate counters (Morris, 1978) is to estimate the order of magnitude of the number of increments. As a simple example, assume we want to increment a counter and the current value it stores is X. We increment X with probability 2^{-X}, and otherwise do nothing. In the end, a statistically reasonable estimate of the number of increment attempts is $2^X - 1$. This idea can be improved in different ways to allow for different level of precision (Morris, 1978) or to have a behavior similar to that of a floating-point representation (Csűrös, 2010).

The benefit of using approximate counters is obvious: we can greatly reduce the amount of memory required to store the counts; for instance, we could decide to use only 8 bits per counter, or even 4. However, this raises two questions. First, can we use approximate counters while preserving the statistical performance? Second, what is the impact on runtime performance? In Figure 6, we show that, for

8-bit counters, the approximation has no consequence on the statistical performance. More surprisingly, perhaps, the approximate counters lead to a gain in runtime performance (in our experience, this gain increases with the number of topics) despite the fact that an increment now requires drawing from a uniform distribution. This is likely because we perform fewer writes to memory when incrementing approximate counters, since each write happens only with some probability. In addition, the reads from memory for a warp only need to load 32 bytes instead of 128.

4.2. Coalescing Memory Accesses

Finally, a key point about Algorithm 1 is the memory access to the ϕ matrix in the innermost loop. Consider that several threads, working on distinct documents, are each processing a distinct word. As they all execute this memory access in a SIMD fashion, they access non-contiguous chunks of memory. This is a so-called un-coalesced memory access, and it is a significant bottleneck. To cope with this performance issue, we need to have the threads work together to bring the relevant data from memory to the registers, and indeed we can entirely redesign the algorithm so that the data is shared between the threads. This "butterfly" optimization (Steele & Tristan, 2015) is also applicable to sampling from categorical distributions in other mixture models. The runtime effects of this optimization for LDA with Mean-for-Mode estimation are presented in Figure 7.

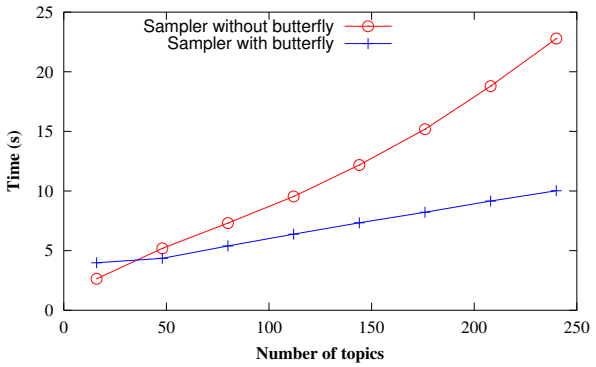

Figure 7. Time to draw 100 samples as a function of topic number for the Mean-for-Mode estimation with the butterfly optimization turned on or off. The butterfly optimization is very effective as the number of topics becomes larger.

5. Sparse Representation of Probability Matrices

The count matrices `wpt` and `tpd` are typically very sparse. It is possible for collapsed Gibbs samplers to take advantage of this sparsity (Porteous et al., 2008; Yao et al., 2009) to significantly improve runtime performance. However, this sparsity technique is difficult to use with GPUs. As Lu et al. (2013) note, the problem is that as threads draw new values for the latent variables and update the count matrices, some of the entries which were zero will become non-zero. This is fine on CPUs, where we can use sparse matrix representations that can easily be resized dynamically. However, these representations are not efficient on GPUs.

Meanwhile, in an uncollapsed Gibbs sampler, the function that draws latent variables reads from the probability matrices $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$, but updates only the `wpt` and `tpd`. Since the $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ matrices are static during latent variable assignment, a sparse representation would not need to be dynamically resized. Unfortunately, these matrices are drawn from Dirichlet distributions, so they are not sparse. This is a considerable drawback, and in fact it makes it difficult to justify using an uncollapsed Gibbs sampler, despite the ample parallelism.

Fortunately, Mean-for-Mode estimation overcomes both of these issues. Since the algorithm uses deterministic point estimates for the values of the $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ matrices, these matrices can be stored sparsely. In addition, these matrices are not modified while drawing new values for the latent variables, so there is no need to resize them dynamically. We use a dense representation of `wpt` and `tpd` so that they can be updated during latent variable assignment. This makes it possible to use sparsity to reduce the sampling complexity, even on GPUs.

The total unnormalized probability mass for the topic assignment of word v appearing in document m is proportional to

$$\sum_k \left[(\texttt{tpd}[m][k] + \alpha) \times \frac{\texttt{wpt}[k][v] + \beta}{\texttt{wt}[k] + \beta V} \right]$$

so, as pointed out by Yao et al. (2009), we can decompose the probability calculation as the sum of three terms ("buckets") S, R_m, and Q_{mv} where

$$S = \sum_k \frac{\alpha\beta}{\texttt{wt}[k] + \beta V} \qquad R_m = \sum_k \frac{\texttt{tpd}[m][k]\beta}{\texttt{wt}[k] + \beta V}$$

$$Q_{mv} = \sum_k \frac{\texttt{tpd}[m][k] + \alpha}{\texttt{wt}[k] + \beta V} \times \texttt{wpt}[k][v]$$

where S is the same for every document, R_m is the same for every word in document m, and Q_{mv} depends on both the word and the document. To make use of this rewriting when choosing a topic (cf. Figure 1, lines 13–18) we pre-compute S, as well as R_m for each m. Then, as the thread processing document m needs to sample a topic for some word v, it computes Q_{mv}. It then decides which "bucket" to draw from based on their relative masses. If it selects buckets R_m or Q_{mv}, we consider only topics whose counts are non-zero,

Algorithm 3 LDA sampler using both sparse and dense matrices

```
estimate_phi()
estimate_theta()
for i = 0 to T do
    float S
    if i <= D then
        draw_z()
    else
        sparse_draw_z(S)
    end if
    if i < D or else i == T then
        estimate_phi()
        estimate_theta()
    else
        S = pre_compute()
        estimate_phi_sparse()
        estimate_theta_sparse()
        segmented reduction thetas
    end if
end for
```

which greatly reduces the number of multiplications that must be done.

Algorithm 3 presents the sampler. It corresponds to the high-level Mean-for-Mode algorithm presented in Figure 2. For performance reasons, it is best to start out using dense matrices, then switch to using sparse representation after a few iterations, once the count matrices become sufficiently sparse. The number of initial iterations that are done using dense probability matrices corresponds to the parameter D.

Once the sampler has reached the point where it uses sparse probability matrices, each iteration works as follows. Starting after the latent variables have been drawn, we first precompute (function "pre_compute") many useful terms, including $Z = 1/(\beta V + \texttt{wt}[k])$, and αZ^{-1}, $\alpha\beta Z^{-1}$, βZ^{-1} for all k. In the remainder, for simplicity of presentation, we assume that these values are pre-computed and stored. We perform a reduction over the terms $\alpha\beta Z^{-1}$ to obtain S.

Then the functions "estimate_phi_sparse" and "estimate_theta_sparse" take as input the counts and produce sparse matrices, respectively in Compressed Sparse Column representation and Compressed Sparse Row representation. Although we call these matrices `phis` and `thetas`, in analogy with the non-sparse algorithm, they are now really terms that are used in computing the R and Q buckets described above:

$$\texttt{thetas} = m \mapsto \left\{ \left(k, \frac{\texttt{tpd}[m][k]}{\texttt{wt}[k] + \beta V} \right) : \texttt{tpd}[m][k] \neq 0 \right\}$$

$$\texttt{phis} = v \mapsto \{(k, \texttt{wpt}[k][v]) : \texttt{wpt}[k][v] \neq 0\}$$

Note that when we refer to `tpd` and `wpt` above, we mean their values at the time that the `thetas` and `phis` matrices are produced. In the remainder, when using these sparse matrices, we use the following notations (described for `phis` but valid for all three matrices). For a given word v, $\texttt{phis}[v]$ refers to the set $\{(k, \texttt{wpt}[k][v]) : \texttt{wpt}[k][v] \neq 0\}$. We write $k \in \texttt{phis}[v]$ if there exists a value a such that $(k, a) \in \texttt{phis}[v]$. If $k \in \texttt{phis}[v]$, we use $\texttt{phis}[k][v]$ to refer to the location in the array `phis` as well as the value at that location.

Once `thetas` has been computed, it is simple to obtain the R_m term for each document by performing a segmented reduction over `thetas` (stored as an array):

$$R_m = \sum_k \texttt{thetas}[m][k] \cdot \beta$$

The function that draws the topics in the sparse case (function "sparse_draw_z") is different from Algorithm 1. First, we need to decide which bucket to draw from by calculating Q_{mv}. Then we need to select from topics with non-null probabilities. To calculate Q_{mv}, the thread processing document m computes:

$$\left(\frac{\alpha}{\texttt{wt}[k] + \beta V} + \texttt{tpd}[m][k] \right) \times \texttt{wpt}[k][v]$$

for each k and sums them. Since these terms are zero whenever $\texttt{phis}[k][v]$ is, the sum can be computed by iterating through the sparse representation of the $\texttt{phis}[k]$ row, instead of all possible values of k.[4]

In Figure 8 we show how different values of D affect the sampling complexity. The case where $D = T$ is the standard Mean-for-Mode. Choosing $D = 0$ (that is, using only sparse matrices) is initially much slower, but eventually, once the matrices become sparse, it becomes much quicker to draw a sample. Finally, by switching from dense to sparse at a good time, it is possible to benefit from both types of learning. For example, in the streaming setting, the dense version could be used to train an initial model quickly, and then the sparse version could be used to quickly update the model as subsequent documents become available (Yao et al., 2009).

6. Related Work

Many papers have been written about parallelizing and distributing Gibbs sampling for LDA. Most follow from the work on AD-LDA by Newman et al. (2009), which is based on the collapsed Gibbs sampler of Griffiths & Steyvers

[4] A careful reader may note that this computation requires accessing the `tpd` matrix, which could be costly since it is sparse. An effective implementation can implement a filter to quickly test whether $\texttt{tpd}[m][k]$ is in the sparse matrix. One particularly space-efficient implementation could use a Bloom filter (Bloom, 1970).

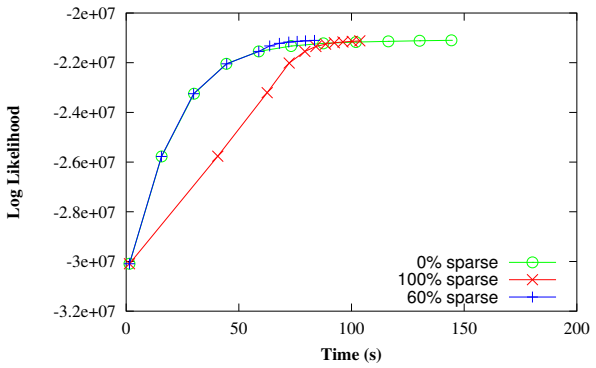

Figure 8. Evolution of log likelihood over time for the Mean-for-Mode estimator with different number of sparse iterations. The experiment is plotted every 10 iterations for 20 topics and 50,000 documents. The version of the sampler that uses sparse representations throughout takes longer to draw initial samples compared to the dense version, but as it begins to converge, subsequent samples are drawn much more quickly. Balancing the use of dense and sparse matrices produces the fastest version overall.

(2004). Distributed implementations include Wang et al. (2009); Smola & Narayanamurthy (2010); Liu et al. (2011). Some of these, such as the implementation of Smola & Narayanamurthy (2010) are very efficient.

Some authors have also implemented parallel versions of the collapsed Gibbs sampler on GPUs. Yan et al. (2009) provide an implementation based on AD-LDA. Of course, they cannot afford to replicate the matrix of counts `wpt` as one must do for a distributed implementation, and to cope with this issue, they make sure that different threads are in charge of different parts of the vocabulary. This unfortunately introduces synchronization rounds that are linear with the number of threads, which will not scale well with more recent GPU architectures and the ever-increasing number of cores.

Lu et al. (2013) provide an implementation also based on AD-LDA, but the counts of `wpt` are shared between all of the threads and are accessed concurrently. This implementation also makes use of sparsity, although the implementation is fairly complex. However, the use of a collapsed Gibbs sampler precludes the use of approximate counters which are critical to fit more data on the graphics card. Also, their benchmarks show that scalability is limited, and this is probably due to the much higher number of memory accesses required to keep track of the counts.

The experiments by Tristan et al. (2014) show that an uncollapsed Gibbs sampler is a good candidate for a GPU implementation. However, it suffers from the statistical performance issues described by Newman et al. (2009) and cannot use the sparsity of the count matrices.

Instead, our algorithm is based on an improved uncollapsed Gibbs sampler and designed in the first place to scale on GPUs. Another example that designs a novel inference method for GPUs is the work of Zhao et al. (2014). Although our algorithm is quite different, we believe it could benefit from the techniques they describe, and vice-versa.

7. Conclusion

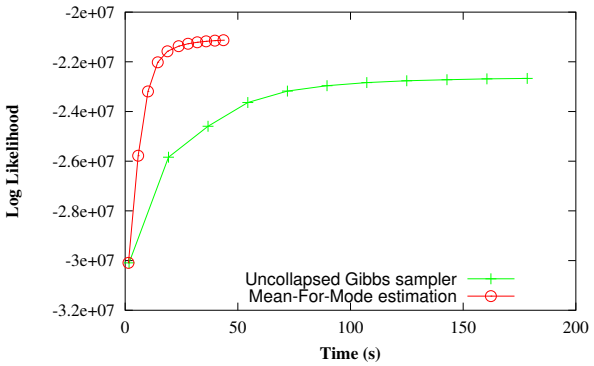

Figure 9. Evolution of log likelihood over time for a highly-optimized uncollapsed Gibbs sampler compared to the Mean-for-Mode estimator presented in this paper. The Mean-for-Mode estimation also has a much smaller memory footprint and can handle more data, which is useful given the scalability characteristics of GPU implementation of training for LDA.

We have shown how Mean-for-Mode estimation, a variant of uncollapsed Gibbs sampling that uses the mean of Dirichlet distributions to make point estimates, can be implemented efficiently on GPUs. The algorithm exposes a lot of parallelism and has good statistical performance. It also lends itself to reducing the total amount of computation by taking advantage of sparsity. The overall gain in performance on our running example is presented in Figure 9, and the gain grows larger with increased number of documents or topics. In addition to exposing sufficient parallelism, Mean-for-Mode estimation also enables the use of techniques to reduce memory footprint and bandwidth use. By combining sparse representations, approximate counters, and avoiding the need to store latent variables, we are able to process larger data sets on a single GPU node.

Although we have focused here on using Mean-for-Mode estimation for LDA, we believe similar techniques could be applied to parameter estimation for other large mixture models. In addition, the memory-saving optimizations we describe here may be applicable in the distributed setting, where minimizing communication and maximizing the amount of data processed per node is important.

Acknowledgments

The authors thank Adam Pocock and Jeffrey Alexander for installing, configuring, and maintaining the software and hardware that we rely on for our research.

References

Asuncion, Arthur, Welling, Max, Smyth, Padhraic, and Teh, Yee Whye. On smoothing and inference for topic models. In *Proc. Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pp. 27–34, Arlington, Virginia, USA, 2009. AUAI Press. ISBN 978-0-9749039-5-8. http://dl.acm.org/citation.cfm?id=1795114.1795118 See also http://www.auai.org/uai2009/proceedings.html.

Bishop, Christopher M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

Blei, David M. Probabilistic topic models. *Commun. ACM*, 55(4):77–84, April 2012. ISSN 0001-0782. doi: 10.1145/2133806.2133826.

Blei, David M., Ng, Andrew Y., and Jordan, Michael I. Latent Dirichlet allocation. *J. Machine Learning Research*, 3:993–1022, March 2003. ISSN 1532-4435. http://dl.acm.org/citation.cfm?id=944919.944937.

Bloom, Burton H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. ISSN 0001-0782. doi: 10.1145/362686.362692.

Celeux, Gilles, Chauveau, Didier, and Diebolt, Jean. On stochastic versions of the EM algorithm. Technical Report RR-2514, Institut National de Recherche en Informatique et Automatique, 1995. https://hal.inria.fr/inria-00074164.

Csűrös, Miklós. Approximate counting with a floating-point counter. In Thai, M. T. and Sahni, Sartaj (eds.), *Computing and Combinatorics (COCOON 2010)*, number 6196 in Lecture Notes in Computer Science, pp. 358–367. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14030-3. doi: 10.1007/978-3-642-14031-0_39. See also http://arxiv.org/pdf/0904.3062.pdf.

Griffiths, Thomas L. and Steyvers, Mark. Finding scientific topics. *Proc. National Academy of Sciences of the United States of America*, 101(suppl 1):5228–5235, 2004. doi: 10.1073/pnas.0307752101.

Liu, Zhiyuan, Zhang, Yuzhou, Chang, Edward Y., and Sun, Maosong. PLDA+: Parallel latent Dirichlet allocation with data placement and pipeline processing. *ACM Trans. Intelligent Systems Technologies*, 2(3):26:1–26:18, May 2011. ISSN 2157-6904. doi: 10.1145/1961189.1961198.

Lu, Mian, Bai, Ge, Luo, Qiong, Tang, Jie, and Zhao, Jiuxin. Accelerating topic model training on a single machine. In Ishikawa, Yoshiharu, Li, Jianzhong, Wang, Wei, Zhang, Rui, and Zhang, Wenjie (eds.), *Web Technologies and Applications (APWeb 2013)*, volume 7808 of *Lecture Notes in Computer Science*, pp. 184–195. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37400-5. doi: 10.1007/978-3-642-37401-2_20.

Morris, Robert. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, October 1978. ISSN 0001-0782. doi: 10.1145/359619.359627.

Murphy, Kevin P. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

Newman, David, Asuncion, Arthur, Smyth, Padhraic, and Welling, Max. Distributed algorithms for topic models. *J. Machine Learning Research*, 10:1801–1828, December 2009. ISSN 1532-4435. http://dl.acm.org/citation.cfm?id=1577069.1755845.

Porteous, Ian, Newman, David, Ihler, Alexander, Asuncion, Arthur, Smyth, Padhraic, and Welling, Max. Fast collapsed Gibbs sampling for latent Dirichlet allocation. In *Proc. 14th ACM SIGKDD Intl. Conf. Knowledge Discovery and Data Mining*, KDD '08, pp. 569–577, New York, 2008. ACM. ISBN 978-1-60558-193-4. doi: 10.1145/1401890.1401960.

Smola, Alexander and Narayanamurthy, Shravan. An architecture for parallel topic models. *Proc. VLDB Endowment*, 3(1-2):703–710, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920931.

Steele, Jr, Guy L. and Tristan, Jean-Baptiste. Using butterfly-patterned partial sums to optimize GPU memory accesses for drawing from discrete distributions. *ArXiv e-prints*, arXiv:1505.03851 [cs.DC], 2015.

Tristan, Jean-Baptiste, Huang, Daniel, Tassarotti, Joseph, Pocock, Adam C., Green, Stephen, and Steele, Guy L., Jr. Augur: Data-parallel probabilistic modeling. pp. 2600–2608. Curran Associates, Inc., 2014. http://papers.nips.cc/book/year-2014.

Wallach, Hanna M., Mimno, David, and McCallum, Andrew. Rethinking LDA: Why priors matter. In *Advances in Neural Information Processing Systems 22*, pp. 1973–1981. Curran Associates, Inc., 2009. http://papers.nips.cc/book/year-2009.

Wang, Yi, Bai, Hongjie, Stanton, Matt, Chen, Wen-Yen, and Chang, EdwardY. PLDA: Parallel latent Dirichlet allocation for large-scale applications. In Goldberg, Andrew V. and Zhou, Yunhong (eds.), *Algorithmic Aspects in Information and Management (AAIM 2009)*, volume

5564 of *Lecture Notes in Computer Science*, pp. 301–314. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02157-2. doi: 10.1007/978-3-642-02158-9_26.

Yan, Feng, Xu, Ningyi, and Qi, Yuan. Parallel inference for latent Dirichlet allocation on graphics processing units. In *Advances in Neural Information Processing Systems 22*, pp. 2134–2142. Curran Associates, Inc., 2009. http://papers.nips.cc/book/year-2009.

Yao, Limin, Mimno, David, and McCallum, Andrew. Efficient methods for topic model inference on streaming document collections. In *Proc. 15th ACM SIGKDD Intl. Conf. Knowledge Discovery and Data Mining*, KDD '09, pp. 937–946, New York, 2009. ACM. ISBN 978-1-60558-495-9. doi: 10.1145/1557019.1557121.

Zhao, Huasha, Jiang, Biye, and Canny, John. SAME but different: Fast and high-quality Gibbs parameter estimation. *CoRR*, September 2014. http://arxiv.org/abs/1409.5402.