Accelerating Topic Model Training on a Single Machine

Mian Lu [1], Ge Bai[2], Qiong Luo[2], Jie Tang[3], Jiuxin Zhao[2]

[1]A*STAR Institute of High Performance Computing, Singapore
lum@ihpc.a-star.edu.sg
[2]Hong Kong University of Science and Technology
{luo, gbai, zhaojx}@cse.ust.hk
[3]Tsinghua University
jietang@tsinghua.edu.cn

Abstract. We present the design and implementation of GLDA, a library that utilizes the GPU (Graphics Processing Unit) to perform Gibbs sampling of Latent Dirichlet Allocation (LDA) on a single machine. LDA is an effective topic model used in many applications, e.g., classification, feature selection, and information retrieval. However, training an LDA model on large data sets takes hours, even days, due to the heavy computation and intensive memory access. Therefore, we explore the use of the GPU to accelerate LDA training on a single machine. Specifically, we propose three memory-efficient techniques to handle large data sets on the GPU: (1) generating document-topic counts as needed instead of storing all of them, (2) adopting a compact storage scheme for sparse matrices, and (3) partitioning word tokens. Through these techniques, the LDA training which would take 10 GB memory originally, can be performed on a commodity GPU card with only 1 GB GPU memory. Furthermore, our GLDA achieves a speedup of 15X over the original CPU-based LDA for large data sets.

1 Introduction

Statistical topic models have recently been successfully applied to text mining tasks such as classification, topic modeling, and recommendation. Latent Dirichlet Allocation (LDA) [1], one of the recent major developments in statistical topical modeling, immediately attracted a considerable amount of interest from both research and industry. However, due to its high computational cost, training an LDA model may take multiple days [2]. Such a long running time hinders the use of LDA in applications that require online performance. Therefore, there is a clear need for methods and techniques to efficiently learn a topic model.

In this paper, we present a solution to the efficiency problem of training the LDA model: acceleration with graphics processing units (GPUs). GPUs have recently become a promising high-performance parallel architecture for a wide range of applications [3]. The current generation GPU provides a 10× higher computation capability

Acknowledgement: This work was supported by grant 617509 from the Research Grants Council of Hong Kong.

and a 10× higher memory bandwidth than a commodity multi-core CPU. Earlier studies of using GPUs to accelerate the LDA have shown significant speedups compared with CPU counterparts [4, 5]. However, due the limited size of GPU memory, these methods are not scalable to large data sets. The GPU memory size is relatively small compared with the main memory size, e.g., up to 6GB for GPUs in the market. This limits the use of GPU-based LDA for practical applications since the memory size required of many real-world data sets clearly exceeds the capability of the GPU memory. Moreover, some parallel implementations of LDA, such as AD-LDA [2], cannot be directly adapted on the GPU either since the overall memory consumption would easily exceed the available GPU memory size due to a full copy of word-topic counts maintained for each processor.

To address this issue, we study how to utilize limited hardware resources for large-scale GPU-based LDA training. Compared with the existing work on GPU-base LDA, our implementation is able to scale up to millions of documents. This scalability is achieved through the following three techniques:

1. *On-the-fly generation of document-topic counts*. We do not precompute and store the global document-topic matrix. Instead, the counts for specific documents are generated only when necessary.
2. *Sparse storage scheme*. We investigate a more compact storage scheme to store the document-topic and word-topic count matrices since they are usually sparse.
3. *Word token partitioning*. Word tokens in the data set are divided into a few disjoint partitions based on several criteria. Then for each partition, only a part of document-topic and word-topic matrices are needed in each iteration.

Based on these techniques, we have successfully trained LDA models on data sets originally requiring up to 10 GB memory on a GPU with only 1 GB memory. Furthermore, our GPU-based LDA training achieved significant performance speedups. These results show that our approach is practical and cost-effective. Also, our techniques can be extended to solutions that involve multiple GPUs.

The remainder of this paper is organized as follows. In Section 2, we briefly introduce the LDA algorithm and review GPGPU. We present the design and implementation of our GLDA in detail in Section 3. In Section 4, we experimentally evaluate our GLDA on four real-world data sets. We conclude in Section 5.

2 Preliminary

2.1 Gibbs Sampling of Latent Dirchlet Allocation

We develop GLDA based on Gibbs sampling of LDA [6] due to its simplicity and high effectiveness for real-world text processing. In the LDA model, D documents are modeled as a mixture over K latent topics. Given a training data set, we use a set x to represent the words in documents, where x_{ij} is the jth word in the ith document. Furthermore, a corresponding topic assignments set z is maintained, where z_{ij} is the assigned topic for the word x_{ij}. The total number of words in the data set is N. Two matrices, n_{jk} and n_{wk} are used in LDA. Specifically, n_{jk} is the document-topic count

– the number of words in document j assigned to topic k, and n_{wk} is the word-topic count – the number of occurrences of word w assigned to topic k. The sizes of matrices n_{jk} and n_{wk} are $D \times K$ and $W \times K$, respectively, where W is the number of words in the vocabulary for the given data set.

The original Gibbs sampling is inherently sequential, as each iteration in the training process depends on the results from the previous iteration. To implement it in parallel, the previous study has proposed a parallel approximate LDA algorithm [2], which has been shown similar accuracy to the sequential algorithm. Therefore, in this work, we adopt a similar parallel approximate LDA algorithm for our GPU implementation.

2.2 Graphics Processing Units (GPUs)

GPUs are widely available as commodity components in modern machines. We adopt the massive thread parallelism programming model from NVIDIA CUDA to design our algorithm. The GPU is modeled as an architecture including multiple SIMD multi-processors. All processors on the GPU share the *device memory*, which has both a high bandwidth and a high access latency. Each multi-processor has a fast on-chip *local memory* (called *shared memory* in NVIDIA's term) shared by all the scalar processors in the multi-processor. The size of this local memory is small and the access latency is low. The threads on each multi-processor are organized into *thread blocks*. *Threads* in a thread block share the computation resources on a multi-processor. Thread blocks are dynamically scheduled on the multi-processors by the runtime system. Moreover, when the addresses of the memory accesses of the multiple threads in a thread block are consecutive, these memory accesses are grouped into one access. This feature is called *coalesced access*.

2.3 Parallel and Distributed LDAs

Two parallel LDA algorithms, named AD-LDA and HD-LDA, are proposed by Newman et al. [2]. The speedup of AD-LDA on a 16-processor computer was up to 8 times. Chen et al. [7] have implemented AD-LDA using MPI and MapReduce for large-scale LDA applications. Their implementations achieved a speedup of 10× when 32 machines were used. Another asynchronous distributed LDA algorithm was proposed by Asuncion et al. [8], which had a speedup of 15-25× when there were 32 processors.

Among existing studies on GPU-based LDAs, Masada et al. [4] have accelerated collapsed variational Bayesian inference for LDA using GPUs and achieved a speedup of up to 7 times compared with the standard LDA on the CPU. Moreover, Yan et al. [5] have implemented both collapsed Gibbs sampling (CGS) and variational Bayesian (CVB) methods of LDA on the GPU. Compared with the sequential implementation on the CPU, the speedups for CGS and CVB are around 26x and 196x, respectively. However, none of these GPU-based LDAs has effectively addressed the scalability issue. In contrast, our GLDA can train very large data sets on the GPU efficiently.

3 GLDA Design and Implementation

3.1 Parallel Gibbs Sampling of LDA on GPUs

A traditional parallel approach [2] for LDA (AD-LDA) is that both D documents and document-topic counts n_{jk} are distributed evenly over p processors. However, each processor needs to maintain a local copy of all word-topic counts. Then each processor performs LDA training based on its own local documents, document-topic, and word-topic counts independently. At the end of each iteration, a reduction operation is performed to update all copies of local n_{wk} counts. The major difficulty in applying this approach to GPUs is that multiple copies of the word-topic count matrix may not fit into the limited GPU memory. Therefore, Yan et al. [5] have proposed a data partitioning method to avoid access conflicts while maintaining only one copy of n_{wk} matrix. The major issue of their implementation is it may cause workload imbalance since the partitioning scheme is based on both documents and vocabulary, and different partitions are processed in parallel. The approximate algorithm they have proposed may not work well for non-uniform data distributions. Moreover, it still cannot handle the data set when either n_{jk} or n_{wk} matrix cannot be entirely stored in the GPU memory. Instead, we have adopted a different algorithm without such 2-dimension partitioning and solved the scalability issue.

Algorithm 1: GLDA algorithm for one iteration.

```
Input:
$x^p$: word tokens assigned to the $p$th processor
Output: $n_{jk}$, $n_{wk}$, $z_{ij}$
for all processors in parallel do
    foreach $x_{ij} \in x^p$ do
        Sample $z_{ij}$ with global counts $n_{jk}$ and $n_{wk}$
        /* Global synchronization */
        Update $n_{jk}$
        Atomic update $n_{wk}$
```

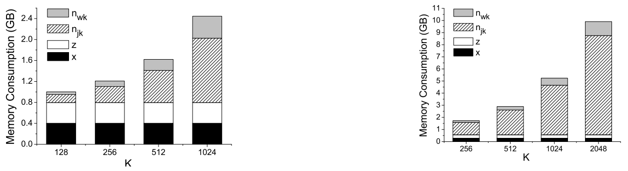
Our parallel algorithm is based on the following fact: With atomic increment and decrement operations, concurrent execution of these operations is guaranteed to produce a correct result. Therefore, in our implementation, we also maintain only one copy of n_{wk} matrix, and adopt two modifications. (1) We use atomic increment and decrement operations, which are supported by recent generation GPUs, in count updates. (2) We serialize the computation and update on the n_{wk} matrix. With these modifications, Algorithm 1 outlines our parallel LDA algorithm for one iteration. Compared with AD-LDA, which does not perform global updates until the end of each iteration, our algorithm performs global updates after each p words are sampled in p processors in parallel and guarantees the updated results are correct. Since the multi-processors on the GPU are tightly coupled, the communication overhead is insignificant in this implementation. Specifically, we map each thread block in CUDA as a processor, and the inherent data parallelism of the sampling algorithm is implemented using multiple threads within each thread block.

3.2 Memory Consumption of Original LDA

We estimate the total memory consumption of the original LDA algorithm in the following four components: the memory used by the word token set $x = \{x_{ij}\}$ and topic assignment set $z = \{z_{ij}\}$ are both $sizeof(int) \times N$ bytes. The document-topic and word-topic count matrix consume $sizeof(int) \times D \times K$ bytes and $sizeof(int) \times W \times K$ bytes, respectively. Therefore, the estimated total memory size is at least M bytes:

$$M = sizeof(int) \times (2 \times N + (D + W) \times K)$$

Where N, D, and W are fixed for a given data set, and the number of topics K is assigned by users. A large K is usually required for large data sets, for example, a typical value is $\sqrt{D}$. Figure 1 shows the estimated memory consumption for NYTimes and PubMed data sets that are used in our evaluations (see Table 1 for experimental setup). Obviously, current generations of GPUs cannot support such large amounts of required memory space. A straightforward solution is to send the data set x_{ij} and topic assignment set z_{ij} to the GPU on-the-fly. However, this method does not solve the problem when either n_{jk} or n_{wk} matrix cannot be entirely stored in the GPU memory. Therefore, we study more advanced techniques to enable our GLDA to handle large data sets.



(a) NYTimes

(b) PubMed

Fig. 1. Estimated memory consumptions for NYTimes and PubMed data sets (Table 1) with number of topics varied.

3.3 Discarding Global Document-Topic Matrix

We have observed that to generate the n_{jk} counts for a given document j, only the word tokens in that document are necessary to scan. Moreover, the sequential sampling of words in a single document can share an array with K elements storing the document-topic counts corresponding to that document. Therefore, we consider discarding the storage for the global document-topic counts. We only produce the necessary n_{jk} counts for specific documents when these documents are processed (the technique is denoted as no_n_{jk}). Since we process p words in p different documents in parallel, the estimated memory size required for this method is $M_{no\_n_{jk}}$ bytes.

$$M_{no\_n_{jk}} = sizeof(int) \times (2 \times N + W \times K + p \times K)$$

Moreover, the additional cost introduced is equivalent to one scan on the topic assignment set z for each iteration. Due to the high memory bandwidth of the GPU, the generation of document-topic counts can be implemented efficiently through the coalesced access pattern.

Algorithm 2: Generating document-topic counts for a specific document through the coalesced access.

```
Input:
$z$: the topic assignment array for a given document in the device memory
$l$: the array with $K$ elements in the local memory
$n$: the number of elements in $z$
$tx$: the thread index in a thread block
$nt$: the number of threads in a thread block
Output:
$n_{jk}$: the document-topic count array with length $K$ in the device memory for the given document
$l$ is initialized to 0;
for $i = tx$; $i < n$; $i = i + nt$ do
    atomicInc($l[z[i]]$);
/*synchronization within the thread block*/
for $k = tx$; $k < K$; $k = k + nt$ do
    $n_{jk}[k] = l[k]$;
```

We use a thread block to generate the document-topic counts for a given document. Algorithm 2 demonstrates the GPU code for a thread to produce document-topic counts based on the coalesced access. We have adopted a temporary array stored in the local memory to improve the memory access efficiency for counting as well as to facilitate the coalesced access for both reads and writes on the device memory. Without storing all n_{jk} counts, the memory saving is significant with a low computation overhead.

In contrast, the n_{wk} counts are stored. This design choice is made because recalculating n_{wk} is expensive - as words are distributed across documents, generating n_{wk} for each iteration would require approximately N scans of the original data set. Fortunately, the number of topics and the size of vocabulary are both much smaller than the data set size; therefore, it is feasible to store the n_{wk} counts even for large data sets. In the following, we further discuss how to store these matrices efficiently and how to partition them if they do not fit into memory.

3.4 Sparse Storage Scheme for Count Matrices

We observe that a considerable portion of the values in the n_{wk} and n_{jk} matrices become zeros after a burn-in period. For example, after about 100 iterations of training on the NIPS data set, which is used in our evaluation, the percentages of non-zero values are around 45% and 10% for n_{jk} and n_{wk} counts, respectively. This observation has motivated us to study the sparse matrix storage scheme for these two counts.

We only store the topic-count pair for non-zero elements for the n_{jk} or n_{wk} count matrix. An array of K topic counts is generated from the sparse storage decompressed for a given document or word token. The decompression can also be implemented efficiently through the coalesced access as well as the local memory usage, which is similar to Algorithm 2. Suppose we use a data structure including two elements to store a topic-count pair, namely id and $count$. Then to generate an array including all topic counts for a specific document or word, the value of $count$ will be written to the idth element in the array for specific pairs. Since p word tokens in p different documents are processed in parallel, we need to decompress those corresponding n_{jk} and n_{wk} counts. The estimated memory required is M_{sparse} bytes.

$$\begin{aligned} M_{sparse} = sizeof(int) \times (2 \times N + 2 \times p \times K + \\ 2 \times (D \times K \times nz_{jk}\% + W \times K \times nz_{wk}\%)) \end{aligned}$$

Where $nz_{jk}\%$ and $nz_{wk}\%$ refer to the percentage of non-zero elements for n_{jk} and n_{wk} counts, respectively. Note that, when $nz_{jk}\%$ or $nz_{wk}\%$ is over 50%, this storage scheme will consume more memory than the original storage format for the n_{jk} or n_{wk} matrix, respectively. Therefore, to decide whether adopting the sparse matrix storage, we should first estimate the memory space saving.

Although the introduced computation overhead is inexpensive, there is a critical problem for the GPU-based implementation. Since the percentage of non-zero elements is not guaranteed to monotonically decrease with iterations, a new topic-count pair may be inserted when an original zero element becomes non-zero. Thus an additional checking step is necessary after each word sampling process to examine whether new pairs should be inserted. Unfortunately, our evaluated GPUs do not support dynamical memory allocation within the GPU kernel code, and the kernel code cannot invoke the CPU code at runtime. To perform the checking step and also the GPU memory allocation, some information must be copied from the device memory to the main memory. The number of such memory copies is around N/p. The overhead of such a large number of memory copies may be expensive. Our evaluation results confirm this concern. With the newer generation GPUs, this concern may be partially or fully addressed.

3.5 Data Set Partitioning

Either no_n_{jk} or sparse storage scheme only partially addresses the memory limitation issue. To perform LDA for an arbitrary size of data set on the GPU, we consider a partitioning scheme for word tokens. The basic idea is that the GPU only holds the necessary information for a subset of word tokens.

Specifically, we divided the set of documents and the vocabulary into n and m disjoint subsets by ranges of document IDs or word IDs, respectively. Then there are totally $(n \times m)$ partitions and corresponding $(n \times m)$ computation phases in each iteration. For the $(i \times m + j)$th phase, where $i \in \{0, 1, ..., (n-1)\}$ and $j \in \{0, 1, ..., (m-1)\}$, only the word tokens belonging to the ith document subset and jth vocabulary subset are processed on the GPU, and corresponding document-topic and word-topic counts are generated on the GPU. The topic assignments set z_{ij} is processed exactly the same as the word token set x_{ij}. Through this partitioning scheme, the GPU can process an arbitrary size of data set since we can always decompose a large data set to several smaller partitions that can fit into the GPU memory.

The algorithms of generating necessary n_{jk} and n_{wk} counts are similar to Algorithm 2. The computation overhead depends on the order of partition processing. Since generating n_{wk} counts is more expensive, we process the partitions in a vocabulary-oriented style. Specifically, for the given jth vocabulary subset, the partitions $\{i \times m + j | 0 \leq i < n\}$ are processed sequentially. This way, each n partitions can share n_{wk} partitioning counts for the same word token subset. Therefore the overhead is $(1 + m)$ more scans on the original data set. We denote this global partitioning scheme as *doc-voc-part*. Partitioning schemes with $m = 1$ and $n = 1$ are denoted as *doc-part* and *voc-part*, respectively. *doc-part* and *voc-part* are suitable when the word-topic and document-topic matrix can be entirely stored in the GPU memory, respectively. For a uniform data distribution with range partitioning, the memory size requirement is estimated as M_{part} bytes:

$$M_{part} = sizeof(int) \times (\frac{2 \times N}{n \times m} + \frac{D \times K}{n} + \frac{W \times K}{m})$$

Note that the workload difference between different partitions will not hurt the overall performance significantly since these partitions are processed in different computation phases sequentially rather than in parallel on the GPU. Moreover, if the entire word token set, the topic assignment set, and all n_{jk} and n_{wk} counts can be stored in the main memory, we can directly copy the required words and counts from the main memory to the GPU memory and avoid the additional computation for the corresponding n_{jk} and n_{wk} counts generation in each iteration. This can further reduce the overhead but requires large main memory. In our implementation, we do not store n_{jk} and n_{wk} counts in the main memory thus our implementation can also be used on a PC with limited main memory.

3.6 Discussion

The three techniques we have proposed can also be combined, e.g., either no_n_{jk} or sparse storage scheme can be used together with the word token partitioning. To choose appropriate techniques, we give the following suggestions based on our experimental evaluation results: *doc-part* should be first considered if the data set and topic assignments cannot be entirely stored in the GPU memory. Then if all n_{jk} counts cannot be held in the GPU memory, no_n_{jk} or *doc-part* should be considered. Finally, the *voc-part* should be further adopted if the GPU memory is insufficient to store all n_{wk} counts, and the sparse matrix storage may be used for n_{wk}.

4 Empirical Evaluation

4.1 Experimental Setup

Table 1. Statistics of four real-world data sets.

	KOS	NIPS	NYTimes	PubMed
D	3,430	1,500	300,000	1,000,000
W	6,906	12,419	102,660	116,324
N	0.47×10^6	1.9×10^6	100×10^6	70×10^6

We conducted the experiments on a Windows XP PC with 2.4 GHz Intel Core2 Duo Quad processor, 2GB main memory and an NVIDAI GeForce GTX 280 GPU (G280). G280 has 240 1.3 GHz scalar processors, and 1GB device memory. Our GLDA is developed using NVIDIA CUDA. The CPU counterparts (CPU LDA) are based on a widely used open-source package GibbsLDA++. Note that CPU LDA is a sequential program using only one core. Due to the unavailability of the code from the previous work on GPU-based LDA [5], we could not compare our GLDA with it. However, the speedup reported by their paper is similar to ours when the data can be entirely stored on the GPU.

Four real-world data sets are used in our evaluations , which are illustrated in Table 1. We use a subset of the original PubMed data set since the main memory cannot hold the original PubMed data set. The two small data sets KOS and NIPS are used to measure the accuracy of GLDA, and the two large data sets NYTimes and PubMed are used to evaluate the efficiency. Moreover, we also have a randomly sampled subset with a size half of the original NY Times dataset (denoted as *Sub-NYTimes*) to study the overhead of different techniques when all data can be stored in the GPU memory. For each data set, we randomly split it to two subsets. 90% of the original documents are used for training, and the remaining 10% are used for test. Throughout our evaluations, the hyperparameters α and β are set to $50/K$ and 0.1, respectively [6].

We use the *perplexity* [9] to evaluate the accuracy of GLDA. Specifically, after obtaining the LDA model from 1000 iteration training, we calculate the perplexity of the test data.

4.2 Perplexity Performance

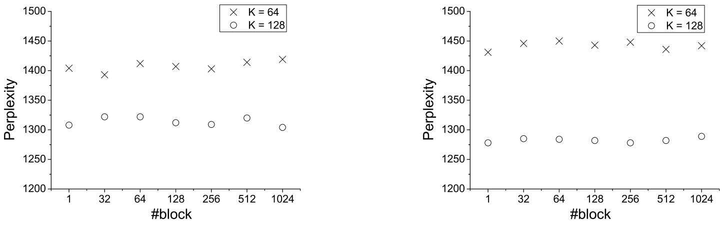
Figure 2 shows the perplexity results for the KOS and NIPS data sets with number of thread blocks varied when K=64 and K=128. Note that when there is only one thread block, the implementation is equivalent to the standard sequential LDA. Therefore, perplexity with multiple thread blocks is expected to be similar to that with one thread block. This figure shows that there is no significant difference for the perplexity results between the standard LDA model and our parallel LDA algorithm.

4.3 Memory Consumption and Efficiency

We first study the performance impact of the thread parallelism when all data is stored in the GPU using the data set Sub-NYTimes. We first set the number of threads in each thread block to 32 when varying the number of blocks. Then the number of thread block is fixed to 512 and the number of threads in each thread block is varied. Figure 3 shows the elapsed time of one iteration for the GLDA with the number of thread blocks and number of threads in each thread block varied. Compared with the CPU-LDA's performance of 10 minutes per iteration, this figure shows significant speedups when either number of thread blocks or threads per block is increased. However, it nearly maintains a constant or slightly reduced when the GPU resource is fully utilized.

http://gibbslda.sourceforge.net/

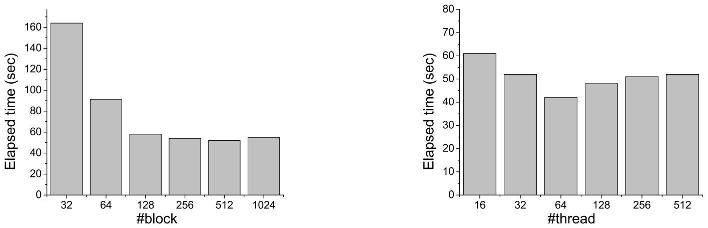
http://archive.ics.uci.edu/ml/datasets/Bag+of+Words



(a) KOS

(b) NIPS

Fig. 2. Perplexity of KOS and NIPS data sets with number of thread blocks varied.



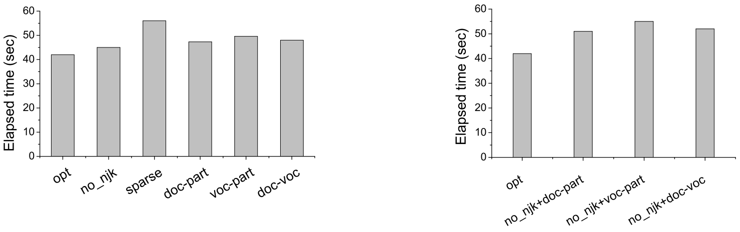
(a) Number of blocks varied.

(b) Number of threads varied.

Fig. 3. Elapsed time of one iteration on the GPU for Sub-NYTimes with the number of blocks and threads per block varied, K=256.

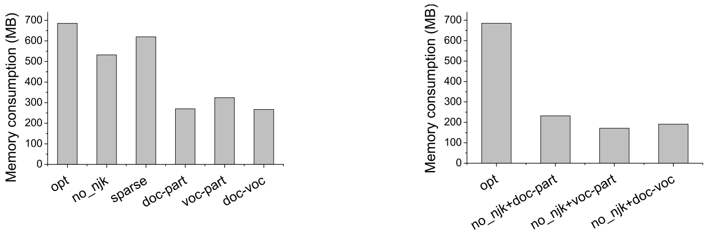
Next, we study the overhead of techniques used to address the scalability issue. We first measure the performance when only a single method is adopted. Figure 4(a) shows the elapsed time of GLDA adopting different techniques. *opt* refers to the implementation without any additional techniques and is optimized through the thread parallelism. There are four partitions for both *doc-part* and *voc-part*. *doc-voc-part* has divided both documents and vocabulary into two subsets, thus also resulting four partitions. The figure shows that the sparse storage scheme has a relatively large overhead compared with the other methods. Through our detailed study, we find the additional overhead is mainly from the large number of memory copies between the main memory and GPU memory. Moreover, *voc-part* is slightly more expensive than *doc-part* and no_n_{jk} since it needs to scan the original data set for each partition. For data sets used in our evaluations, the sparse storage scheme is always more expensive than other techniques, thus we focus on the other two techniques in the following evaluations. We further study the performance overhead when the technique no_n_{jk} and partitioning are used together. Figure 4(b) shows the elapsed time when these two techniques are combined. The combined approach is slightly more expensive than only one technique adopted since the overhead from two techniques are both kept for the combined method.

To investigate the memory space saving from different techniques, Figure 5 shows the GPU memory consumptions corresponding to Figure 4. It demonstrates that the memory requirement is reduced by around 14%-60% through various techniques. S-



(a) Adopting single technique (b) Adopting multiple techniques

Fig. 4. Elapsed time of one iteration on the GPU for Sub-NYTimes when adopting different techniques, K=256.

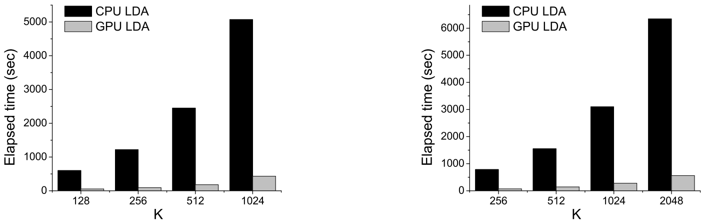


(a) Adopting single technique (b) Adopting multiple techniques

Fig. 5. Overall memory consumption on the GPU for Sub-NYTimes of adopting different techniques, K=256.

ince the number of topics is not very large and the partitioning scheme can also reduce the memory size consumed for the word token set and topic assignment set, partitioning is more effective than the other two techniques. Moreover, Figure 5(b) shows that the combined technique can further reduce the memory consumption compared with a single technique adopted.

Finally, we show the overall performance comparisons based on the NYTimes and PubMed data sets with reasonable numbers of topics. The CPU counterparts are implemented using the same set of techniques since the main memory is also insufficient to handle such data set for the original CPU implementation. The estimated memory consumptions using a traditional LDA algorithm for these two data sets are presented in Figure 1, and the actual GPU memory consumption is around 700MB for each data sets. The technique selection is based on the discussion in the implementation section. Specifically, the evaluation with NYTimes has adopted the techniques *doc-part*, while *doc-voc-part* and no_j_{jk} are both adopted for the PubMed data set. Figure 6 shows that our GLDA implementations are around 10-15x faster than their CPU counterparts. Such speedup is significant especially for large data sets. For example, suppose 1000 iterations are required for the PubMed with 2048 topics, GLDA could accelerate the computation from originally more than two months to only around 5 days.



(a) NYTimes

(b) PubMed

Fig. 6. Performance comparisons for one iteration of NYTimes and PubMed data sets with number of topics varied.

5 Conclusion

We implemented GLDA, a GPU-based LDA library that features high speed and scalability. On a single GPU with 1 GB GPU memory, we have evaluated the performance using the data sets originally requiring up to 10 GB memory successfully. Our experimental studies demonstrate that GLDA can handle such large data sets and provide a performance speedup of up to 15X on a G280 over a popular open-source LDA library on a single PC.

References

1. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. Journal of Machine Learning Research **3** (2003) 993–1022
2. Newman, D., Asuncion, A., Smyth, P., Welling, M.: Distributed inference for latent dirichlet allocation. In: NIPS. (2007)
3. Owens, J.D., Luebke, D., Govindaraju, N.K., Harris, M., Kruger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. In: Eurographics 2005, State of the Art Reports. (2005)
4. Masada, T., Hamada, T., Shibata, Y., Oguri, K.: Accelerating collapsed variational bayesian inference for latent dirichlet allocation with nvidia cuda compatible devices. In: IEA/AIE'09. (2009)
5. Yan, F., Xu, N., Qi, Y.: Parallel inference for latent dirichlet allocation on graphics processing units. In: NIPS'09. (2009) 2134–2142
6. Griffiths, T.L., Steyvers, M.: Finding scientific topics. In: Proceedings of the National Academy of Sciences (PNAS'04). (2004)
7. Chen, W.Y., Chu, J.C., Luan, J., Bai, H., Wang, Y., Chang, E.Y.: Collaborative filtering for orkut communities: discovery of user latent behavior. In: WWW'09. (2009)
8. Asuncion, A., Smyth, P., Welling, M.: Asynchronous distributed learning of topic models. In: NIPS. (2008)
9. Azzopardi, L., Girolami, M., van Risjbergen, K.: Investigating the relationship between language model perplexity and ir precision-recall measures. In: SIGIR'03. (2003) 369–370