Parallel Latent Dirichlet Allocation on GPUs

Gordon E. Moon, Israt Nisa, Aravind Sukumaran-Rajam,
Bortik Bandyopadhyay, Srinivasan Parthasarathy, and P. Sadayappan

The Ohio State University, Columbus OH 43210, USA,
{moon.310, nisa.1, sukumaranrajam.1, bandyopadhyay.14, parthasarathy.2, sadayappan.1}@osu.edu

Abstract. Latent Dirichlet Allocation (LDA) is a statistical technique for topic modeling. Since it is very computationally demanding, its parallelization has garnered considerable interest. In this paper, we systematically analyze the data access patterns for LDA and devise suitable algorithmic adaptations and parallelization strategies for GPUs. Experiments on large-scale datasets show the effectiveness of the new parallel implementation on GPUs.

Keywords: Parallel Topic Modeling · Parallel Latent Dirichlet Allocation · Parallel Machine Learning.

1 Introduction

Latent Dirichlet Allocation (LDA) is a powerful technique for topic modeling originally developed by Blei et al. [1]. Given a collection of documents, each represented as a collection of words from an active vocabulary, LDA seeks to characterize each document in the corpus as a mixture of latent topics, where each topic is in turn modeled as a mixture of words in the vocabulary.

The sequential LDA algorithm of Griffiths et al. [2] uses collapsed Gibbs sampling (CGS) and was extremely compute-intensive. Therefore, a number of parallel algorithms have been devised for LDA, for a variety of targets, including shared-memory multiprocessors [13], distributed-memory systems [6,12], and GPUs (Graphical Processing Units) [15,5,11,17,14]. In developing a parallel approach to LDA, algorithmic degrees of freedom can be judiciously matched with inherent architectural characteristics of the target platform. In this paper, we conduct an exercise in architecture-conscious algorithm design and implementation for LDA on GPUs.

In contrast to multi-core CPUs, GPUs offer much higher data-transfer bandwidths from/to DRAM memory but require much higher degrees of exploitable parallelism. Further, the amount of available fast on-chip cache memory is orders of magnitude smaller in GPUs than CPUs. Instead of the fully sequential collapsed Gibbs sampling approach proposed by Griffiths et al. [2], different forms of *uncollapsed* sampling have been proposed by several previous efforts [10,11] in order to utilize parallelism in LDA. We perform a systematic exploration of the space of partially collapsed Gibbs sampling strategies by

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93701-4_20

a) performing an empirical characterization of the impact on convergence and perplexity, of different sampling variants and

b) conducting an analysis of the implications of different sampling variants on the computational overheads for inter-thread synchronization, fast storage requirements, and implications on the expensive data movement to/from GPU global memory.

The paper is organized as follows. Section 2 provides the background on LDA. Section 3 presents the high-level overview of our new LDA algorithm (AGA-LDA) for GPUs, and Section 4 details our algorithm. In Section 5, we compare our approach with existing state-of-the-art GPU implementations. Section 6 summarizes the related works.

2 LDA Overview

Algorithm 1 Sequential CGS based LDA

Input: *DATA*: *D* documents and **x** word tokens in each document, *V*: vocabulary size, *K*: number of topics, α, β: hyper-parameters
Output: *DT*: document-topic count matrix, *WT*: word-topic count matrix, *NT*: topic-count vector, *Z*: topic assignment matrix

```
1: repeat
2:   for document = 0 to D − 1 do
3:     L ← document_length
4:     for word = 0 to L − 1 do
5:       current_word ← $DATA$[document][word]
6:       old_topic ← $Z$[document][word]
7:       decrement $WT$[current_word][old_topic]
8:       decrement $NT$[old_topic]
9:       decrement $DT$[document][old_topic]
10:      sum ← 0
11:      for k = 0 to K − 1 do
12:        sum←sum + $\frac{WT[current\_word][k]+\beta}{NT[k]+V\beta}$ $(DT[document][k] + \alpha)$
13:        p[k] ← sum
14:      end for
15:      U ← random_uniform() × sum
16:      for new_topic = 0 to K − 1 do
17:        if U < p[new_topic] then
18:          break
19:        end if
20:      end for
21:      increment $WT$[current_word][new_topic]
22:      increment $NT$[new_topic]
23:      increment $DT$[document][new_topic]
24:      $Z$[document][word] ← new_topic
25:    end for
26:  end for
27: until convergence
```

Latent Dirichlet Allocation (LDA) is an effective approach to topic modeling. It is used for identifying latent topics distributions for collections of text documents [1]. Given *D* documents represented as a collection of words, LDA determines a latent topic distribution for each document. Each document j of D documents is modeled as a random mixture over K latent topics, denoted by θ_j. Each topic k is associated with a multinomial distribution over a vocabulary of V unique words denoted by ϕ_k. It is assumed that θ and ϕ are drawn from Dirichlet priors α and β. LDA iteratively improves θ_j and ϕ_k until convergence. For the i^{th} word token in document j, a topic-assignment variable z_{ij} is sampled according to the topic distribution of the document $\theta_{j|k}$, and the word x_{ij} is drawn from the topic-specific distribution of the word $\phi_{w|z_{ij}}$. Asuncion et al. [9] succinctly describe various inference techniques, and their similarities and differences for state-of-the-art LDA algorithms. A more recent survey [3] discusses in greater detail the vast amount of work done on LDA. In context of our work, we first discuss two main variants, viz., *Collapsed Gibbs Sampling (CGS)* and *Uncollapsed Gibbs Sampling (UCGS)*.

Collapsed Gibbs Sampling To infer the posterior distribution over latent variable z, a number of studies primarily used Collapsed Gibbs Sampling (CGS) since it reduces the variance considerably through marginalizing out all prior distributions of $\theta_{j|k}$ and $\phi_{w|k}$ during the sampling procedure [6,15,16]. Three key data structures are updated as each word is processed: a 2D array DT maintaining the document-to-topic distribution, a 2D array WT representing word-to-topic distribution, and a 1D array NT holding the topic-count distribution. Given the three data structures and all words except for the topic-assignment variable z_{ij}, the conditional distribution of z_{ij} can be calculated as:

$$P(z_{ij} = k|\mathbf{z}^{\neg ij}, \mathbf{x}, \alpha, \beta) \propto \frac{WT^{\neg ij}_{x_{ij}|k} + \beta}{NT^{\neg ij}_{k} + V\beta}(DT^{\neg ij}_{j|k} + \alpha) \tag{1}$$

where $DT_{j|k} = \sum_w S_{w|j|k}$ denotes the number of word tokens in document j assigned to topic k; $WT_{w|k} = \sum_j S_{w|j|k}$ denotes the number of occurrences of word w assigned to topic k; $NT_k = \sum_w N_{w|k}$ is the topic-count vector. The superscript $\neg ij$ means that the previously assigned topic of the corresponding word token x_{ij} is excluded from the counts. The hyper-parameters, α and β control the sparsity of DT and WT matrices, respectively. Algorithm 1 shows the sequential CGS based LDA algorithm.

Uncollapsed Gibbs Sampling The use of Uncollapsed Gibbs Sampling (UCGS) as an alternate inference algorithm for LDA is also common [10,11]. Unlike CGS, UCGS requires the use of two additional parameters θ and ϕ to draw latent variable z as follows:

$$P(z_{ij} = k|\mathbf{x}) \propto \phi_{x_{ij}|k}\theta_{j|k} \tag{2}$$

Rather than immediately using DT, WT and NT to compute the conditional distribution, at the end of each iteration, newly updated local copies of DT, WT and NT are used to sample new values on θ and ϕ that will be levered in the next iteration. Compared to CGS, this approach leads to slower convergence since the dependencies between the parameters (corresponding word tokens) is not fully being utilized [6,11]. However, the use of UCGS facilitates a more straightforward parallelization of LDA.

3 Overview of Parallelization Approach for GPUs

As seen in Figure 1, the standard CGS algorithm requires updates to the DT, WT and NT arrays after each sampling step to assign a new topic to a word in a document. This is inherently sequential. In order to achieve high performance on GPUs, a very high degree of parallelism (typically thousands or tens/hundreds of thousands of independent operations) is essential. We therefore divide the corpus of documents into mini-batches which are processed sequentially, with the words in the mini-batch being processed in parallel. Different strategies can be employed for updating the three key data arrays DT, WT and NT. At one extreme, the updates to all three arrays can be delayed until the end of processing of a mini-batch, while at the opposite end, immediate concurrent updates can be performed by threads after each sampling step. Intermediate choices between these two extremes for processing updates also exist, where some of the data arrays are immediately updated, while others are updated at the end of a mini-batch. There are several factors to consider in devising a parallel LDA scheme on GPUs:

– Immediate updates to all three data arrays DT, WT and NT would likely result in faster convergence since this corresponds most closely to fully CGS. At the other extreme, delayed updates for all three arrays may be expected to result in the slowest convergence, with immediate updates to a subset of arrays resulting in an intermediate rate of convergence.
– Immediate updating of the arrays requires the use of atomic operations, which are very expensive on GPUs, taking orders of magnitude more time than arithmetic operations. Further, the cost of atomics depends on the storage used for the operands, with atomics on global memory operands being much more expensive than atomics on data in shared memory .
– While delayed updates mean that we can avoid expensive atomics, additional temporary storage will be required to hold information about the updates to be performed at the end of a mini-batch, since storage is scarce on GPUs, especially registers and shared-memory.
– The basic formulation of CGS requires an expensive division operation (Equation 1) in the innermost loop of the computation for performing sampling. If we choose to perform delayed updates to DT, an efficient strategy can be devised whereby the old DT entries corresponding to a minibatch can be scaled by the division operation by means of the denominator term in Equation 1 once before processing of a mini-batch commences. This will enable the innermost loop for sampling to no longer requires an expensive division operation.

In order to understand the impact on convergence rates for different update choices for DT, WT and NT, we conducted an experiment using four datasets and all possible combinations of immediate versus delayed updates for the three key data arrays. As shown in Figure 1, standard CGS (blue line) has a better convergence rate per-iteration than fully delayed updates (red line). However, standard CGS is sequential and is not suitable for GPU parallelization. On the other hand, delayed update scheme is fully parallel but suffers from a lower convergence rate per-iteration. In our scheme, we divide the documents into mini-batches. Each document within a mini-batch is processed using delayed updates. At the end of each mini-batch, DT, WT and NT are updated and the next mini-batch uses the updated DT, WT and NT values. Note that the mini-batches are processed sequentially.

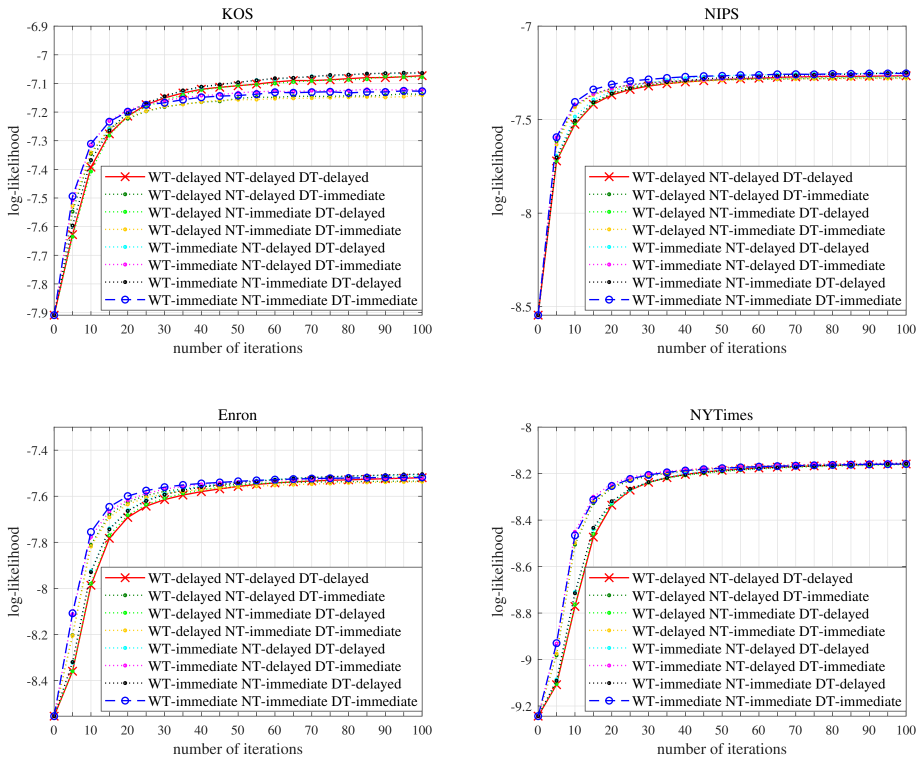

Fig. 1. Convergence over number of iterations on KOS, NIPS, Enron and NYTimes datasets. The mini-batch sizes are set to 330, 140, 3750 and 28125 for KOS, NIPS, Enron and NYTimes, respectively. X-axis: number of iterations; Y-axis: per-word log-likelihood on test set.

Each data structure can be updated using either delayed updates or atomic operations. In delayed updates, the update operations are performed at the end of each mini-batch and is faster than using atomic operations. The use of atomic operations to update DT, WT and NT makes the updates closer to standard sequential CGS, as each update is immediately visible to all the threads. Figure 1 shows the convergence rate of using delayed updates and atomic updates for each DT, WT and NT. Using atomic-operations enables a better convergence rate per-iteration. However, global memory atomic operations are expensive compared to shared memory atomic operations. Therefore, in order to reduce the overhead of atomic operations, we map WT to shared memory. In addition to reducing the overhead of atomics, this also helps to achieve good data reuse for WT from shared memory.

In order to achieve the required parallelism on GPUs, we parallelize across documents and words in a mini-batch. GPUs have a limited amount of shared-memory per SM. In order to take advantage of the shared-memory, we map WT to shared-memory. Each mini-batch is partitioned into columns such that the WT corresponding to each column panel fits in the shared-memory. Shared-memory also offers lower atomic operation costs. DT is streamed from global memory. However, due to mini-batching most of these accesses will be served by the L2 cache (shared across all SMs). Since multiple threads work on the same document and DT is kept in global memory, expensive global memory atomic updates are required to update DT. Hence, we use delayed updates for DT. Figure 2 depicts the overall scheme.

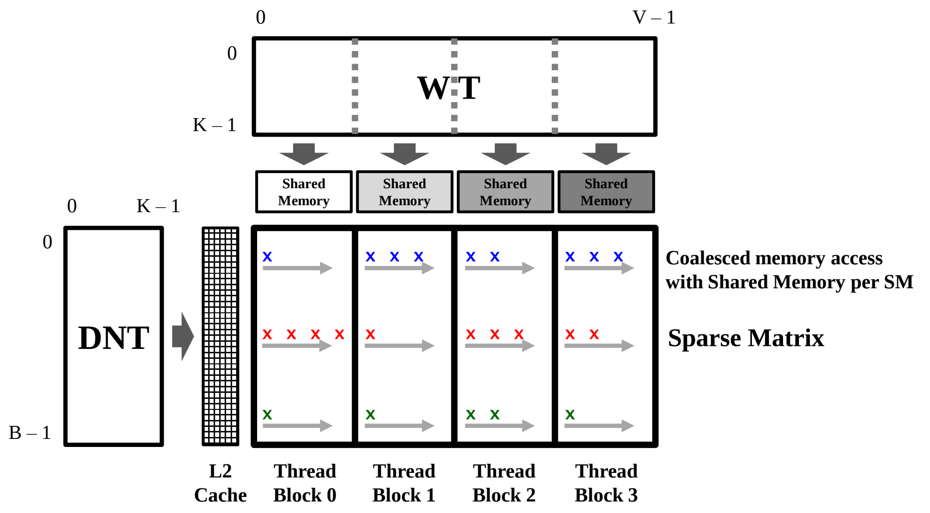

Fig. 2. Overview of our approach. V: vocabulary size, B: number of documents in the current mini-batch, K: number of topics

4 Details of Parallel GPU Algorithm

Algorithm 2 GPU implementation of sampling kernel

Input: DOC_IDX, $WORD\_IDX$, Z_IDX: document index, word index and topic index for each nnz in CSB format corresponding to the current mini-batch, $lastIdx$: a vector which stores the start index of each tile, V: vocabulary size, K: number of topics, β: hyper-parameter

```
 1: tile_id = block_id
 2: tile_start = $lastIdx$[tile_id]
 3: tile_end = $lastIdx$[tile_id + 1]
 4: $shared\_WT$[column_panel_width][K]
 5: warp_id = thread_id / WARP_SIZE
 6: lane_id = thread_id % WARP_SIZE
 7: n_warp_k = thread_block_size / WARP_SIZE
    // Coalesced data load from global memory to shared memory
 8: for i=warp_id to column_panel step n_warp_k do
 9:    for w = 0 to K step WARP_SIZE do
10:       $shared\_WT$[i][w+lane_id] = $WT$[(tile_id×col_panel_width+i)][w+lane_id]
11:    end for
12: end for
13: __syncthreads()
14: for nnz = thread_id+tile_start to tile_end step thread_block_size do
15:    curr_doc_id = $DOC\_IDX$[nnz]
16:    curr_word_id = $WORD\_IDX$[nnz]
17:    curr_word_shared_id = curr_word_id − tile_id × column_panel_width
18:    old_topic = $Z\_IDX$[nnz]
19:    atomicSub ($shared\_WT$[curr_word_shared_id][old_topic], 1)
20:    atomicSub ($NT$[old_topic], 1)
21:    sum = 0
22:    for k = 0 to K − 1 do
23:       sum += ($shared\_WT$[curr_word_shared_id][k]+$\beta$)×$DNT$[curr_doc_id][k]
24:    end for
25:    U = curand_uniform() × sum
26:    sum = 0
27:    for new_topic = 0 to K − 1 do
28:       sum += ($shared\_WT$[curr_word_shared_id][k]+$\beta$)×$DNT$[curr_doc_id][k]
29:       if U < sum then
30:          break
31:       end if
32:    end for
33:    atomicAdd ($shared\_WT$[curr_word_shared_id][new_topic], 1)
34:    atomicAdd ($NT$[new_topic], 1)
35:    $Z\_IDX$[nnz] = new_topic
36: end for
    // Update WT in global memory
37: for i=warp_id to column_panel step n_warp_k do
38:    for w = 0 to K step WARP_SIZE do
39:       $WT$[(tile_id×col_panel+i)][w+lane_id] = $shared\_WT$[i][w+lane_id]
40:    end for
41: end for
42: __syncthreads()
```

As mentioned in the overview section, we divide the documents into mini-batches. All the documents/words within a mini-batch are processed in parallel, and the processing across mini-batches is sequential. All the words within a mini-batch are partitioned to form column panels. Each column panel is mapped to a thread block.

Shared Memory: Judicious use of shared-memory is critical for good performance on GPUs. Hence, we keep WT in shared-memory which helps to achieve higher memory access efficiency and lower cost for atomic operations. Within a mini-batch, WT gets full reuse from shared-memory.

Reducing global memory traffic for the cumulative topic count: In the original sequential algorithm (Algorithm 1) the cumulative topic is computed by multiplying WT with DT and then dividing the resulting value with NT. The cumulative count with respect to each topic is saved in an array p as shown in Line 13 in Algorithm 1. Then a random number is computed and is scaled by the topic-count-sum across all topics. Based on the scaled random number the cumulative topic count array is scanned again to compute the new topic. Keeping the cumulative count array in global memory will increase the global memory traffic especially as these accesses are uncoalesed. As data movement is much more expensive than computations, we do redundant computations to reduce data movement. In order to compute the topic-count-sum across all topics, we perform a dot product of DT and WT in Line 23 in Algorithm 2. Then a random number which is scaled by the topic sum is computed. The product of DT and WT is recomputed, and based on the value of scaled random number, the new topic is selected. This strategy helps to save global memory transactions corresponding to $2 \times number\ of\ words \times number\ of\ topics$ (read and write) words.

Reducing expensive division operations: In Line 12 in Algorithm 1, division operations are used during sampling. Division operations are expensive in GPUs. The total number of division operations during sampling is equal to $total\ number\ of\ words\ across\ all\ documents \times number\ of\ features$. We can pre-compute $DNT = DT/NT$ (Algorithm 4) and then use this variable to compute the cumulative topic count as shown in Line 23 in Algorithm 2. Thus a division is performed per document as opposed to per word which helps to reduce the total number of division operations to $total\ number\ of\ documents \times number\ of\ features$.

Reducing global memory traffic for DT (DNT): In our algorithm, DT is streamed from global memory. The total amount of DRAM (device memory) transactions can be reduced if we can substitute DRAM access with L2 cache accesses. Choosing an appropriate size for a mini-batch can help to increase L2 hit rates. For example, choosing a low mini-batch size will increase the probability of L2 hit rates. However, if the mini-batch size is very low, there will not be enough work in each mini-batch. In addition, the elements of the sparse matrices are kept in segmented Compressed Sparse Blocks (CSB) format. Thus, the threads with a column panel process all the words in a document before moving on to the next document. This ensures that within a column panel the temporal reuse of DT (DNT) is maximized.

Algorithm 2 shows our GPU algorithm. Based on the column panel, all the threads in a thread block collectively bring in the corresponding WT elements from global memory to shared memory. WT is kept in column major order. All the threads in a warp bring one column of WT and different wraps bring different columns of WT (Line 10). Based on the old topic, the copy of WT in shared memory and NT is decremented using atomic operations (Line 19 and 20).

The non-zero elements within a column panel are cyclically distributed across threads. Corresponding to the non-zero, each thread computes the topic-count-sum by computing the dot product of WT and DNT (Line 23). A random number is then computed and scaled by this sum (Line 25). The product of WT and DNT is then recomputed to find the new topic with the help of the scaled random number (Line 28). Then the copy of WT in shared memory and NT is incremented using atomic operations (Line 33 and 34).

At the end of each column panel, each thread block collectively updates the global WT using the copy of WT kept in shared memory (Line 39).

Algorithm 3 GPU implementation of updating the DT

Input: DOC_IDX, Z_IDX: document index and topic index for each nnz in CSB format corresponding to the current mini-batch

```
1: curr_doc_id = $DOC\_IDX$[thread_id]
2: new_topic = $Z\_IDX$[thread_id]
3: **atomicAdd** ($DT$[curr_doc_id][new_topic], 1)
```

Algorithm 4 GPU implementation of updating the DNT

Input: V: vocabulary size, α, β: hyper-parameters

```
1: curr_doc_id = blockIdx.x
2: $DNT$[curr_doc_id][thread_id] = $\frac{DT[curr\_doc\_id][thread\_id]+\alpha}{NT[thread\_id]+V\beta}$
```

At the end of each mini-batch, we need to update DT and pre-compute DNT for the next mini-batch. Algorithm 3 shows our algorithm to compute DT. All the DT elements are initially set to zero using cudaMemset. We iterate over all the words across all the documents. Corresponding to the topic of each word, we increment the document topic count using atomic operations (Line 3). The pre-computation of DNT is shown in Algorithm 4. In this algorithm, each document is processed by a thread block and the threads within a thread block are distributed across different topics. Based on the document and thread id, each thread computes the DNT as shown in Line 2.

5 Experimental Evaluation

Table 1. Machine configuration

Machine	Details
GPU	GTX TITAN (14 SMs, 192 cores/MP, 6 GB Global Memory, 876 MHz, 1.5MB L2 cache)
CPU	Intel(R) Xeon(R) CPU E5-2680(28 core)

Two publicly available GPU-LDA implementations, **Lu-LDA** by Lu et al. [5] and **BIDMach-LDA** by Zhao et al. [17], are used in the experiments to compare the performance and accuracy of the approach developed in this paper. We label our new implementation as Approximate GPU-Adapted LDA (**AGA-LDA**). We also use GibbsLDA++ [7] (**Sequential CGS**), a standard C++ implementation of sequential LDA with CGS, as a baseline. We use four datasets: the KOS, NIPS, Enron and NYTimes from the UCI Machine Learning Repository [4]. While Table 2 shows the characteristics of the datasets, Table 1 shows the configuration of the machines used for experiments.

Table 2. Dataset characteristics. D is the number of documents, W is the total number of word tokens and V is the size of the active vocabulary.

Dataset	D	W	V
KOS	3,430	467,714	6,906
NIPS	1,500	1,932,365	12,375
Enron	39,861	6,412,172	28,099
NYTimes	299,752	99,542,125	101,636

In BIDMach-LDA, the train/test split is dependent on the size of the mini-batch. To ensure a fair comparison, we use the same train/test split across different LDA algorithms. The train set consists of 90% of documents and the remaining 10% is used as the test set. BIDMach-LDA allows changing the hyper-parameters such as α. We tuned the mini-batch size for both BIDMach-LDA and AGA-LDA and we report the best performance. In AGA-LDA, the hyper-parameters, α and β are set to 0.1. The number of topics (K) in all experiments is set to 128.

5.1 Evaluation Metric

To evaluate the accuracy of LDA models, we use the per-word log-likelihood on the test set. The higher the log-likelihood, the better the generalization of the model on unseen data.

$$log(p(\mathbf{x}^{test})) = \prod_{ij} log \sum_{k} \frac{WT_{w|k} + \beta}{\sum_w WT_{w|k} + V\beta} \frac{DT_{j|k} + \alpha}{\sum_k DT_{j|k} + K\alpha} \tag{3}$$

$$\text{per-word log-likelihood} = \frac{1}{W^{test}} log(p(\mathbf{x}^{test})) \tag{4}$$

where W^{test} is the total number of word tokens in the test set. For each LDA model, training and testing algorithms are paired up.

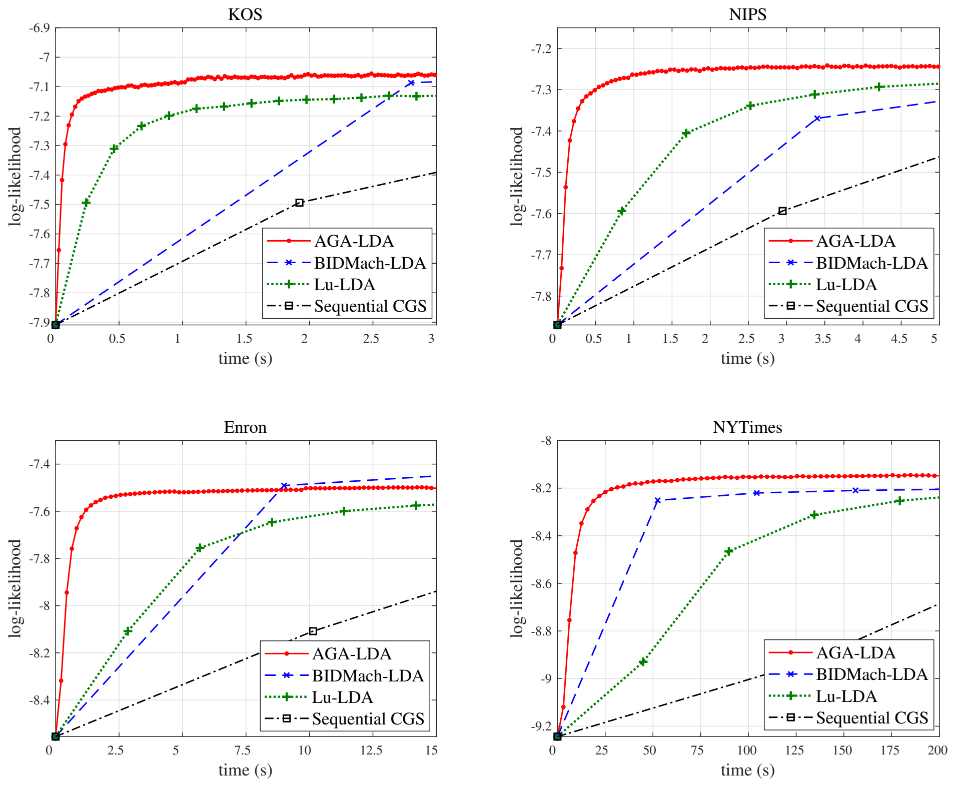

Fig. 3. Convergence over time on KOS, NIPS, Enron and NYTimes datasets. The mini-batch sizes are set to 330, 140, 3750 and 28125 for KOS, NIPS, Enron and NYTimes, respectively.

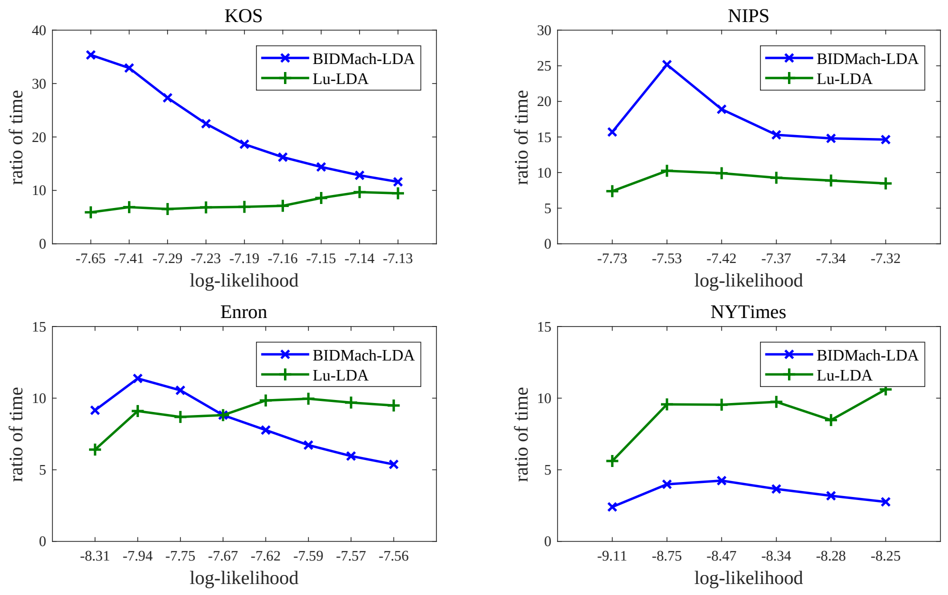

Fig. 4. Speedup of AGA-LDA over BIDMach-LDA and Lu-LDA.

5.2 Speedup

Figure 3 shows the log-likelihood versus elapsed time of the different models. Compared to BIDMach-LDA, AGA-LDA achieved 2.5×, 15.8×, 2.8× and 4.4× on the KOS, NIPS, Enron and NYTimes datasets, respectively. AGA-LDA consistently performs better than other GPU-based LDA algorithms on all datasets. Figure 4 shows the speedup of our approach over BIDMach-LDA and Lu-LDA. The y-axis in Figure 4 is the ratio of time for BIDMach-LDA and Lu-LDA to achieve a log-likelihood to how long AGA-LDA took. The result shows that y-values of all points are greater than one for all cases, indicating that AGA-LDA is faster than the existing state-of-the-art GPU-based LDA algorithms.

6 Related Work

The LDA algorithm is computationally expensive as it has to iterate over all words in all documents multiple times until convergence is reached. Hence many works have focused on efficient parallel implementations of the LDA algorithm both in multi-core CPU as well as many-core GPU platforms.

Multi-core CPU platform Newman et al. [6] justifies the importance of distributed algorithms for LDA for large scale datasets and proposed an *Approxi-*

mate Distributed LDA (AD-LDA) algorithm. In AD-LDA, documents are partitioned into several smaller chunks and each chunk is distributed to one of the many processors in the system, which performs the LDA algorithm on this preassigned chunk. However, global data structures like word-topic count matrix and topic-count matrix have to be replicated to the memory of each processor, which are updated locally. At the end of each iteration, a reduction operation is used to update all the local counts thereby synchronizing the state of the different matrices across all processors. While the quality and performance of the LDA algorithm is very competitive, this method incurs a lot memory overhead and has performance bottleneck due to the synchronization step at the end of each iteration. Wang et al. [12] tries to address the storage and communication overhead by an efficient MPI and MapReduce based implementation. The efficiency of CGS for LDA is further improved by Porteous et al. [8] which leveraging the sparsity structure of the respective probability vectors, without any approximation scheme. This allows for accurate yet highly scalable algorithm. On the other hand, Asuncion et al. [9] proposes approximation schemes for CGS based LDA in the distributed computing paradigm for efficient sampling with competitive accuracy. Xiao et al. [13] proposes a dynamic adaptive sampling technique for CGS with strong theoretical guarantees and efficient parallel implementation. Most of these works either suffer from memory overhead and synchronization bottleneck due to multiple local copies of global data-structures which are later used for synchronization across processors, or have to update key data structures using expensive atomic operations to ensure algorithmic accuracy.

Many-core GPU platform One of the first GPU based implementations using CGS is developed by Yan et al. [15]. They partition both the documents and the words to create a set of disjoint chunks, such that it optimizes memory requirement, avoids memory conflict while simultaneously tackling a load imbalance problem during computation. However, their implementation requires maintaining local copies of global topic-count data structure. Lu et al. [5] tries to avoid too much data replication by generating document-topic counts on the fly and also use succinct sparse matrix representation to reduce memory cost. However, their implementation requires atomic operations during the global update phase which increases processing overhead. Tristan et al. [11] introduces a variant of UCGS technique which is embarrassingly parallel with competitive performance. Zhao et al. [17] proposes a state-of-the-art GPU implementation which combines the SAME (State Augmentation for Marginal Estimation) technique with mini-batch processing.

7 Conclusion

In this paper, we describe a high-performance LDA algorithm for GPUs based on approximated Collapsed Gibbs Sampling. The AGA-LDA is designed to achieve high performace by matching characteristics of GPU architecture. The algorithm is focused on reducing the required data movement and overheads due to atomic operations. In the experimental section, we show that our approach achieves significant speedup when compared to the existing state-of-the-art GPU LDA implementations.

References

1. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. JMLR (2003)
2. Griffiths, T.L., Steyvers, M.: Finding scientific topics. Proceedings of the National academy of Sciences **101**(suppl 1), 5228–5235 (2004)
3. Jelodar, H., Wang, Y., Yuan, C., Feng, X.: Latent dirichlet allocation (lda) and topic modeling: models, applications, a survey. arXiv:1711.04305 (2017)
4. Lichman, M.: UCI machine learning repository (2013), `http://archive.ics.uci.edu/ml`
5. Lu, M., Bai, G., Luo, Q., Tang, J., Zhao, J.: Accelerating topic model training on a single machine. In: APWeb. Springer (2013)
6. Newman, D., Asuncion, A., Smyth, P., Welling, M.: Distributed algorithms for topic models. JMLR (2009)
7. Phan, X.H., Nguyen, C.T.: Gibbslda++: A c/c++ implementation of latent dirichlet allocation (lda) (2007)
8. Porteous, I., Newman, D., Ihler, A., Asuncion, A., Smyth, P., Welling, M.: Fast collapsed gibbs sampling for latent dirichlet allocation. In: SIGKDD. ACM (2008)
9. Smyth, P., Welling, M., Asuncion, A.U.: Asynchronous distributed learning of topic models. In: NIPS (2009)
10. Tristan, J.B., Huang, D., Tassarotti, J., Pocock, A.C., Green, S., Steele, G.L.: Augur: Data-parallel probabilistic modeling. In: NIPS (2014)
11. Tristan, J.B., Tassarotti, J., Steele, G.: Efficient training of lda on a gpu by mean-for-mode estimation. In: ICML (2015)
12. Wang, Y., Bai, H., Stanton, M., Chen, W.Y., Chang, E.Y.: Plda: Parallel latent dirichlet allocation for large-scale applications. AAIM (2009)
13. Xiao, H., Stibor, T.: Efficient collapsed gibbs sampling for latent dirichlet allocation. In: ACML (2010)
14. Xue, P., Li, T., Zhao, K., Dong, Q., Ma, W.: Glda: Parallel gibbs sampling for latent dirichlet allocation on gpu. In: ACA. Springer (2016)
15. Yan, F., Xu, N., Qi, Y.: Parallel inference for latent dirichlet allocation on graphics processing units. In: NIPS (2009)
16. Zhang, B., Peng, B., Qiu, J.: High performance lda through collective model communication optimization. Procedia Computer Science (2016)
17. Zhao, H., Jiang, B., Canny, J.F., Jaros, B.: Same but different: Fast and high quality gibbs parameter estimation. In: SIGKDD. ACM (2015)