Parallel Algorithms for Machine Learning

Dissertation

Presented in Partial Fulfillment of the Requirements
for the Degree Doctor of Philosophy
in the Graduate School of The Ohio State University

By

Gordon Euhyun Moon, B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2019

Dissertation Committee:

Prof. P. (Saday) Sadayappan, Advisor

Prof. Srinivasan Parthasarathy

Prof. Eric Fosler-Lussier

Abstract

Machine learning is becoming an integral part of everyday life. Therefore, development of a high performance genre of machine learning algorithms is becoming increasingly significant from the perspectives of performance, efficiency, and optimization. The current solution is to use machine learning frameworks such as TensorFlow, PyTorch and CNTK, which enable us to utilize specialized architectures such as multi-core CPUs, GPUs, TPUs and FPGAs. However, many machine learning frameworks facilitate high productivity, but are not designed for high performance. There is a significant gap in the performance achievable by these frameworks and the peak compute capability of the current architectures. In order for machine learning algorithms to be accelerated for large-scale data, it is essential to develop architecture-aware machine learning algorithms. Since many machine learning algorithms are very computationally demanding, parallelization has garnered considerable interest. In order to achieve high performance, data locality optimization is extremely critical, since the cost of data movement from memory is significantly higher than the cost of performing arithmetic/logic operations on current processors. However, the design and implementation of new algorithms in machine learning has been largely driven by a focus on computational complexity.

In this dissertation, the parallelization of three extensively used machine learning algorithms, Latent Dirichlet Allocation (LDA), Non-negative Matrix Factorization (NMF), and Word2Vec, is addressed by a focus on minimizing the data movement overhead through the

memory hierarchy, using techniques such as 2D-tiling and rearrangement of data computation. While developing each parallel algorithm, a systematic analysis of data access patterns and data movements of the algorithm is performed and suitable algorithmic adaptations and parallelization strategies are developed for both multi-core CPU and GPU platforms. Experimental results of the large-scale datasets demonstrate that our new parallel algorithms achieved a significant reduction of data movement from/to main memory and improved performance over existing state-of-the-art algorithms.

Dedicated to my family for their love and support.

Acknowledgments

I would like to thank my advisor, Dr. P. Sadayappan, for his steady support and advice throughout my doctoral studies. I am very fortune to have had the opportunity to work with him. The completion of this dissertation would not have been possible without his constant guidance and encouragement.

I would also like to thank my dissertation committee members, Dr. Srinivasan Parthasarathy and Dr. Eric Fosler-Lussier for their contributions to this dissertation. Dr. Parthasarathy provided constructive feedback on the LDA, NMF and Word2Vec algorithms to further deepen my understanding on latent representation learning methods. I am grateful for Dr. Fosler-Lussier for his advice on the word embedding algorithms and inspiring discussions on the latest emerging trends in Natural Language Processing field.

Additionally, I deeply appreciate Dr. Aravind Sukumaran-Rajam for his contributions throughout my doctoral program. He spent endless hours working with me despite his busy schedule and was always willing to give effective feedback and directions to move forward with the projects. Many thanks to all of my lab mates at OSU. The productive and collegial environment motivated me to work hard.

Lastly, I am blessed to be surrounded by the people I love. Thanks to my parents for their boundless support and encouragement. Words cannot express my deepest gratitude and appreciation to my wife Yoonna for her devoted love and support.

Vita

August 1, 1982 ..... Born, USA.

2011 ..... B.S.,
Computer Science and Industrial System Engineering,
Yonsei University, South Korea.

2013 ..... M.S.,
Computer Science,
Indiana University, USA.

Publications

Gordon Euhyun Moon, Denis Newman-Griffis, Jinsung Kim, Aravind Sukumaran-Rajam, Eric Fosler-Lussier, and P. Sadayappan, "Parallel Data-Local Training for Optimizing Word2Vec Embeddings for Word and Graph Embeddings," Status: Under review at a conference.

Gordon Euhyun Moon, Aravind Sukumaran-Rajam, Srinivasan Parthasarathy, and P. Sadayappan, "PL-NMF: Parallel Locality-Optimized Non-negative Matrix Factorization," *arXiv preprint arXiv:1904.07935*, 2019. Status: Under review at a journal

Gordon Euhyun Moon, Israt Nisa, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Srinivasan Parthasarathy, and P. Sadayappan, "Parallel Latent Dirichlet Allocation on GPUs," *Proceedings of the 2018 International Conference on Computational Science* (ICCS), pp. 259-272, 2018.

Gordon Euhyun Moon, Aravind Sukumaran-Rajam, and P. Sadayappan, "Parallel LDA with Over-Decomposition," *Proceedings of the 2017 IEEE 24th International Conference on High Performance Computing Workshops* (HiPCW), pp. 25-31, 2017.

Gordon Euhyun Moon and Jihun Hamm, "A Large-Scale Study in Predictability of Daily Activities and Places," *Proceedings of the 8th EAI International Conference on Mobile Computing, Applications and Services* (MobiCASE), pp. 86-97, 2016.

Fields of Study

Major Field: Computer Science and Engineering

Studies in High-Performance Machine Learning: Prof. P. (Saday) Sadayappan

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Motivation

Machine learning is becoming an integral part of everyday life, and these days the scale of data is ever-increasing at a considerable rate. The critical constraint facing the arena of machine learning today is that there have been limited attempts to deal with large-scale data volumes in the past. In order for machine learning algorithms to be adopted for large-scale data, development of a high-performance genre of machine learning algorithms is becoming increasingly significant from the perspectives of performance, efficiency, and optimization. In addition, many machine learning algorithms are very computationally demanding in time and therefore parallelization of the algorithms has garnered considerable interest. It is essential to promulgate a specific method of consilience for machine learning algorithms with high-performance computing so that they can be adjusted to play at their best across specialized architectures such as multi-core CPUs, GPUs, TPUs and FPGAs.

In order to accelerate training time of machine learning algorithms on current architectures, designing a data-aware algorithm is crucial. In practice, data-aware modifications of current state-of-the-art machine learning algorithms enhance the performance much more than directly parallelizing the original algorithms. In this dissertation, we address different strategies for data-aware machine learning algorithms to achieve high performance on

multi-core CPUs and GPUs platforms. Furthermore, technology trends have made the cost of data movement increasingly dominant, both in terms of energy and time, over the cost of performing arithmetic operations across nodes in a multi-node distributed-memory system and within the memory hierarchy within a node. However, the design and implementation of new algorithms in machine learning has been largely driven by a focus on computational complexity. The core computation of many machine learning algorithms involves a large number of dot-product operations. The dot-product computation is inherently memory-bandwidth limited because only two floating point operations are performed per pair of data elements read from memory. Since the peak floating-point processing rates of all current/emerging processors have increased much more rapidly than peak data movement throughputs (between nodes or within the memory hierarchy of a node), minimization of data movement overheads is increasingly critical. In order to achieve high performance, the achieved operational intensity (ratio of arithmetic/logic operations executed to the total volume of data moved) must be greater than the machine balance parameter (ratio of peak floating-point rate to peak data movement bandwidth) of the computer system. While machine learning algorithms in general can differ greatly in their amenability to achieving a high operational intensity through an effective implementation, certain machine learning algorithms have an inherently unfavorable data access pattern that makes it impossible to achieve a sufficiently high operational intensity for even an optimal implementation on a given target platform. However, previous studies in the machine learning area have not yet sufficiently addressed the minimization of data movement overhead through the memory hierarchy, using techniques such as matrix tiling and rearrangement of data computation.

1.2 Overview

In this dissertation, we explore the parallelization of three state-of-the-art machine learning algorithms, i.e., Latent Dirichlet Allocation, Non-negative Matrix Factorization, and Word2Vec, on various parallel platforms. The goal is to enable acceleration of various machine learning algorithms in applications in which they might be constrained by their excessive training time. As we aim to improve parallel machine learning algorithms in comparison to currently available state-of-the-art algorithms, each parallelization is examined by performing an in-depth theoretical analysis of data access patterns and data movements of the algorithms pertaining to achievable operational intensity. Efficient realizations of the parallel algorithms on multi-core CPUs and GPU platforms are developed, demonstrating significant reduction of data movement from/to main memory and improved performance over current state-of-the-art parallel implementations. To elaborate in detail, the contributions of this dissertation lie in mainly three folds: Parallel Topic Modeling, Parallel Locality-Optimized Non-negative Matrix Factorization, Parallel Data-Locality Training for Optimizing Word2Vec Embedding Model.

Parallel Topic Modeling: Latent Dirichlet Allocation (LDA) proposed by Blei et al. [7] is a statistical technique for topic modeling. Prior efforts to parallelize LDA have either used expensive atomic operations or weakened the sampling model to enable parallelization without heavy use of atomics [64, 98, 102]. In this work, we present a parallel LDA implementation on multi-core CPUs that uses an over-decomposed 2D tiling strategy to overcome the limitations of previous parallelization schemes. An alternate implementation that uses some approximation is also presented that fully avoids use of atomic operations. Furthermore, 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 multi-core CPUs and GPUs.

Parallel Locality-Optimized Non-negative Matrix Factorization: Non-negative Matrix Factorization (NMF) proposed by Lee and Seung [44] is a key kernel for unsupervised dimension reduction used in a wide range of applications, including topic modeling [85], recommender systems [33] and bioinformatics [58, 97, 104]. Due to the compute intensive nature of applications that must perform repeated NMF, several parallel implementations have been developed in the past. However, existing parallel NMF algorithms have not addressed data locality optimizations, which are critical for high performance since data movement costs greatly exceed the cost of arithmetic/logic operations on current computer systems. In this work, we devise a parallel NMF algorithm based on the Hierarchical Alternating Least Squares (HALS) scheme that incorporates algorithmic transformations to enhance data locality. Efficient realizations of the algorithm on multi-core CPUs and GPUs are developed, demonstrating significant performance improvement over existing state-of-the-art parallel NMF algorithms.

Parallel Data-Locality Training for Optimizing Word2Vec Embedding Model: The Word2Vec model proposed by Mikolov et al. [59,60] is a neural network-based unsupervised word embedding technique widely used in applications such as natural language processing [63, 67, 106], bioinformatics [61] and graph mining [26, 69]. As Word2Vec repeatedly performs Stochastic Gradient Descent (SGD) to minimize the objective function, it is very compute-intensive. However, existing methods for parallelizing Word2Vec are not optimized enough for data locality to achieve high performance. In this work, we develop a parallel data-locality-enhanced Word2Vec algorithm based on Skip-gram with a novel negative sampling method that decouples loss calculation with positive and negative samples; this

allows us to efficiently reformulate matrix-matrix operations for the negative samples over the sentence. Experimental results demonstrate our parallel implementations on multi-core CPUs and GPUs achieve significant performance improvement over the existing state-of-the-art parallel Word2Vec implementations while maintaining evaluation quality. We also show the utility of our Word2Vec implementation within the Node2Vec algorithm [26] which accelerates embedding learning for large graphs.

1.3 Key Contributions

The main contributions are presented in three chapters and organized as follows:

- Chapter 2 presents new parallel LDA algorithms on multi-core CPUs and GPUs. As for multi-core CPUs, we present a parallel LDA that adopts 2D-tiling and over-decomposition techniques, as well as the alternative fully atomics-free scheme. Furthermore, we propose an approximated LDA algorithm with mini-batch processing on GPUs. We compare our parallel approaches with existing state-of-the-art GPU implementations.

- Chapter 3 presents new parallel NMF algorithms on multi-core CPUs and GPUs based on the HALS scheme. The associativity of addition is utilized to judiciously reorder additive contributions in updating elements of factorized matrices, to enable matrix tiling of a computationally intensive component of the algorithm. We develop a function of tile size which leads to a model for selection of effective tile size. An experimental evaluation with datasets used in previous studies demonstrates significant performance improvement over state-of-the-art alternatives available for parallel NMF.

- Chapter 4 presents new parallel data-locality-enhanced Word2Vec algorithms on multi-core CPUs and GPUs based on Skip-gram with a novel negative sampling method. The main bottleneck of the original Word2Vec algorithm is identified by conducting data movement analysis. In order to increase data reuse involving negative sampling method, the operations required for negative samples are reformulated with efficient matrix-matrix multiplications. An extensive comparative evaluation of the new algorithms with a number of state-of-the-art implementations of Word2Vec are performed on multi-core CPUs and GPUs platforms. The utility of our Word2Vec implementation is also presented within the Node2Vec node embedding algorithm.

Chapter 2: Parallel Latent Dirichlet Allocation on Multi-Core CPUs and GPUs

2.1 Introduction

Topic modeling is a technique for finding the latent topics that occur in a collection of documents. Each document may have multiple topics and the words included in the document may have different topics. The latent topics generated by topic modeling technique can be interpreted as clusters of similar words and documents. Latent Dirichlet Allocation (LDA) is a powerful technique for topic modeling originally developed by Blei et al. [7]. 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 fully sequential LDA algorithm of Griffiths et al. [24] 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 [101], distributed-memory systems [64, 98], and GPUs (Graphical Processing Units) [55, 95, 102, 103, 111]. Many previous efforts at parallelizing LDA have sought to relax CGS and use approximations because strict CGS imposes constraints on efficient parallelization. One approach that has been explored [64] is to divide up the documents among

parallel threads/processes, so that the document-to-topic array is disjointly partitioned and exclusively read/written by only the owning thread/process. However, since common words are present in documents processed by different threads/processes, it is impossible to also disjointly partition and access the word-to-topic distribution matrix across threads/processes. In a shared-memory parallel environment, a solution is to use atomic update operations on word-to-topic distribution matrix, but the overheads can be high. A second alternative that has been considered is to perform uncollapsed Gibbs sampling [94, 95], by using an independent copy of three key data structures for each thread/process and only updating the local copy, thereby avoiding the need to use atomic operations. At the end of an iteration, the local copies of three key data structures are merged to sample new parameters of document-to-topic and word-to-topic distributions, and then redistributed to the threads/processes before the start of the next iteration. However, the use of uncollapsed sampling leads to slower convergence [64].

In this chapter, we develop a 2D tiled approach to perform efficient parallel LDA with CGS for multi-core CPUs. We first develop a parallel algorithm that avoids the need for any atomic operations on document-to-topic/word-to-topic matrices, but still requires atomic operations for the topic-count array. Load-balancing is a significant issue since different documents can have widely varying word counts. We use over-decomposition as a means of addressing the problem of load imbalance. We also devise an alternate atomics-free approach that introduces an approximation by maintaining local copies for topic-count array. The performance and quality of solution for the two approaches are compared.

In developing a parallel approach to LDA for GPUs, algorithmic degrees of freedom can be judiciously matched with inherent architectural characteristics of the target platform. In this chapter, 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. We perform a systematic exploration of the space of partially collapsed Gibbs sampling strategies by carrying out two tasks. First, we empirically characterize the impact on convergence and perplexity of different sampling variants. Second, we analyze 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. Experiments on large-scale datasets show the effectiveness of the new parallel implementation on both multi-core CPUs and GPUs platforms.

Chapter 2 is organized as follows. Section 2.2 and 2.3 provide some background information on LDA and related prior works. Section 2.4 presents the parallel LDA algorithms that uses 2D-tiling and over-decomposition, as well as the alternative fully atomics-free scheme. Then we present the high-level overview of our new mini-batched LDA algorithm for multi-core CPUs and GPUs. Section 2.5 details our new algorithm for GPUs.

2.2 Background

2.2.1 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is an effective approach to topic modeling. It is used for identifying latent topics distributions for collections of text documents [7]. 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 $\phi_{w|z_{ij}}$. The commonly used notations for LDA algorithms in this chapter are described in Table 2.1.

Table 2.1: Common notations for LDA algorithms

Notation	Description
$DATA$	Document-term matrix
DT	Document-topic count matrix
WT	Word-topic count matrix
NT	Topic-count vector
Z	Topic assignment matrix
θ	Document-topic probability distribution matrix
ϕ	Word-topic probability distribution matrix
D	Number of documents in the corpus
W	Number of words in the corpus
V	Number of distinct words in the corpus
K	Number of topics
P	Number of partitions
B	Number of documents in each mini-batch
α, β	Dirichlet priors

Asuncion et al. [3] succinctly describe various inference techniques, and their similarities and differences for state-of-the-art LDA algorithms. In context of our work, we first discuss two main variants, viz., *Collapsed Gibbs Sampling (CGS)* and *Uncollapsed Gibbs Sampling (UCGS)*. In Section 2.3, we then proceed to discuss about various efficient implementations of these schemes developed over the years in modern computation platforms.

2.2.2 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 [24, 64, 103, 108]. 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:

$$P(z_{ij} = k|\mathbf{z}^{\neg ij}, \mathbf{x}, \alpha, \beta) \propto \frac{WT^{\neg ij}_{x_{ij}|k} + \beta}{NT^{\neg ij}_{k} + V\beta}(DT^{\neg ij}_{j|k} + \alpha) \tag{2.1}$$

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. The CGS based LDA algorithm performs a number of passes through the collection of documents. In each pass, each word of each document is processed. Three key data structures, WT, DT and NT, are read and written as each word is processed. Each word in each document is initially assigned randomly to one of the K latent topics, and as the iterations proceed, the word-to-topic and document-to-topic distributions in arrays WT and DT and the topic distribution array NT converge. When a word in a document is processed, some elements of all three arrays are read and one element of each array is modified. Thus, as a later word is processed, the modified distribution

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

repeat
  for document = 0 to D − 1 **do**
    L ← document_length
    for word = 0 to L − 1 **do**
      current_word ← *DATA*[document][word]
      old_topic ← Z[document][word]
      decrement WT[current_word][old_topic]
      decrement NT[old_topic]
      decrement DT[document][old_topic]
      sum ← 0
      for k = 0 to K − 1 **do**
        sum ← sum + $\frac{WT[current\_word][k]+\beta}{NT[k]+V\beta}$ $(DT[document][k]+\alpha)$
        p[k] ← sum
      end for
      U ← random_uniform() × sum
      for new_topic = 0 to K − 1 **do**
        if U < p[new_topic] **then**
          break
        end if
      end for
      increment WT[current_word][new_topic]
      increment NT[new_topic]
      increment DT[document][new_topic]
      Z[document][word] ← new_topic
    end for
  end for
until convergence

reflecting the processing of all previous words is used. This process of using the updated distributions is called Collapsed Gibbs sampling (CGS).

2.2.3 Uncollapsed Gibbs Sampling

The use of Uncollapsed Gibbs Sampling (UCGS) as an alternate inference algorithm for LDA is also common [94, 95]. Unlike CGS, UCGS requires the use of two additional parameters θ and ϕ to draw latent variable z as follows:

$$P(z_{ij} = k|\mathbf{x}) \propto \phi_{x_{ij}|k}\theta_{j|k} \tag{2.2}$$

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 [64, 95]. However, the use of UCGS facilitates a more straightforward parallelization of LDA. Algorithm 2 shows the sequential UCGS based LDA algorithm.

Algorithm 2 Sequential UCGS 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: θ: document-topic distribution, ϕ: word-topic distribution, DT: document-topic count matrix, WT: word-topic count matrix, NT: topic-count vector

```
repeat
    Initialize DT, WT and NT to zero
    for document = 0 to D − 1 do
        L ← document_length
        for word = 0 to L − 1 do
            current_word ← DATA[document][word]
            sum ← 0
            for k = 0 to K − 1 do
                sum ← sum + φ[current_word][k] θ[document][k]
                p[k] ← sum
            end for
            U ← random_uniform() × sum
            for new_topic = 0 to K − 1 do
                if U < p[new_topic] then
                    break
                end if
            end for
            increment DT[document][new_topic]
            increment WT[current_word][new_topic]
            increment NT[new_topic]
        end for
    end for
    Sample new θ and φ based on current DT, WT and NT
until convergence
```

2.3 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 CPUs as well as many-core GPUs platforms.

2.3.1 Multi-Core CPU Platform

Newman et al. [64] justifies the importance of distributed algorithms for LDA for large scale datasets and proposed an *Approximate 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 pre-assigned 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. [98] 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. [72] 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. [88] proposes approximation schemes for CGS based LDA in the distributed computing paradigm for efficient sampling with competitive accuracy. Xiao et al. [101] proposes a dynamic

adaptive sampling technique for CGS with strong theoretical guarantees and efficient parallel implementation. Most of these works suffer from memory overhead and synchronization bottleneck due to multiple local copies of global data-structures which are later used for synchronization across processors. In practice, a global synchronization has the problem of load imbalance since each document in different processors contains different number of word tokens. Therefore, the processor which has a document with small number of word tokens has to be wait for the other processors which have relatively large number of word tokens.

Further, all prior parallel implementation of LDA have either used expensive atomic operations to ensure algorithmic accuracy of CGS or have deviated from CGS by creating local copies of WT and/or DT matrices to avoid using atomic for updates [64, 98, 102]. Unlike these prior efforts, the parallel LDA algorithm we describe in the next chapter maintains CGS without atomic operations to update the WT or DT matrices. This is achieved by partitioning the set of documents as well as the vocabulary into disjoint subsets, as explained in the next chapter. We show how CGS can be parallelized efficiently on multi-cores, thus maintaining a higher convergence rate of CGS.

2.3.2 Many-Core GPU Platform

One of the first GPU based implementations using CGS is developed by Yan et al. [103]. 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. [55] 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. [95] introduces a variant of UCGS technique which is embarrassingly parallel with competitive performance. Zhao et al. [111] proposes a state-of-the-art GPU implementation which combines the SAME (State Augmentation for Marginal Estimation) technique with mini-batch processing.

2.4 Parallel Latent Dirichlet Allocation on Multi-Core CPUs

2.4.1 Parallel LDA with Over-Decomposition

Our parallel CPU implementation of LDA is based on the CGS approach. A naive parallelization approach will lead to very high perplexity (very low log-likelihood) as the sequential constraints will not be respected. As with prior approaches, the set of documents is disjointly partitioned for processing by different processors. The main difference is in the approach to handling conflicts in processing the word-to-topic count matrix. For example, consider a parallelization approach where the set of documents are distributed across different threads. Since multiple documents can have the same words, two threads could simultaneously try to read/update the WT matrix, leading to a race condition. While atomic operations can be used to overcome the race condition, they can be expensive. Moreover, the number of atomic reads required are high as it is performed in the inner most loop of the kernel. Alternatively, if the parallelization was across the words, then the read/update to the DT matrix would lead to a race condition. In either case, there is an additional atomic update required on the NT vector.

Our parallel implementation is based on 2D tiling of the DT matrix and is designed to considerably reduce the required number of atomic operations. The main idea is to partition

both the collection of documents as well as the active vocabulary in the corpus into P disjoint sets, and reorganize the sampling over words in documents along diagonals in a 2D tiling of the document-word space, so as to avoid any conflicts between processors in modifying either the WT or DT matrices.

2.4.1.1 Data Partitioning

One of the goal of our parallel LDA is to improve load-balance on multiple processors by distributing the data based on the even (same) range of the indices of words and documents. Figure 2.1 illustrates the data/work partitioning scheme. Given the dataset, we first mapped each word token into their unique word index and sorted the work tokens in each document ordered by word index. Then the entire vocabulary (set of all words across all documents) is divided into P disjoint partitions. Similarly, the documents are also divided into P partitions. Thus, the entire document-word matrix is divided into $P \times P$ partitions. For example, words 0 to 9 in documents 0 to 3 form partition 0. Similarly, words 10 to 19 in documents 0 to 3 form partition 1. Assume that document 0 contains words {1, 2, 30, 44}, document 1 contains words {4, 9, 99}, document 2 contains words {50, 60} and document 3 contains words {5, 24, 30}. Partition 0 will consist of words {1, 2} in document 0, words {4, 9} in document 1, and word {5} in document 3. In other words, every block has the same number of grouped documents and the words within the same range of word indices. This partition scheme guarantees that the same words in different documents do not have memory read/write conflicts simultaneously on both WT and DT.

2.4.1.2 Algorithm

In Figure 2.1, partitions which have mutually exclusive words and documents are marked with the same color and shading pattern. For example, consider partitions 0, 5, 10 and 15

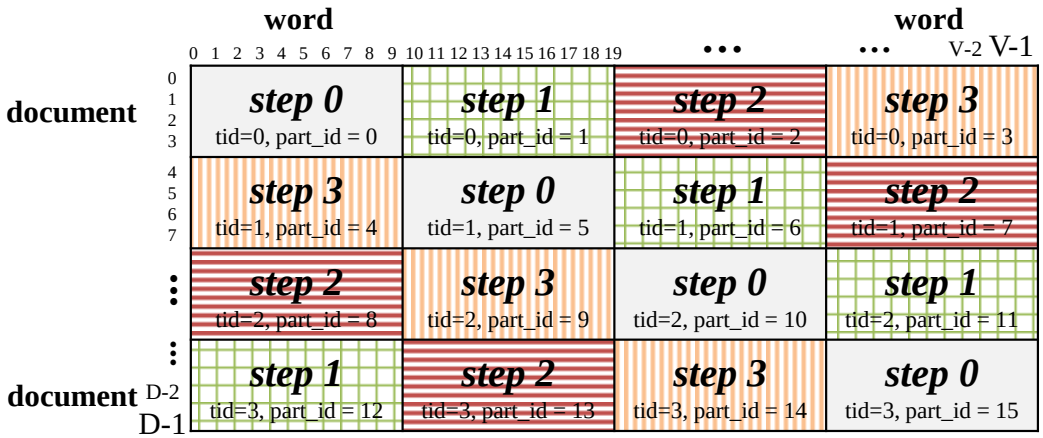


Figure 2.1: Data partitioning scheme, if $P = 4$

which are along the diagonal of the partitioned document-word matrix. The set of words in each of these partitions is mutually exclusive. The documents in these partitions are also mutually exclusive. Hence, these partitions can be processed in parallel in a race-free manner without the need for any expensive atomic operations on the WT and DT arrays. Our parallelization approach is based on this scheme. All the partitions processed in parallel in the same step are distinct. If there are $P \times P$ partitions, then there will be P steps, each processing partitions along a wrap-around diagonal in the set of 2D data-tiles. Algorithm 3 shows the parallel LDA algorithm. In Line 1, the number of documents in each partition is computed, and in Line 2, the number of words in each partition is computed. Line 4 iterates over the number of steps. Each step corresponds to processing mutually exclusive partitions in parallel (Line 5). In Lines 6 to 9, each thread determines the set of documents and words that it will process in the current step. In Lines 10 to 33, we compute and update the WT, DT, and NT. The only atomic operations needed are in processing the NT vector.

Algorithm 3 Parallel LDA with Over-Decomposition

Input: *DATA*: D documents and **x** word tokens in each document, V: vocabulary size, K: number of topics, α, β: hyper-parameters, P: number of partitions

Output: DT: document-topic count matrix, WT: word-topic count matrix, NT: topic-count vector, Z: topic assignment matrix

doc_part_size ← ceil(D / P)
voc_part_size ← ceil(V / P)
repeat
  for $step = 0$ to $P-1$ **do**
    for $part\_id = 0$ to $P-1$ **in parallel do**
      r_start ← part_id × doc_part_size
      r_end ← min(r_start + doc_part_size, $|D|$)
      c_start ← (part_id + step) % (P × voc_part_size)
      c_end ← min(c_start + voc_part_size, $|V|$)
      for $doc = r\_start$ to $r\_end - 1$ **do**
        for $word = c\_start$ to $c\_end - 1$ **do**
          cur_word ← *DATA_PREP*[doc][word]
          old_topic ← Z[doc][word]
          decrement DT[doc][old_topic]
          decrement WT[cur_word][old_topic]
          atomic decrement NT[old_topic]
          sum ← 0
          for $k = 0$ to $K-1$ **do**
            sum ← sum + $\frac{N[cur\_word][k]+\beta}{NT[k]+V\beta}$ $(M[doc][k]+\alpha)$
            p[k] ← sum
          end for
          U ← rand_uniform() × sum
          for $k = 0$ to $K-1$ **do**
            if $U < p[k]$ **then**
              break
            end if
          end for
          increment DT[doc][k]
          increment WT[cur_word][k]
          atomic increment NT[k]
          Z[doc][word] ← k
        end for
      end for
    end for
  end for
until convergence

2.4.1.3 Load Balancing

An important issue to be addressed is load imbalance. Each document only uses a subset of words in the vocabulary. Hence, in each 2D partition of the document-word matrix, the number of words can be quite non-uniform. It is even possible that some partitions are empty. Note that the amount of work in each partition is proportional to the number of words.

Consider Figure 2.1, where the document-word matrix is divided into 4×4 partitions and the number of threads is 4. Consider step 0, in which partitions 0, 5, 10, and 15 are processed by threads 0, 1, 2, and 3, respectively. Assume that the number of words in partition 0, 5, 10, and 15 are 100, 0, 10, 15, respectively. Thread 0 has much more work when compared to other threads. Hence, threads 1, 2, and 3 will be idle as they wait for thread 0 to complete. Note that there is an implicit barrier synchronization at the end of each step.

In order to achieve load balancing, we use over-decomposition. Instead of dividing the document-word matrix into $P \times P$ partitions, where P is the number of threads, we divide the document-word matrix into $(O \times P) \times (O \times P)$ partitions, where O is the over-decomposition factor. Since there is no synchronization within each step, the threads can dynamically acquire a partition. For example, in Figure 2.2, threads 0, 1, 2, and 3 initially process partitions 5, 18, 31, and 44, respectively. Thread 1 and thread 2 only have 10 units of work, compared to 100 units for thread 0 and 35 for thread 3. Therefore thread 1 and 2 will complete partitions 18 and 31 and then process partition 57 and 70, respectively.

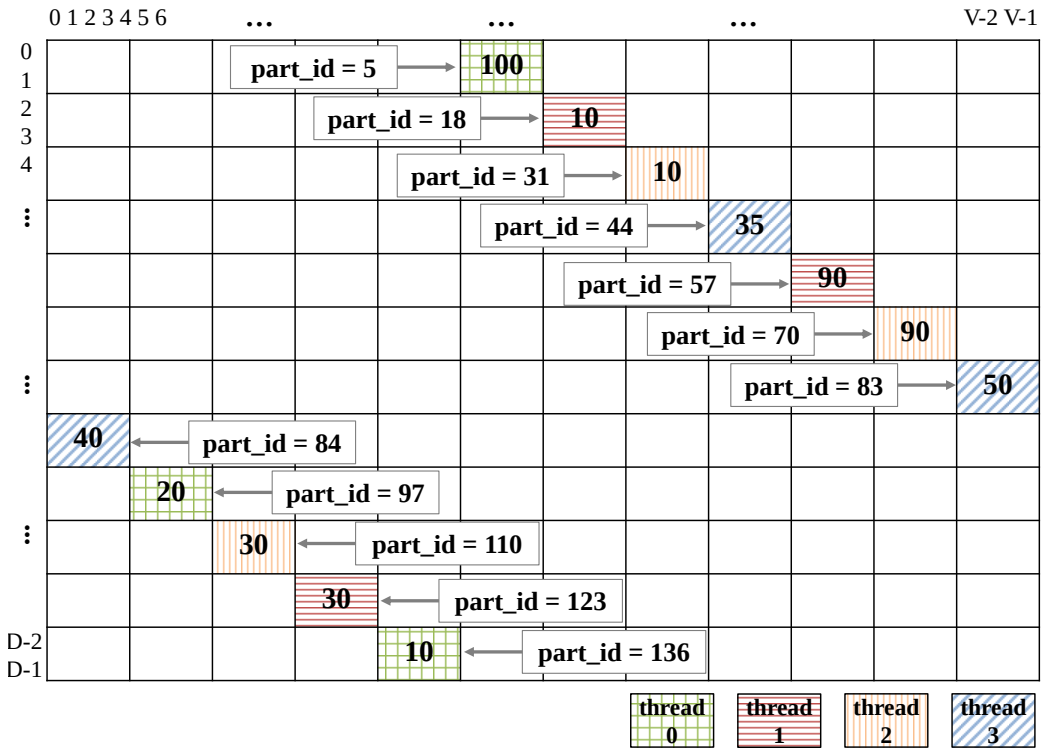


Figure 2.2: Distribution of threads in a step via over-decomposition, for 12 partitions and 4 threads. The number in each data block indicates the number of word tokens.

2.4.1.4 Alternate Atomics-free Scheme

The previously presented approach requires atomic operations to update the NT vector. In this Section 2.4.1.4, we provide an alternate scheme that does not require any atomic operations. The core idea behind the new scheme is to use a thread-local *delta* vector to keep track of the local changes to the NT vector. The global state of topic-count is maintained by the global NT vector. When a thread requires a particular topic's count, it first reads the corresponding global NT vector value and adds the corresponding value in its local *delta* vector. When a thread increments/decrements the NT vector, the increment/decrement operation is simply performed on the local *delta* vector. Since the updates are made locally,

atomic operations are not required. The entire set of *delta* vectors can be viewed as matrix

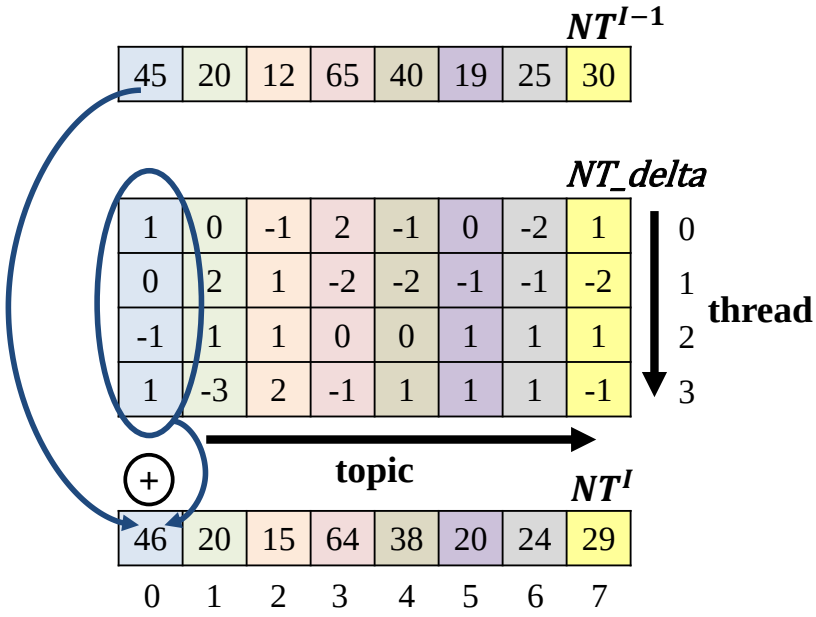


Figure 2.3: Reduction in alternate atomics-free scheme

(*NT_delta* matrix). Figure 2.3 shows the *NT_delta* matrix. The number of rows is equal to the number of threads and the number of columns is equal to the number of topics.

At the beginning of each outer iteration, the *NT_delta* matrix is initialized to zero. In Algorithm 3, atomic operations are used to increment (in Line 30) and decrement (in Line 16) the *NT* vector. Instead, in the new approach, the increment is done as: *NT_delta*[tid][topic] $+= 1$, and decrement operation is done similarly. In the new scheme, a thread reads the *NT* vector (equivalent of Line 19) as follows: *NT_delta*[tid][topic] + *NT*[topic]. Thus, there are no atomic operations for reads or writes.

At the end of each iteration, the sum of each column of *NT_delta* is added to the corresponding column of the *NT* vector. In order to perform this operation, the columns of

the NT_delta matrix are partitioned into column panels and each column panel is processed by a single thread. Thus, the latter reduction step can also be performed in parallel and without atomic operations.

2.4.1.5 Experimental Evaluation

We evaluate the new parallel LDA variants using two common datasets: the NIPS and NYTimes datsets from the UCI Machine Learning Repository [50]. Documents that do not contain any word token are discarded. Table 2.2 describes the characteristics of the datasets. We use 90% of the documents as a training set, and the remaining 10% as the test set and

Table 2.2: Statistics of datasets used in the experiments. 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
NIPS	1,500	1,932,365	12,375
NYTimes	299,752	99,542,125	101,636

used cross-validation evaluation [40]. As suggested in [24], the α and β are set to $50.0/K$ and 0.1, respectively. We use GibbsLDA++ [71], which is a standard C++ implementation of sequential LDA with CGS, to compare the performance and accuracy of the parallel OD-LDA implementation. We first present an evaluation of the base algorithm and then discuss the fully atomics-free alternate scheme. The LDA models we use in our experiments are:

- Sequential collapsed: sequential GibbsLDA++ with CGS
- Parallel OD-LDA: parallel LDA with Over-Decomposition

We evaluate performance on an 8-core Intel Xeon CPU. Table 2.3 shows the details of the benchmarking machine.

Table 2.3: Machine details

Hardware	Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz (8 cores) 16 GB RAM (1866 MHz)
Software	Red Hat Enterprise Linux Server release 6.7 GCC version 4.9.2

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.

$$log(p(\mathbf{x}^{test})) = \prod_{ij} log \sum_{k} \frac{WT_{w|k} + \beta}{\sum_{w} WT_{w|k} + V\beta} \frac{DT_{j|k} + \alpha}{\sum_{k} DT_{j|k} + K\alpha} \tag{2.3}$$

$$\text{per-word log-likelihood} = \frac{1}{W^{test}} log(p(\mathbf{x}^{test})) \tag{2.4}$$

where W^{test} is the total number of word tokens in the test set.

In Figure 2.4, the convergence rates of sequential and parallel CGS are approximately the same at the same number of iterations, implying that the quality of our parallel implementation maintains closer to the sequential implementation. Figure 2.5 shows the performance variation when the number of partitions is varied for the NIPS dataset. The parallel LDA with 100 partitions converges 2.8$\times$ faster than 8 partitions because of better load balance. Intuitively, it is clear that smaller partitions used in parallel model definitely required each data block to process relatively bigger number of documents and word tokens to complete CGS procedure. Table 2.4 compares our approach with the sequential approach. We measure the elapsed time per iteration to evaluate speedup of the parallel algorithm. The execution time of the baseline sequential collapsed model is similar to the parallel OD-LDA model

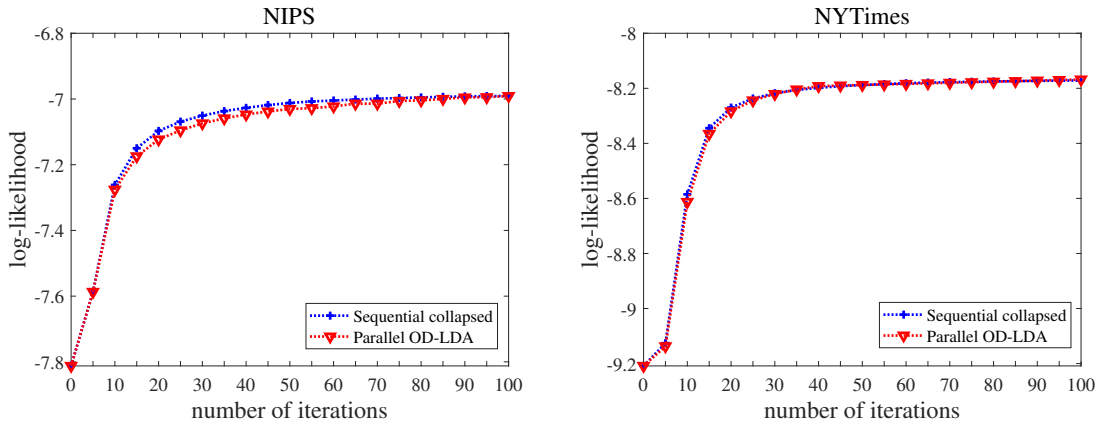


Figure 2.4: Comparison of convergence over iterations on NIPS and NYTimes datasets, K=128. For parallel OD-LDA, the number of partitions and threads are set to 100 and 4, respectively. X-axis is the number of iterations and y-axis is per word log-likelihood.

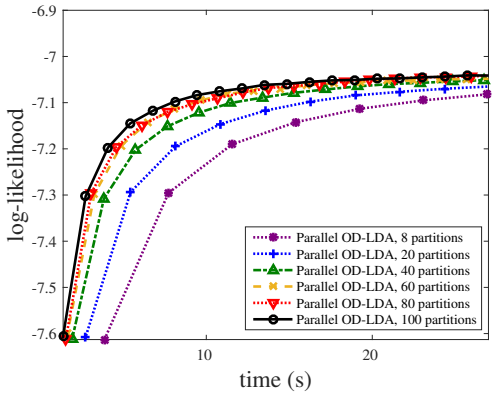


Figure 2.5: Convergence over time across different number of partitions on NIPS dataset, K=128, 4 threads.

with 1 thread. As expected, the execution time decreases as the number of threads increases. With 8 threads, the parallel LDA scheme achieved 4.1$\times$ speedup on the NIPS dataset and 4.9$\times$ speedup on the NYTimes dataset. Figure 2.6 compares the log-likelihood versus time

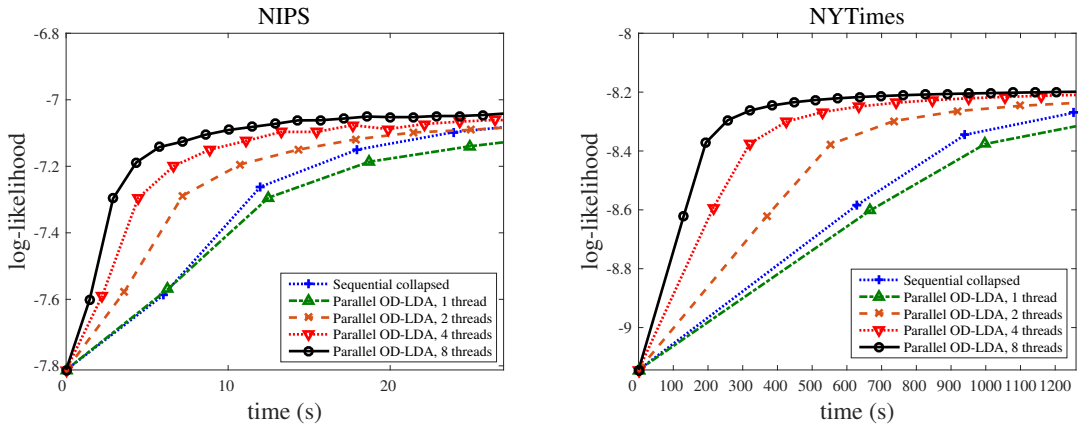


Figure 2.6: Convergence over time across 1, 2, 4 and 8 threads, on NIPS and NYTimes datasets, K=128. The number of partitions is set to 300 for both NIPS and NYTimes datasets. X-axis: elapsed time in seconds; Y-axis: per-word log-likelihood.

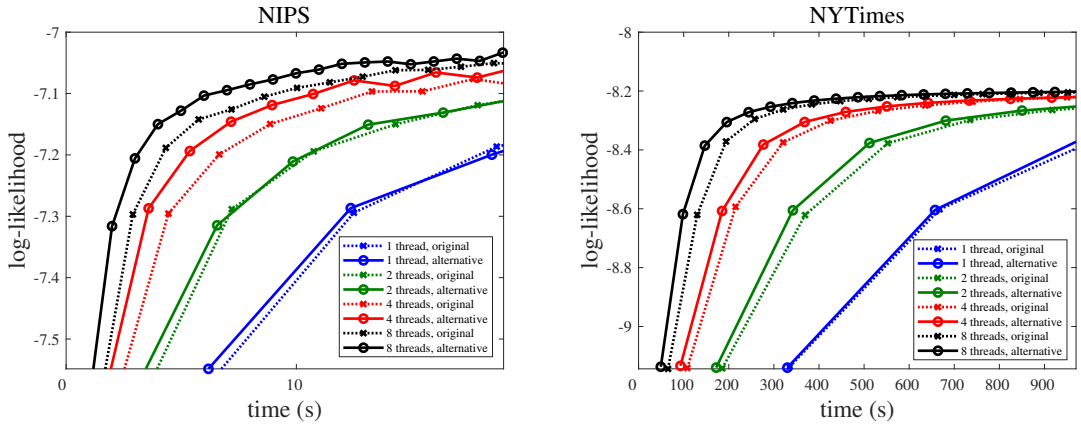


Figure 2.7: Comparison between alternate atomics-free scheme and the original scheme

for various approaches. The parallel approach converges much faster than the sequential version while maintaining similar log-likelihood. Finally, Figure 2.7 shows performance comparison of the alternate atomics-free scheme and original scheme. It can be seen that the alternative scheme takes less time for convergence.

Table 2.4: Comparison of the elapsed time in second per iteration on NIPS and NYtimes datasets, K=128 and P=300 on NIPS and NYTimes dataset.

	Number of threads	**NIPS** Elapsed time per iteration (s)	**NYTimes** Elapsed time per iteration (s)
Sequential collapsed	None	1.1929	62.6343
Parallel OD-LDA	1	1.2406	66.2232
Parallel OD-LDA	2	0.7132	36.5454
Parallel OD-LDA	4	0.4421	21.0296
Parallel OD-LDA	8	0.2857	12.6711

2.4.2 Parallel LDA with Mini-Batch Processing

2.4.2.1 Overview of Mini-Batch Processing

As seen in Algorithm 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 multi-core CPUs and 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 with mini-batch processing:

- 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 multi-core CPUs, taking orders of magnitude more time than arithmetic operations.

- Delayed updates requires additional temporary storage to hold information about the updates to be performed at the end of a mini-batch.

- The basic formulation of CGS requires an expensive division operation (Equation 2.1) 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 mini-batch can be scaled by the division operation by means of the denominator term in Equation 2.1 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 2.8, 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 parallelization. On the other hand, delayed update scheme is fully parallel but suffers from a lower

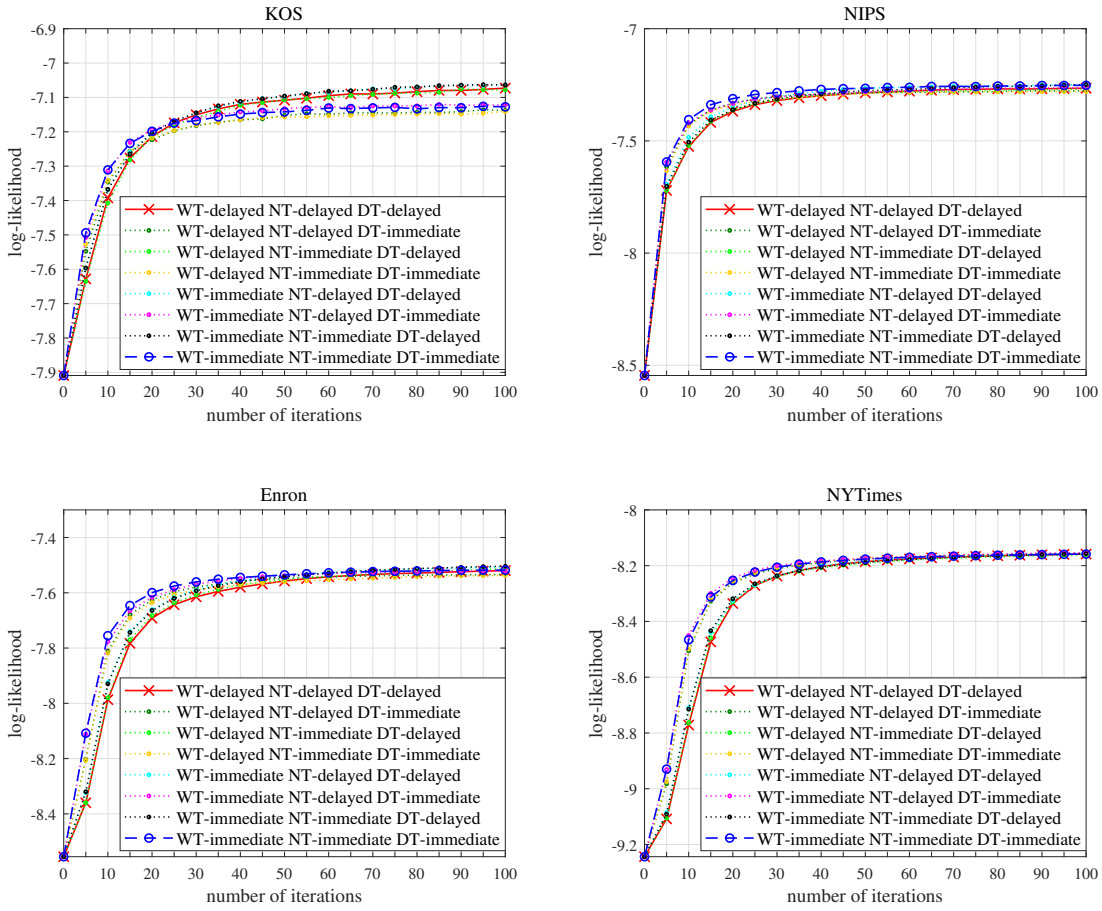


Figure 2.8: Convergence over number of iterations on KOS, NIPS, Enron and NYTimes datasets. The mini-batch sizes are set to 330, 140, 3750 and 28125 for KOS, NIPS, Enron and NYTimes, respectively. X-axis: number of iterations; Y-axis: per-word log-likelihood on test set.

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 processing, 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.

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.

2.4.2.2 Details of Mini-Batched Lock-Free Algorithm

As shown in Figure 2.8, fully UCGS (red line) converges slower than fully CGS (blue line), but the convergence gap between UCGS and CGS is not huge. Since fully UCGS algorithm can be trained with completely in parallel, we adapted lock-free inference algorithm that updates DT, WT and NT at the end of each mini-batch processing. Algorithm 4 shows our parallel LDA algorithm with mini-batch processing. In Line 1, the total number of mini-batches M is computed given the total number of documents D and each mini-batch size B. The parallelism is in each mini-batch across the documents.

Within each mini-batch, we use *TOPICS_delta* array to keep track of the old topic assigned to current word token before sampling new topic, and new topic assigned to current word token after sampling. Additional permuted order array, *PERM* is necessary to point the indices where the topic assignments of current word token have to be stored in the *TOPICS_delta* array. In *PERM* array, the word indices in *TOPICS_delta* array can be directly sorted in ascending order to easily group the same word indices in each mini-batch. Algorithm 6 shows the pre-processing procedure of *PERM* array. Therefore, the use of *TOPICS_delta* array enables us to achieve fully lock-free algorithm for updating key data structures. Based on the values in *TOPICS_delta* array, new DT, WT and NT are generated to use it in the next mini-batch. Another advantage of this scheme is that the complexity of computation on conditional distribution can be reduced through removing a division operation in Line 12 in Algorithm 1. Since DT is document-specific matrix, we can pre-compute normalized DNT at the end of each mini-batch processing as shown in Line 20

Algorithm 4 Parallel Mini-batched Gibbs Sampling for Multi-Core CPUs

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

```
M ← ceil(D / B)
repeat
  for minibatch_id = 0 to M − 1 do
    for document = 0 to B − 1 in parallel do
      current_document ← document + (B × minibatch_id)
      L ← current_document_length
      for word = 0 to L − 1 do
        current_word ← DATA[current_document][word]
        old_topic ← Z[current_document][word]
        TOPICS_delta[PERM[minibatch_id][document][word]][0] ← old_topic
        new_topic ← sampling_without_update()
        TOPICS_delta[PERM[minibatch_id][document][word]][1] ← new_topic
        Z[current_document][word] ← new_topic
      end for
    end for
    update WT based on TOPICS_deta
    update NT based on TOPICS_deta
    memclear DT
    update DT based on Z
    update DNT
  end for
until convergence
```

Algorithm 5 Sampling without updating parameters

```
sum ← 0
for k = 0 to K − 1 do
  sum += (WT[curr_word][k]+β)×DNT[curr_doc][k]
  p[k] ← sum
end for
U ← random_uniform() × sum
for new_topic = 0 to K − 1 do
  if U < p[new_topic] then
    break
  end if
end for
```

Algorithm 6 Pre-processing the $PERM$ for each mini-batch

Input: zero-initialized $COUNT[M][V]$, $PTR[V+1]$
Output: permuted order matrix $PERM$

for minibatch_id $= 0$ to M $-$ 1 **do**
  for document $= 0$ to B $-$ 1 **do**
    current_document $\leftarrow$ document + (B $\times$ minibatch_id)
    L $\leftarrow$ current_document_length
    for word $= 0$ to L $-$ 1 **do**
      current_word $\leftarrow$ $DATA$[current_document][word]
      $COUNT$[minibatch_id][current_word] $+= 1$
    end for
  end for
  sum $= 0$
  $PTR[0] =$ sum
  for v $= 1$ to V $+$ 1 **do**
    sum $+=$ $COUNT$[minibatch_id][v $-$ 1]
    PTR[v] $=$ sum
  end for
  for document $= 0$ to B $-$ 1 **do**
    current_document $\leftarrow$ document + (B $\times$ minibatch_id)
    L $\leftarrow$ current_document_lengths
    for word $= 0$ to L $-$ 1 **do**
      current_word $\leftarrow$ $DATA$[current_document][word]
      $PERM$[minibatch_id][document][word] $\leftarrow$ PTR[current_word]
      PTR[current_word]++
    end for
  end for
end for

Algorithm 7 Pre-computing the DNT

Output: pre-computed matrix DNT

for document $= 0$ to B $-$ 1 **in parallel do**
  current_document $\leftarrow$ document + (B $\times$ minibatch_id)
  for k $= 0$ to K $-$ 1 **do**
    DNT[current_document][k] $\leftarrow \frac{DT[current\_document][k]+\alpha}{NT[k]+V\beta}$
  end for
end for

in Algorithm 4. Instead of computing the standard conditional distribution, pre-computed *DNT* will be used in the sampling procedure.

2.4.2.3 Experimental Evaluation

We use Intel(R) Xeon(R) CPU E5-2680 (28-core) machine to measure the performance of parallel LDA with mini-batch processing. The left of the Figure 2.9 depicts the performance

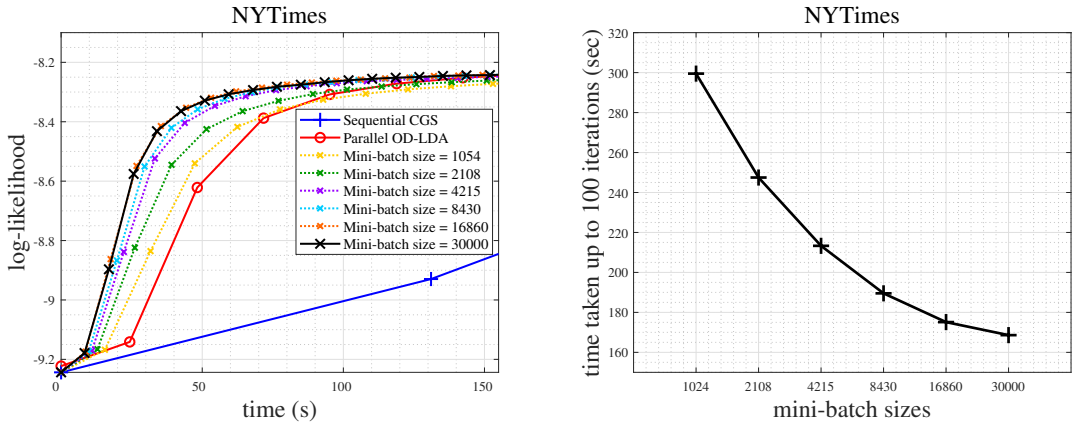


Figure 2.9: **Left**: Convergence over time across different mini-batch sizes on NYTimes dataset, K=128. X-axis: elapsed time in seconds; Y-axis: per-word log-likelihood. **Right**: Time taken to 100 iterations over different mini-batch sizes on NYTimes dataset, K=128. X-axis: mini-batch sizes; Y-axis: elapsed time in second.

variation when the size of mini-batches is varied on NYTimes dataset. It may be seen that larger size of mini-batches converge faster than smaller ones while achieving the same log-likelihood. As shown in the right graph in Figure 2.9, it is clear that the elapsed time reduces as the size of mini-batches increases, since the documents in the mini-batch are processed in parallel. The smaller size of mini-batches used in our parallel model definitely requires more iterations to finish. Experiments on an 28-core multi-processor show that

the parallel mini-batched implementation converges 1.5× faster than parallel atomics-free OD-LDA, while maintaining the same quality.

2.5 Parallel Latent Dirichlet Allocation on GPUs

2.5.1 Graphical Processing Units (GPUs)

Graphical Processing Units (GPUs) are consists of a set of Streaming Multiprocessor (SM). The lowest of level of parallelism on GPUs is on thread level. The group of threads makes WARP which is executed in a lock-step manner in an SM. A set of WARPs forms thread block that maps directly to SM. In GPUs, L2 cache is accessible by all thread blocks. On-chip shared memory is private to each thread block and registers are private to each thread. The main challenges to developing an efficient GPU algorithm is to ensure load balancing, low thread divergence, and high occupancy.

2.5.2 Overview of GPU Algorithm

As described in Section 2.4.2.1, the mini-batched LDA algorithm using delayed updates avoids the need to use atomic operations. However, in this case, an additional temporary storage is required to store information about the updated topic assignment. At the end of each mini-batch processing, these information will be used to update key structures. Since storage is scarce on GPUs, especially registers and shared-memory, keeping an additional space for delayed updates has a limitation on GPU implementation.

Moreover, the mini-batched LDA algorithm using immediate updates 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 atomic operations 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. As seen in Figure 2.8, using atomic-operations

Algorithm 8 Parallel Mini-batched Gibbs sampling for GPUs

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

```
1: M ← ceil(D / B)
2: repeat
3:    for minibatch_id = 0 to M − 1 do
4:       sampling_kernel() // generate new topic for each word in a mini-batch by sampling
         from DNT and WT and updating WT and NT
5:       memclear DT
6:       update_DT() // update DT from current Z
7:       update_DNT() // update DNT from current DT and NT
8:    end for
9: until convergence
```

enables a better convergence rate per-iteration. However, global memory atomic operations are expensive compared to shared memory atomic operations. GPUs have a limited amount of shared memory per SM. Therefore, in order to take advantage of the shared memory, 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. 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 the mini-batch processing, 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 2.10 depicts the overall scheme for our GPU implementation.

Algorithm 9 sampling_kernel(): sampling with atomic updates on the WT and NT

```
1:  for document = 0 to B − 1 in parallel do
2:    current_document ← document + (B × minibatch_id)
3:    L ← current_document_length
4:    for word = 0 to L − 1 do
5:      current_word ← DATA[current_document][word]
6:      old_topic ← Z[current_document][word]
7:      atomic decrement WT[current_word][old_topic]
8:      atomic decrement NT[old_topic]
9:      sum ← 0
10:     for k = 0 to K − 1 do
11:       sum += (WT[curr_word][k]+β)×DNT[curr_doc][k]
12:       p[k] ← sum
13:     end for
14:     U ← random_uniform() × sum
15:     for new_topic = 0 to K − 1 do
16:       if U < p[new_topic] then
17:         break
18:       end if
19:     end for
20:     atomic increment WT[current_word][new_topic]
21:     atomic increment NT[new_topic]
22:     Z[current_document][word] ← new_topic
23:   end for
24: end for
```

2.5.3 Details of Parallel GPU Algorithm

As mentioned in the previous Section 2.5.2, 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

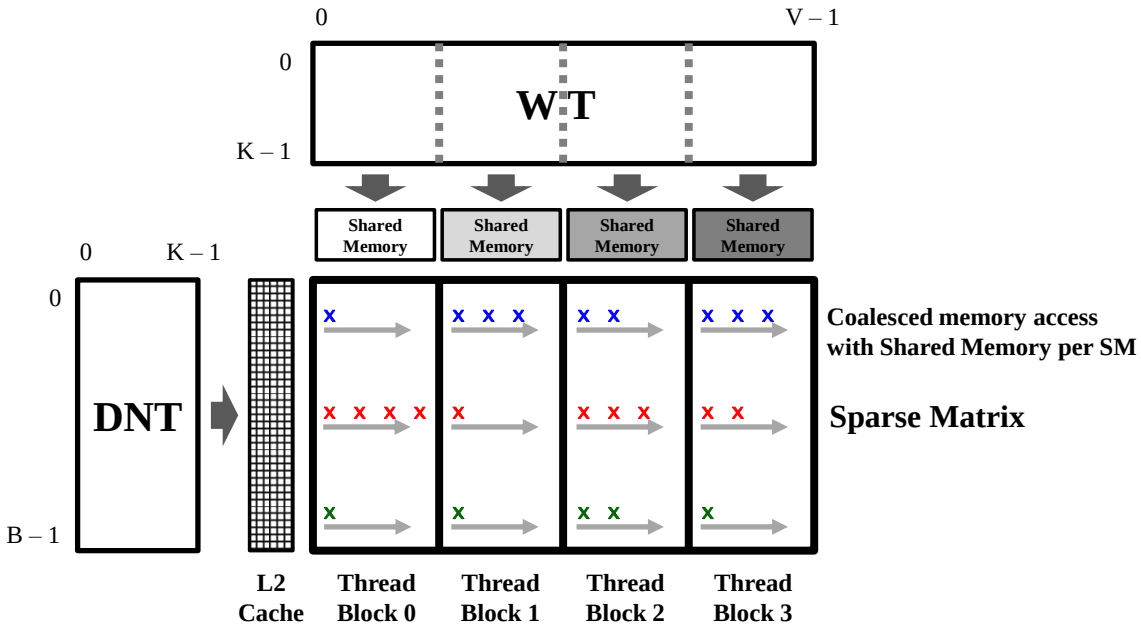


Figure 2.10: Overview of our GPU implementation. V: vocabulary size, B: number of documents in the current mini-batch, K: number of topics

column panels. Each column panel is mapped to a thread block. In order to devise the parallel LDA on GPUs, we consider several factors as follows:

- 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 uncoalesced. 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 10. 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 12) and then use this variable to compute the cumulative topic count as shown in Line 23 in Algorithm 10. 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 DNT matrix: In our algorithm, DNT 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.

- Segmented Compressed Sparse Row (CSR) representation: As describe in Figure 2.11, the elements of the sparse matrics are kept in segmented CSR representation. Thus, the threads with a column panel process all the words in a document before moving on to the next document. Since *DNT* is accessed from cache, exposing temporal reuse will increase the probability of a cache hit. This ensures that, within a column panel, the temporal reuse of *DNT* is maximized. As shown in Figure 2.10, the segmented CSR provides coalesced access on the row and column indices of the sparse matrix.

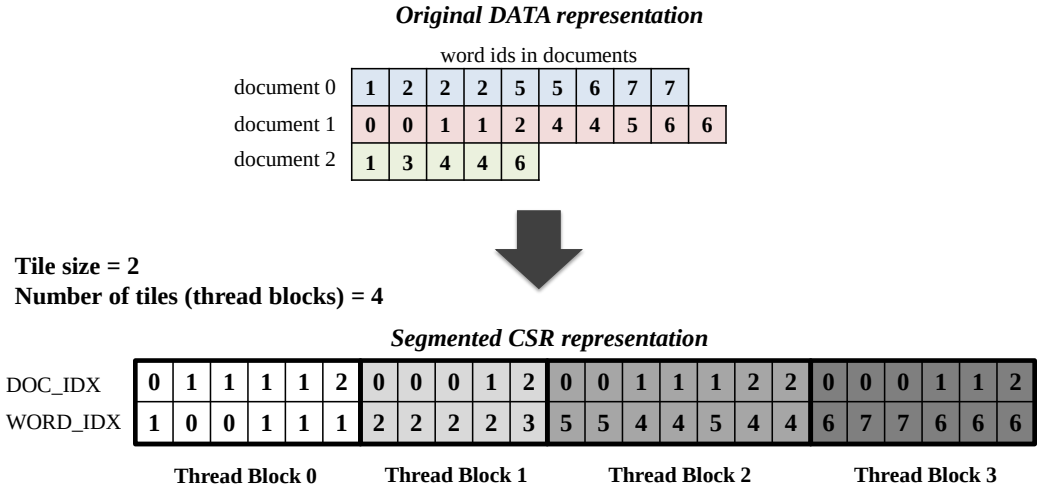


Figure 2.11: Example of converting original data representation to segmented CSR representation.

Algorithm 10 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

Algorithm 10 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

```
tile_id = block_id
tile_start = lastIdx[tile_id]
tile_end = lastIdx[tile_id + 1]
shared_WT[column_panel_width][K]
warp_id = thread_id / WARP_SIZE
lane_id = thread_id % WARP_SIZE
n_warp_k = thread_block_size / WARP_SIZE
// Coalesced data load from global memory to shared memory
for i=warp_id to column_panel step n_warp_k do
    for w = 0 to K step WARP_SIZE do
        shared_WT[i][w+lane_id] = WT[(tile_id×col_panel_width+i)][w+lane_id]
    end for
end for
__syncthreads()
for nnz = thread_id+tile_start to tile_end step thread_block_size do
    curr_doc_id = DOC_IDX[nnz]
    curr_word_id = WORD_IDX[nnz]
    curr_word_shared_id = curr_word_id − tile_id × column_panel_width
    old_topic = Z_IDX[nnz]
    atomicSub (shared_WT[curr_word_shared_id][old_topic], 1)
    atomicSub (NT[old_topic], 1)
    sum = 0
    for k = 0 to K − 1 do
        sum += (shared_WT[curr_word_shared_id][k]+β)×DNT[curr_doc_id][k]
    end for
    U = curand_uniform() × sum
    sum = 0
    for new_topic = 0 to K − 1 do
        sum += (shared_WT[curr_word_shared_id][k]+β)×DNT[curr_doc_id][k]
        if U < sum then
            break
        end if
    end for
    atomicAdd (shared_WT[curr_word_shared_id][new_topic], 1)
    atomicAdd (NT[new_topic], 1)
    Z_IDX[nnz] = new_topic
end for
// Update WT in global memory
for i=warp_id to column_panel step n_warp_k do
    for w = 0 to K step WARP_SIZE do
        WT[(tile_id×col_panel+i)][w+lane_id] = shared_WT[i][w+lane_id]
    end for
end for
__syncthreads()
```

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 11 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 12 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 11 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 12. In Algorithm 12, 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.

2.5.4 Experimental Evaluation

Two publicly available GPU-LDA implementations, **Lu-LDA** by Lu et al. [55] and **BIDMach-LDA** by Zhao et al. [111], are used in the experiments to compare the performance and accuracy of the approach developed in this section. We label our new implementation as Approximate GPU-Adapted LDA (**AGA-LDA**). We also use GibbsLDA++ [71] (**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 [50]. While Table 2.6 shows the characteristics of the datasets, Table 2.5 shows the configuration of the machines used for experiments.

Table 2.5: 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)

Table 2.6: 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. Using Equation 2.3 and 2.4, we evaluate the accuracy of LDA models on the test set. For each LDA model, training and testing algorithms are paired up. 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.

2.5.4.1 Speedup

Figure 2.12 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 2.13 shows the speedup of our approach over BIDMach-LDA and Lu-LDA. The y-axis in Figure 2.13 is the ratio of time for BIDMach-LDA and Lu-LDA to achieve a log-likelihood to how long AGA-LDA took.

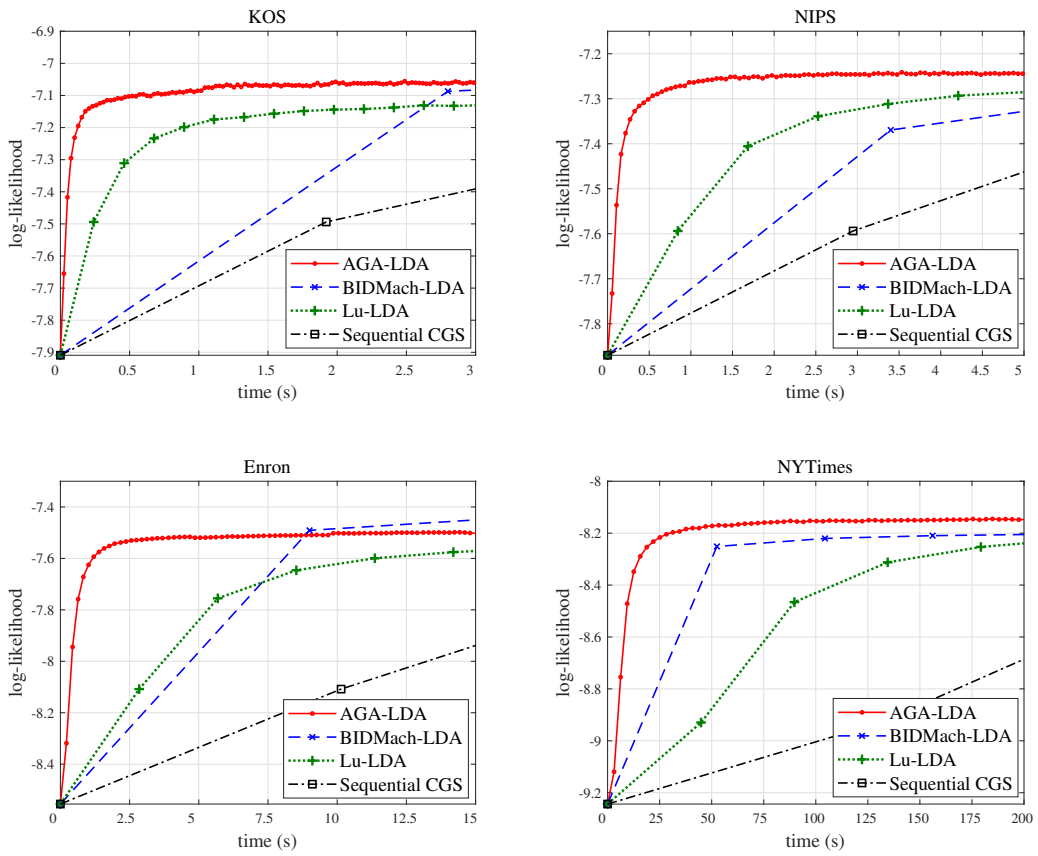


Figure 2.12: Convergence over time on KOS, NIPS, Enron and NYTimes datasets. The mini-batch sizes are set to 330, 140, 3750 and 28125 for KOS, NIPS, Enron and NYTimes, respectively.

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.

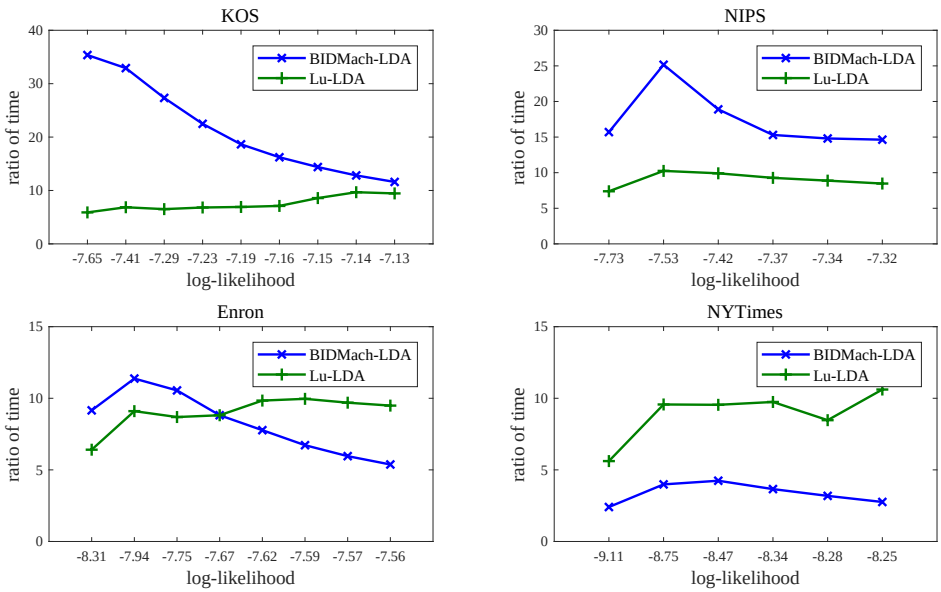


Figure 2.13: Speedup of AGA-LDA over BIDMach-LDA and Lu-LDA.

2.6 Conclusion

In this chapter, we have presented a parallel CGS for LDA on multi-core CPUs. The new parallel implementation is able to effectively parallelize the CGS approach while maintaining the same log-likelihood. We use over-decomposition techniques for load balancing. The experimental section demonstrates that we achieve 4.1× speedup for NIPS and 4.9× speedup for NYTimes. We also 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.

Chapter 3: Parallel Locality-Optimized Non-negative Matrix Factorization

3.1 Introduction

Non-negative Matrix Factorization (NMF) proposed by Lee and Seung [44] is a key primitive for unsupervised dimension reduction used in a wide range of applications, including topic modeling [42,85,89], recommender systems [1,33,110] and bioinformatics [58,97,104]. For example, when NMF is used for topic modeling, given a document-word matrix in which a document is represented as a collection of bag-of-words from an active vocabulary, NMF decomposes a document-word matrix into two non-negative matrices where each of the factorized matrices can be interpreted as latent topic distributions for documents and words.

Given a non-negative matrix $A \in \mathbb{R}_+^{V \times D}$ and $K \ll \min(V,D)$, NMF finds two non-negative rank-K matrices $W \in \mathbb{R}_+^{V \times K}$ and $H \in \mathbb{R}_+^{K \times D}$, such that the product of W and H approximates A [44]:

$$A \approx WH \tag{3.1}$$

Several algorithms have been proposed for NMF. They all involve repeated alternating update of some elements of W interleaved with update of some elements of H, with imposition on non-negativity constraints on the elements, until a suitable error norm (either Frobenius

norm or Kullback-Leibler divergence) is lower than a desired threshold. Various previously developed algorithms for NMF differ in the granularity of the number of elements of W that are updated before switching to updating some elements of H. The focus of prior work has been to compare the rates of convergence of alternate algorithms and the parallelization of the algorithms. However, to the best of our knowledge, the minimization of data movement through the memory hierarchy, using techniques like tiling, has not been previously addressed. With costs of data movement from memory being significantly higher than the cost of performing arithmetic operations on current processors, data locality optimization is extremely important.

In this chapter, we address the issue of data locality optimization for NMF. An analysis of the computational components of the FAST-HALS (Hierarchical Alternating Least Squares) algorithm for NMF [14], is first performed to identify data movement overheads. The associativity of addition is utilized to judiciously reorder additive contributions in updating elements of W and H, to enable 3D tiling of a computationally intensive component of the algorithm. An analysis of the data movement overheads as a function of tile size is developed, leading to a model for selection of effective tile sizes. Parallel implementations of the new **P**arallel **L**ocality-optimized **NMF** algorithm (called **PL-NMF**) are presented for both GPUs and multi-core CPUs. An experimental evaluation with datasets used in prior studies demonstrates significant performance improvement over state-of-the-art alternatives available for parallel NMF.

Chapter 3 is organized as follows. In the next section, we present the background on NMF and related prior work. In Section 3.3, we present the high-level overview of PL-NMF algorithm. Sections 3.4 demonstrates details of our PL-NMF for multi-core CPUs and GPUs. In Section 3.5, we compare the data movement cost for PL-NMF and original

FAST-HALS algorithms. Thereafter, we present determination of the tile sizes based on data movement analysis. Section 3.6 compares PL-NMF with existing state-of-the-art parallel implementations.

3.2 Background

3.2.1 Non-negative Matrix Factorization Algorithms

NMF seeks to solve the optimization problem of minimizing reconstruction error between A and the approximation WH. In order to measure the reconstruction error for NMF, Lee et al. [44] adopted various objective functions, such as the Frobenius norm given two matrices and Kullback-Leibler divergence given two probability distributions. The objective functions $D(A||WH)$ based on the Frobenius norm is defined in Equation 3.2.

$$D_F(A||WH) = \frac{1}{2}||A - WH||_F^2 = \frac{1}{2}\sum_{vd}(A_{vd} - (WH)_{vd})^2 \tag{3.2}$$

To efficiently minimize the objective functions (above), several variants of NMF algorithms have been developed: *Multiplicative Update (MU)*, *Additive Update (AU)*, *Alternating Non-negative Least Squares (ANLS)* and *Hierarchical Alternating Least Squares (HALS)*. Table 3.1 describes the notations used in this chapter.

Table 3.1: Common notations for NMF algorithms

Notation	Description
A	Non-negative matrix
W	Non-negative rank-K matrix factor
H	Non-negative rank-K matrix factor
V	Number of rows in A and W
D	Number of columns in A and H
K	Low rank

Multiplicative update (MU) and additive update (AU) proposed by Lee et al. [44] are the simplest NMF algorithms. The MU algorithm updates two rank-K non-negative matrices W and H based on multiplicative rules and ensures convergence. MU strictly conforms to non-negativity constraints on W and H because the elements of W and H that have zero value will not be updated. Unlike MU algorithm, the AU algorithm updates W and H based on the gradient descent method and avoids negative update values using learning rate. However, some studies have reported that the use of MU and AU algorithms leads to weaknesses such as slower convergence and lower convergence rate [23, 37, 51].

Alternating Non-negative Least Squares (ANLS) is a special type of Alternating Least Squares (ALS) approach. At each iteration, the gradients of two objective functions with respect to W and H are used to update each of W and H one after the other. Kim et al. [38] proposed Alternating Non-negative Least Squares based Block Principle Pivoting (ANLS-BPP) algorithm. Under the Karush-Kuhn-Tucker (KTT) conditions, the ANLS-BPP algorithm iteratively finds the indices of non-zero elements (passive set) and zero elements (active set) in the optimal matrices until KTT conditions are satisfied. The values of indices that correspond to the active set will become zero, and the values of passive set are approximated by solving $min||A - WH||_F^2$ which is a standard Least Squares problem.

As an alternative to the basic ANLS approach, Cichocki et al. [15] proposed Hierarchical Alternating Least Squares (HALS), which hierarchically updates only one k-th row vector of $H \in \mathbb{R}_+^{K \times D}$ at a time and then uses it to update a corresponding k-th column vector of $W \in \mathbb{R}_+^{V \times K}$. In other words, HALS minimizes the K set of two local objective functions with respect to K row vectors of H and K column vectors of W at each iteration. A standard HALS algorithm iteratively updates each row of H and each column of W in order within the innermost loop.

Based on the standard HALS algorithm, Cichocki et al. [14] further proposed the extended version of a new algorithm called FAST-HALS algorithm as described in Algorithm 13. Note that H_k and W_k indicate k-th row of H and k-th column of W, respectively. FAST-HALS updates all rows of H before starting the update to all columns of W, instead of alternately updating each row of H and each column of W at a time. Compared to MU algorithm, the FAST-HALS algorithm converges much faster and produces a better solution, while maintaining a similar computational cost as reported in [22, 38]. Interestingly, Kim et al. [38] have shown that FAST-HALS has also been found to converge faster than their ANLS-BPP implementation on real-world text datasets: TDT2 and 20 Newsgroups, while maintaining the same convergence rate (see Figure 5.3 in Kim et al. [38]).

3.3 Related Work on Parallelization of NMF

Since most of the variations of NMF algorithm are highly compute-intensive, many previous efforts have been made to parallelize NMF algorithms. As shown in Table 3.2, previous studies on parallelizing NMF can be broadly categorized into two groups based on implementation for multi-core CPUs [6, 18, 19, 36, 49, 53] versus GPUs [41, 54, 58]. Furthermore, each study used various NMF algorithms for parallel implementations.

3.3.1 Shared-Memory Multiprocessor

Battenberg et al. [6] introduced parallel NMF using MU algorithm for audio source separation task. Fairbanks et al. [19] adopted ANLS-BPP based NMF in order to find the structure of temporal behavior in a dynamic graph given vertex features. Both [6] and [19] developed the parallel NMF implementations on multi-core CPUs using Intel Math Kernel Library (MKL) along with shared-memory multiprocessor.

Table 3.2: Previous studies on parallelization of NMF

Author	Machine	Platform	Algorithm
Battenberg et al. [6]	CPU	Shared-memory	MU
Fairbanks et al. [19]	CPU	Shared-memory	ANLS-BPP
Dong et al. [18]	CPU	Distributed-memory	MU
Liu et al. [53]	CPU	Distributed-memory	MU
Liao et al. [49]	CPU	Distributed-memory	MU
Kannan et al. [36]	CPU	Distributed-memory	ANLS-BPP
Lopes et al. [54]	GPU	Shared-memory	MU, AU
Koitka et al. [41]	GPU	Shared-memory	MU, ALS
Mejía-Roa et al. [58]	GPU	Distributed-memory	MU

3.3.2 Distributed-Memory Systems

Dong et al. [18] demonstrated that MU algorithm and shared-memory based parallel implementation have a limitation of slow convergence. To overcome these problems, they devised a parallel MPI implementation of MU based NMF that improves Parallel NMF (PNMF) proposed by Robila et al. [77]. Different NMF algorithms have previously used tiling/blocking to minimize data movement. Dong et al. [18] partitioned the two factor matrices, W and H, into smaller blocks and each block is distributed to different threads. Each block simultaneously updates corresponding sub-matrices of the two matrices, and a reduction operation is performed by collective communication operations using Message Passing Interface (MPI). Similarly, Liu et al. [53] proposed matrix partition scheme that partitions the two factor matrices along the shorter dimension (K dimension) instead of the longer dimensions (V or D dimensions). Therefore, each matrix is divided up to more partitions compared to partitioning along the longer dimension, so that the data locality is increased and the communication cost is decreased when performing the product of two

matrices. Kannan et al. [36] minimized the communication cost by communicating only with the two factor matrices and other partitioned matrices among parallel threads. Based on the ANLS-BPP algorithm, their implementation also reduced the bandwidth and data latency using MPI collective communication operations. Given an input matrix A and two factorized matrices W and H, they partitioned W and H into P multiple blocks (tiles) across V and D dimensions which are the number of rows in W and columns in H. Hence, the sizes of each block in W and H are $(V/P)\times K$ and $K \times(D/P)$, respectively. Doing so allows the matrix A to be partitioned into P tiles $\times$ P tiles. Then P different processors perform matrix multiplication with the different P tiles of W and H simultaneously. This data partition scheme is appropriate for block-wise updates of W and H based on ANLS-BPP algorithm. Unlike ANLS-BPP algorithm, FAST-HALS requires column-wise/row-wise sequential updates because there is a data dependency between two consecutive columns/rows. Hence, FAST-HALS algorithm is not allowed to divide W and H across V and D dimensions. In our tiling approach, W and H are partitioned across K dimension, and the sizes of each block in W and H are $V \times(K/P)$ and $(K/P)\times D$, respectively. Our key contribution is not tiling/blocking itself, but converting matrix-vector operations to matrix-matrix operations. Tiling enables us to do the latter.

3.3.3 GPU Platform

Lopes et al. [54] proposes several GPU-based parallel NMF implementations that use both MU and AU algorithms for both Euclidean and KL divergence objective functions. Mejía-Roa et al. [58] presents NMF-mGPU that performs MU based NMF algorithm on either a single GPU device or multiple GPU devices through MPI for a large-scale biological

dataset. Koitka et al. [41] presents MU and ALS based GPU implementations binding to the R environment. To our knowledge, our work is the first to develop FAST-HALS based parallel NMF implementation for GPUs.

3.4 Overview of Approach

In this section, we present a high-level overview of our approach to optimize NMF for data locality. We begin by describing the FAST-HALS algorithm [14], one of the fastest algorithms for NMF as demonstrated by previous comparison studies [38]. We analyze the data movement overheads from main memory, for different components of that algorithm, and identify the main bottlenecks. We then show how the algorithm can be adapted by exploiting the associativity of addition to make the computation effectively tileable to reduce data movement from memory, whereas the original form is not tileable.

3.4.1 Overview of FAST-HALS Algorithm

Algorithm 13 shows pseudo-code for the FAST-HALS algorithm [14] for NMF. It is an iterative algorithm that iteratively updates H and W, fully updating all entries in H (lines 4-8) and then updating all entries in W (lines 10-15) during each iteration until convergence. While the updates to H and W are slightly different (due to normalization of W after each iteration), each of the updates involves a pair of matrix-matrix products (lines 4/5 and 10/11 for H and W, respectively) and a sequential loop that steps through features (k loop) to update one row (column) of $H(W)$ at a time. The computation within these k loops involves vector-vector operations and matrix-vector operations. From a computational complexity standpoint, the various matrix-matrix products and the sequential (K times) matrix-vector products all have cubic complexity (O(N^3) if all matrices are square and of side N). But as we show by analysis of data movement requirements in the next sub-section, the collection

Algorithm 13 FAST-HALS algorithm

Input: $A \in \mathbb{R}_{+}^{V \times D}$: non-negative matrix, ε: small non-negative quantity

Initialize $W \in \mathbb{R}_{+}^{V \times K}$ and $H \in \mathbb{R}_{+}^{K \times D}$ with random non-negative numbers
repeat
  // Updating H
  $R \leftarrow A^T W$
  $S \leftarrow W^T W$
  for k $= 0$ to K $- 1$ **do**
    $H_k \leftarrow \max\left(\varepsilon, H_k + R_k - H^T S_k\right)$
  end for
  // Updating W
  $P \leftarrow AH^T$
  $Q \leftarrow HH^T$
  for k $= 0$ to K $- 1$ **do**
    $W_k \leftarrow \max(\varepsilon, W_k Q_{kk} + P_k - WQ_k)$
    // Normalize W_k column vector with $L_2 - norm$
    $W_k \leftarrow \dfrac{W_k}{||W_k||_2}$
  end for
until convergence

of matrix-vector products in lines 7 and 13 dominate. In the following sub-section, we present our approach to alleviating this bottleneck by exploiting the flexibility of instruction reordering via use of the associativity property of addition[1].

3.4.2 Data Movement Analysis for FAST-HALS Algorithm

The code regions with high data movement can be identified by individually analyzing each line in Algorithm 13. Lines 4 and 5 perform matrix multiplication. It is well known that $\frac{2MNK}{\sqrt{C}}$ is the highest order term in the number of data elements moved (between main memory and a cache of size C words) for efficient tiled matrix multiplication of two matrices A, $(M \times K)$ and B, $(K \times N)$[2]. Thus, the data movement costs associated with lines 4 and 5 are $\frac{2DKV}{\sqrt{C}}$ and $\frac{2KKV}{\sqrt{C}}$, respectively. The loop in line 6 performs matrix-vector multiplication and has an associated data movement cost of $K(3D + DK + K)$. Similar to lines 4 and 5, the data movement costs for lines 10 and 11 are $\frac{2VKD}{\sqrt{C}}$ and $\frac{2KKD}{\sqrt{C}}$, respectively. The loop in line 12 has an associated data movement cost of $K(VK + K + 6V + 1)$. The total data movement for Algorithm 13 is shown in Equation 3.3.

$$K(K(V+D)(1+\frac{2}{\sqrt{C}})+\frac{4VD}{\sqrt{C}}+6V+3D+2K+1) \tag{3.3}$$

The main data movement overhead is associated with loops in lines 6 and 12. For example, the combined fractional data movement overhead of lines 7 (within loop in line 6) and 13 (within loop in line 12) is 91% for the 20 Newsgroups dataset. If the operational intensity (defined as the number of operations per data element moved) is very low, the performance will be bounded by memory bandwidth and thus will not be able to achieve the

[1]Floating-point addition is of course not strictly associative, but as shown later by the experimental results, the changed order does not adversely affect algorithm convergence.

[2]An extensive discussion of both lower bounds and data movement volume for several tiling schemes may be found in the recent work of Smith [87].

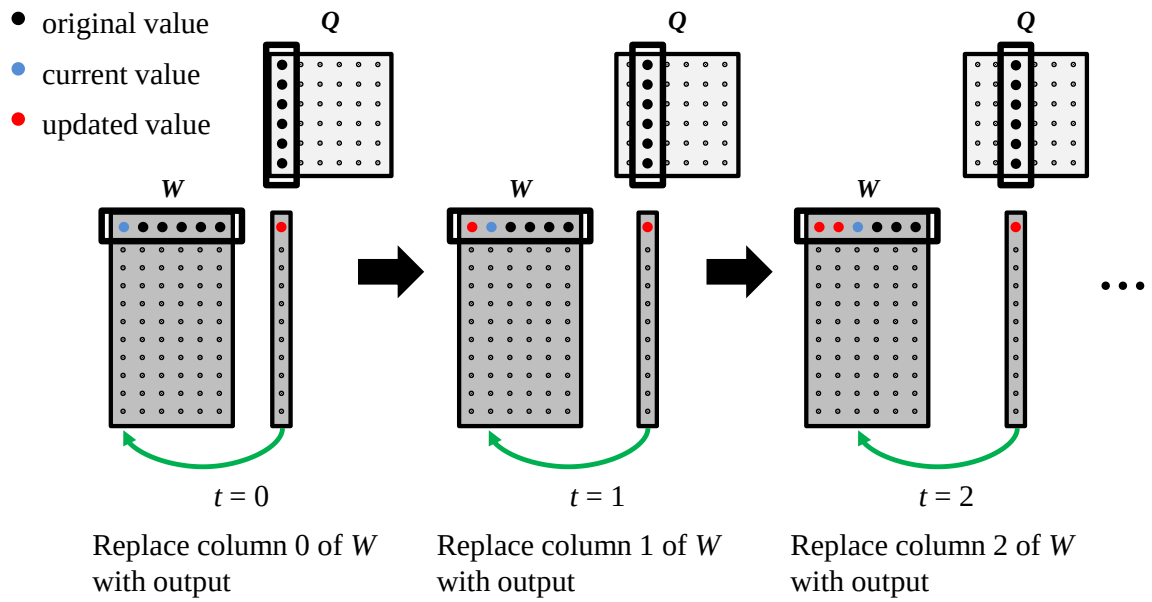


Figure 3.1: FAST-HALS: Update of W.

peak compute capacity. Due to its low operational intensity, the performance of Algorithm 13 is limited by the memory bandwidth. Thus, the major motivation for our algorithm adaptation is to achieve better performance by reducing the required data movement.

3.4.3 Overview of PL-NMF

In this sub-section, we describe how the FAST-HALS algorithm is adapted by exploiting the flexibility of changing the order in which additive contributions to a data element are made. Before describing the adaptation, we first highlight the interaction between different columns of W in the original algorithm. Figure 3.1 depicts the update of W which corresponds to the lines 12 to 16 in Algorithm 13.

In Algorithm 13, t^{th} column of W is updated as the product of W with t^{th} column of Q which is a matrix-vector multiplication operation. Since the update to $(t+1)^{th}$ column depends on t^{th} column, different columns (t: features) are updated sequentially. Let W_old

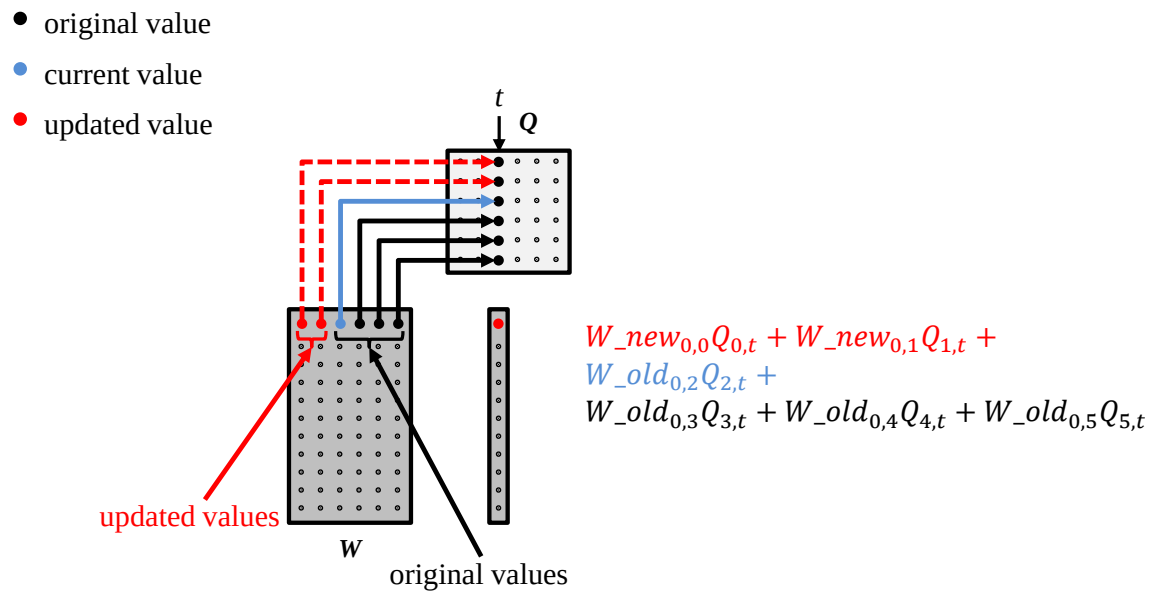


Figure 3.2: FAST-HALS: Updating a single element of W. The dash represents updated value.

represent the values at the beginning of the current outer iteration, and let W_new represent the values at the end of current outer iteration (updated values). Interaction between W_old and W_new is shown in Figure 3.2 which depicts the contributions from W_old and W_new to $W\_new_{i,t}$. $W\_new_{i,t}$ can be obtained by $\sum_{j=0}^{t-1} W\_new_{i,j} \times Q_{j,t} + \sum_{j=t}^{K-1} W\_old_{i,j} \times Q_{j,t}$. Figure 3.3 shows the contributions of $W\_old_{i,t}$ and $W\_new_{i,t}$ to $W\_new_{i,*}$. $W\_old_{i,t}$ contributes to $W\_new_{i,j}$ $\forall j | j \leq t$, and $W\_new_{i,t}$ contributes to $W\_new_{i,j}$ where $\forall j | j > t$. In other words, the old value of column t is used to update the columns to the left of t (and self), and the new/updated value of column t is used to update the columns to the right of column t.

If we partition W into a set of column panels (tiles) of size T, the interactions between columns can be expressed in terms of tiles as depicted in Figure 3.4. Similar to individual columns, the old value of tile τ is used to update the columns to the left of τ (phase 1), and

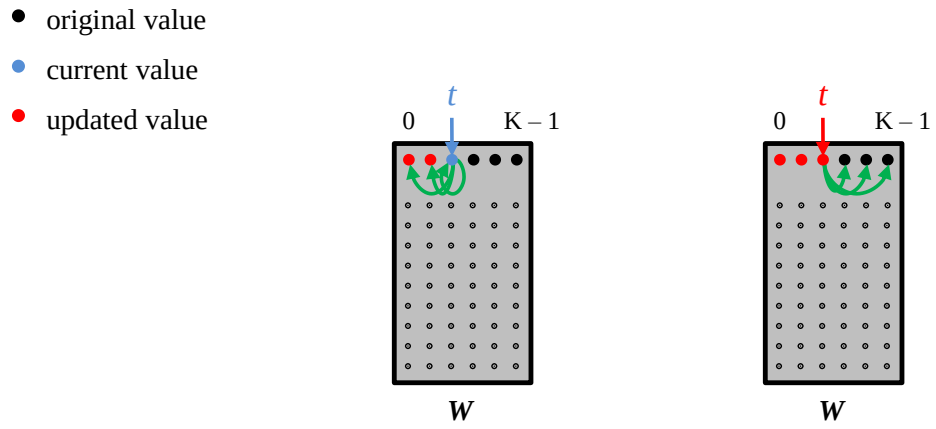


Figure 3.3: The contributions from $W_{0,t}$ to other elements.

the new/updated value of tile τ is used to update the tiles to the right of tile τ (phase 3). The updates to different columns with a tile (phase 2) is done sequentially.

The contributions to tiles to the left of current tile τ can be done as $W\_new_{i,j} -= W\_old_{i,\tau\times T:((\tau+1)\times T)-1} \times Q_{\tau\times T:((\tau+1)\times T)-1,j}$ where $\forall j | j < \tau \times T - 1$. Similarly, contributions to tiles to the right of current tile τ can be done as $W\_new_{i,j} -= W\_new_{i,\tau\times T:((\tau+1)\times T)-1} \times Q_{\tau\times T:((\tau+1)\times T)-1,j}$ where $\forall j | j > (\tau+1) \times T$. Both phases 1 and 3 can be performed using matrix-matrix operations which are known to have much better performance and lower data movement than matrix-vector operations. Note that the total number of operations in both the original formulation and our formulation are exactly the same.

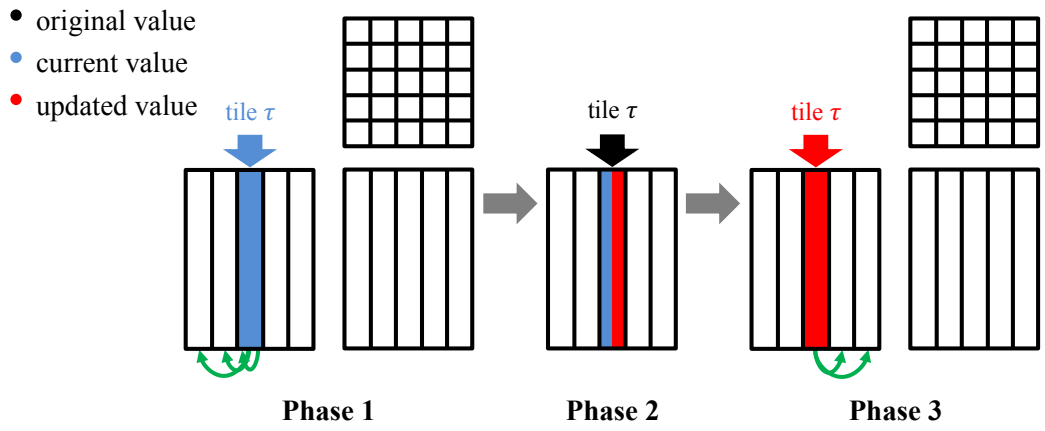


Figure 3.4: Overview of our approach for updating W.

3.5 Details of PL-NMF on Multi-Core CPUs and GPUs

3.5.1 Parallel CPU Implementation

Algorithm 14 shows our CPU pseudo-code for updating W. We begin by computing AH^T (line 1). If A is sparse, then the actual implementation uses **mkl_dcsrmm**() and **cblas_dgemm**() is used otherwise. Line 2 computes the HH^T (using **cblas_dgemm**()). The W values from the previous iteration are kept in W_old. We maintain another data structure called W_new which represents the updated W values. W_new is initialized by the loop in line 4. By using Equation 3.4, phase 1 is done by the loop in line 11. Figure 3.5 illustrates the actual computations of tiled matrix-matrix multiplications for three sequential phases, where τ denotes the index of the current tile and T is the size of each tile. For example, at current tile τ, phase 1 performs multiplication of the same colored/patterned two sub-matrices (tiles) in W_old and Q to update the result matrix W_new.

Algorithm 14 Parallel CPU implementation for updating W

Input: $A \in \mathbb{R}_{+}^{V \times D}$: input matrix, W_old and W_new: $V \times K$ non-negative matrix factor, H: $D \times K$ non-negative matrix factor, T: Tile size, ε: small non-negative quantity, γ: total number of tiles

$P \leftarrow AH^T$
$Q \leftarrow HH^T$
// Initialize *W_new* using *W_old* and *Q*
for v = 0 **to** V − 1 **do**
  for k = 0 **to** K − 1 **do**
    W_new[v][k] ← W_old[v][k] × Q[k][k]
  end for
end for
// Phase 1
$\gamma \leftarrow K$ / T
for tile_id = 0 **to** γ − 1 **do**
  W_new[0:V-1][0:(tile_id× T:((tile_id +1)× T)-1] − = **dgemm**(W_old[0:V-1][tile_id× T:((tile_id +1)× T)-1], Q[tile_id× T:((tile_id +1)× T)-1][0:(tile_id× T)-1])
end for
// Phase 2 & Phase 3
for tile_id = 0 **to** γ − 1 **do**
  // Phase 2
  for t = tile_id × T **to** (tile_id + 1) × T − 1 **do**
    sum_square ← 0
    #pragma omp parallel for reduction(+:sum_square)
    for v = 0 **to** V − 1 **do**
      sum ← 0
      k ← tile_id × T
      #pragma omp simd reduction(+:sum)
      for ; **to** t − 1 **do**
        sum ← sum + W_new[v][k] × Q[t][k]
      end for
      #pragma omp simd reduction(+:sum)
      for k = t; **to** (tile_id + 1) × T − 1 **do**
        sum ← sum + W_old[v][k] × Q[t][k]
      end for
      W_new[v][t] ← max(ε, W_new[v][t] + P[v][t] − sum)
      sum_square ← sum_square + W_new[v][t] × W_new[v][t]
    end for
    #pragma omp parallel for
    for v = 0 **to** V − 1 **do**
      W_new[v][t] ← W_new[v][t] / sqrt(sum_square)
    end for
  end for
  // Phase 3
  W_new[0:V-1][(tile_id +1)× T:K-1] − = **dgemm**(W_new[0:V-1][tile_id× T:((tile_id +1)× T)-1], Q[tile_id× T:((tile_id +1)× T)-1][(tile_id +1)× T:K-1])
end for

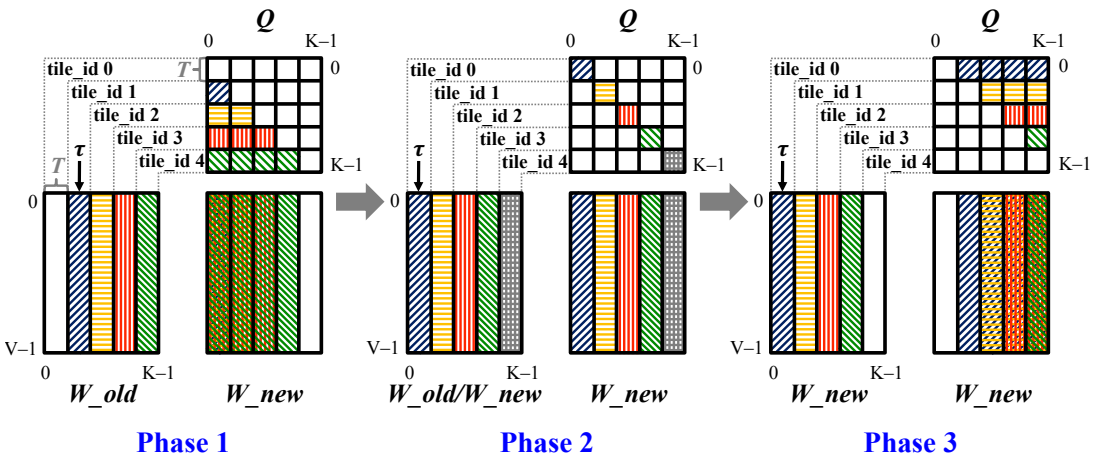


Figure 3.5: Computations of three phases for updating W.

$$
\begin{aligned}
&W\_new[:, 0:(\tau \times T)-1] -= \\
&\quad W\_old[:, (\tau \times T):((\tau+1) \times T)-1] \cdot \\
&\quad Q[(\tau \times T):((\tau+1) \times T)-1, 0:(\tau \times T)-1]
\end{aligned}
\tag{3.4}
$$

The loop in line 17 performs phase 2 computations as formulated in Equation 3.5. In order to take advantage of the vector units, the loops in lines 24 and 28 are vectorized. Additionally, a column-wise normalization for W_new is performed within phase 2 (line 36).

$$
\begin{aligned}
&W\_new[:, (\tau \times T):((\tau+1) \times T)-1] -= \\
&\quad W[:, (\tau \times T):((\tau+1) \times T)-1] \cdot \\
&Q[(\tau \times T):((\tau+1) \times T)-1, (\tau \times T):((\tau+1) \times T)-1] \\
&\quad + P[:, (\tau \times T):((\tau+1) \times T)-1]
\end{aligned}
\tag{3.5}
$$

The matrix-matrix multiplication in line 40 corresponds to the phase 3 computations using Equation 3.6. As depicted in Figure 3.5, the tiles involving phase 3 and phase 1 computations

are different from each other.

$$W\_new[:, ((\tau+1)\times T) : K-1] -= \\ W\_new[:, (\tau \times T) : ((\tau+1)\times T)-1] \cdot \\ Q[(\tau \times T) : ((\tau+1)\times T)-1, ((\tau+1)\times T) : K-1] \tag{3.6}$$

Finally, our parallel CPU implementation completely substitutes lines 10 to 16 in Algorithm 13 for all lines in Algorithm 14. Similarly, H will be updated in the same fashion as updating W except for the normalization part.

3.5.2 Parallel GPU Implementation

Similar to our CPU algorithm, our GPU algorithm also tries to minimize the data movement. Algorithm 15, 16 and 17 show the pseudo-code of our GPU algorithm. Since the overall structure of the GPU algorithm is similar to the CPU algorithm, this section only highlights the differences. Algorithm 15 runs on the host which is responsible for launching GPU kernels. The sparse matrix-dense matrix multiplication is implemented using **cusparseDcsrmm()**, and dense matrix-dense matrix multiplication is implemented using **cublasDgemm()**.

Algorithm 16 shows the pseudo-code for phase 2. In GPUs, the reduction across V (for normalization of W) can be performed using global memory atomic operations which are very expensive. Hence, our implementation uses efficient hierarchical reduction. The reduction within a thread block is done in 4 steps: i) in line 18, the reduction across the threads within a warp is done using efficient warp shuffling primitives, ii) all the threads with lane id 0 write the reduced value to shared memory (line 20), iii) in line 24, the first warp of the thread block loads the previously written values from shared memory and iv) all the threads in the first warp again performs warp reduction (line 26). In order to perform

Algorithm 15 GPU implementation of updating W on host

Input: $A \in \mathbb{R}_{+}^{V \times D}$: input matrix, W_old and W_new: $V \times K$ non-negative matrix factor, H: $D \times K$ non-negative matrix factor, T: Tile size, ε: small non-negative quantity, γ: total number of tiles

1: $P \leftarrow AH^T$
2: $Q \leftarrow HH^T$
3: **// Initialize** W_new **using** W_old **and** Q
4: init_W_new()
5: **// Phase 1**
6: $\gamma \leftarrow K$ / T
7: **for** tile_id $= 0$ **to** $\gamma - 1$ **do**
8: W_new[0:V-1][0:(tile_id$\times$ T)-1] $-=$ **cublasDgemm**(W_old[0:V-1][tile_id$\times$ T:((tile_id +1)$\times$ T)-1], Q[tile_id$\times$ T:((tile_id +1)$\times$ T)-1][0:(tile_id$\times$ T)-1])
9: **end for**
10: **// Phase 2 & Phase 3**
11: **for** tile_id $= 0$ **to** $\gamma - 1$ **do**
12: **// Phase 2**
13: **for** t $=$ tile_id $\times$ T **to** (tile_id $+ 1$) $\times$ $T - 1$ **do**
14: **cudaMemset**(sum_square,0)
15: update_W_phase_2()
16: __cudaDeviceSynchronize()
17: update_W_norm()
18: __cudaDeviceSynchronize()
19: **end for**
20: **// Phase 3**
21: W_new[0:V-1][(tile_id +1)$\times$ T:K-1] $-=$ **cublasDgemm**(W_new[0:V-1][tile_id$\times$ T:((tile_id +1)$\times$ T)-1], Q[tile_id$\times$ T:((tile_id +1)$\times$ T)-1][(tile_id +1)$\times$ T:K-1])
22: **end for**

Algorithm 16 GPU implementation of update_W_phase_2 kernel

Input: W_old, W_new, P, Q, sum_square, t, tile_id, T, V, K, ε

vId $\leftarrow$ blockIdx.x $\times$ blockDim.x + threadIdx.x // threadID
__shared__ $shared\_sum$[1024/32]
sum_reduce = 0.0f
if vId $< V$ **then**
    sum = 0
    for k = tile_id $\times$ T **to** (tile_id + 1) $\times$ T $-$ 1 **do**
        if k $<$ t **then**
            sum $\leftarrow$ sum + W_new[vId + k $\times$ V][k] $\times$ Q[k $\times$ K + t]
        else
            sum $\leftarrow$ sum + W_old[vId + k $\times$ V] $\times$ Q[k $\times$ K + t]
        end if
    end for
    W_new[vId + t $\times$ V] $\leftarrow$ max(ε, W_new[vId + t $\times$ V] + P[vId + t $\times$ V] $-$ sum)
    sum_reduce $\leftarrow$ W_new[vId + t $\times$ V]
end if
sum_reduce $\leftarrow$ sum_reduce $\times$ sum_reduce
// Warp-level reduction
sum_reduce $\leftarrow$ **warp_reduce**(sum_reduce)
// Block-level reduction
if threadIdx.x % 32 == 0 **then**
    $shared\_sum$[threadIdx.x / 32] $\leftarrow$ sum_reduce
end if
__syncthreads()
if threadIdx.x / 32 == 0 **then**
    sum_reduce $\leftarrow$ $shared\_sum$[threadIdx.x]
    sum_reduce $\leftarrow$ **warp_reduce**(sum_reduce)
end if
if threadIdx.x == 0 **then**
    atomicAdd (sum_square, sum_reduce)
end if

Algorithm 17 GPU implementation of update_W_norm kernel

Input: *W_new*, *sum_square*, t, *V*

1: vId $\leftarrow$ blockIdx.x $\times$ blockDim.x + threadIdx.x // threadID
2: **if** vId $< V$ **then**
3: return
4: **end if**
5: *W_new*[vId + t $\times$ *V*] $\leftarrow$ *W_new*[vId + t $\times$ *V*] / sqrt(*sum_square*)

reduction across multiple thread blocks, we use atomic operations which is shown in line 29.

Algorithm 17 shows the pseudo-code for normalization.

3.6 Modeling: Determination of the tile size

In this section, we first compare the data movement cost of our approach with original FAST-HALS algorithm. Then the data movement of our algorithm as a function of T is developed to select effective tile sizes.

$$\sum_{i=0}^{\frac{K}{T}-1} iVT^2(\frac{1}{T}+\frac{2}{\sqrt{C}}) = VT^2(\frac{1}{T}+\frac{2}{\sqrt{C}})(\frac{K^2-KT}{2T^2}) \tag{3.7}$$

$$\sum_{i=0}^{\frac{K}{T}-1} T(VT+T+V) = \frac{K}{T}T(VT+T+V) \tag{3.8}$$

In our approach, W is updated in three phases. Phases 1 and 3 can be implemented using matrix-multiplication, and the corresponding cost is shown in Equation 3.7, where T represents the tile size and C is the cache size. Phase 2 can be implemented using matrix-vector multiplication and the associated cost is shown in Equation 3.8. Since loading matrix W dominates the data movement cost in phase 2, the cost of loading vectors can be ignored. Equation 3.9 shows the total data movement required for updating W.

$$vol(T) = V(\frac{1}{T}+\frac{2}{\sqrt{C}})(K^2-KT)+\frac{K}{T}T(VT) \tag{3.9}$$

The cost of updating H is similar to updating W. Compared to updating W, updating H does not require accessing Q. In addition, since H is not normalized, the cost associated with normalization is also not present.

The data movement cost of the original loop in line 12 in Algorithm 13 is $K(VK+K+6V+1)$. Hence, for the 20 Newsgroups dataset (V=26,214) with low rank K=240 on a machine with 35 MB cache, the data movement cost of original scheme is 1,547,732,400 bytes. However, in our scheme based on Equation 3.9, the cost is only 189,208,129 bytes which is 8.2$\times$ lower than the original scheme.

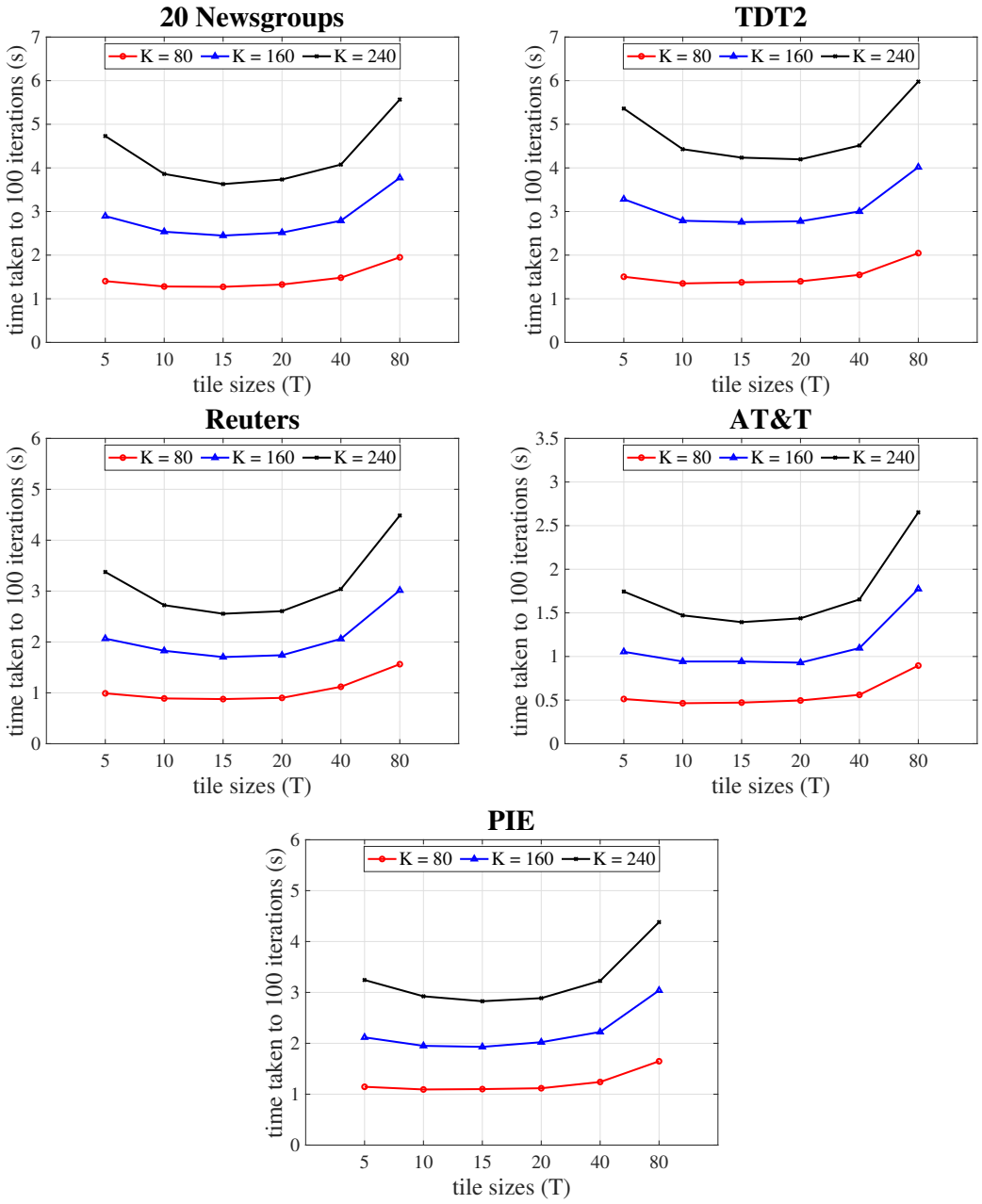


Figure 3.6: The time taken to reach 100 iterations when the tile size T is varied for different K on five datasets. Low-rank K is set to 80, 160 and 240, and T is varied for each K. X-axis: tile size; Y-axis: elapsed time to reach 100 iterations.

The tile size T affects the data movement volume and hence the performance. Equation 3.9 shows the data movement of our algorithm as a function of T. Consider the case when there is only one tile ($T = K$). In this case, there is no work associated with phase 1 (contributions to left) and phase 3 (contributions to the right) as the first term of Equation 3.9 will become zero. The total data movement of phase 2 is VK^2 which is very high. Now consider the other extreme where the tile size is 1 ($T = 1$). In this case, phases 1 and 2 have very high data movement ($> VK^2$). Thus, when T is high, the total data movements required for phases 1 and 3 are low, but phase 2 has high data movement. On the other extreme, when T is low, the total data movements for phases 1 and 3 are high, but phase 2 has low data movement. Hence, we expect the combined data movement for all the phases to decrease as T increases from 1 to some point and then the data movement will increase again as T approaches K. Figure 3.6 shows the performance results across different tile sizes for K = 80, 160 and 240 on five datasets. Since performance is correlated with data movement, the performance as a function of tile size T should show a similar trend with the performance shown in Figure 3.6.

$$\frac{d(vol(T))}{dT} = T^2(\frac{2}{\sqrt{C}} - 1) + K = 0 \tag{3.10}$$

$$T = \sqrt{K - \frac{2}{\sqrt{C}}} \tag{3.11}$$

In order to build a model to determine the tile size for a given K, the derivative of Equation 3.9 with respect to T is set it to zero as shown in Equation 3.10. The solution to Equation 3.10 is shown in Equation 3.11. As a result, for a machine with cache size of 35 MB, the tile sizes computed by our model are 8.94, 12.64 and 15.48 for K = 80, 160 and 240, respectively. As shown in Figure 3.6, tile sizes selected by our model (Equation 3.11)

are optimal/near optimal. For example, when $K = 240$, exhaustive evaluation shows that the best performance is achieved for T=15, which is very close to our model predicted tile size of $T = 15.48$.

3.7 Experimental Evaluation

This section compares the time to convergence and convergence rate of PL-NMF with various state-of-the-art techniques.

3.7.1 Benchmarking Machines

Table 3.3 shows the configuration of the benchmarking machines used for experiments. All the CPU experiments were run on an Intel Xeon CPU E5-2680 v4 running at 2.4 GHz with 128GB RAM. The GPU experiments were run on an NVIDIA Tesla P100 PCIE GPU with 16GB global memory.

Table 3.3: Machine configuration

Machine	Details
CPU	Intel(R) Xeon(R) CPU E5-2680 v4 (28 cores), 128GB ICC version 18.0.3
GPU	Tesla P100 PCIE (56 SMs, 64 cores/MP, 16GB Global Memory, 4 MB L2 cache) CUDA version 9.2.88

3.7.2 Datasets

For experimental evaluations we used three publicly available real-world text datasets – 20 Newsgroups[3], TDT2[3], Reuters[3]. In addition, in order to represent the audio-visual

[3] http://dengcai.zjulearning.org:8081/Data/TextData.html

context analysis in social media platforms, we used two image datasets – AT&T[4] and PIE[5]. 20 Newsgroups, TDT2 and Reuters are sparse matrices, and AT&T and PIE are dense matrices. The 20 Newsgroups dataset contains a document-term matrix in bag-of-words representation associated with 20 topics. TDT2 (Topic Detection and Tracking 2) dataset is a collection of text documents from CNN, ABC, NYT, APW, VOA and PRI. Reuters dataset is a collection of documents from the Reuters newswire in 1987. Both AT&T and PIE datasets contain images of faces in dense matrix format. The size of each image in AT&T and PIE datasets is 92×112 and 64×64 pixels, respectively. Table 3.4 shows the characteristics of each dataset.

Table 3.4: Statistics of datasets used in the experiments. V is the number of rows and D is the number of columns in non-negative matrix A. For the text datasets, V is the vocabulary size and D is the number of documents.

Dataset	V	D	**Total NNZ**	**Sparsity (%)**
20 Newsgroups	26,214	11,314	1,018,191	99.6567
TDT2	36,771	10,212	1,323,869	99.6474
Reuters	18,933	8,293	389,455	99.7519
AT&T	400	10,304	4,121,478	0.0030
PIE	11,554	4,096	47,321,408	0.0080

[4]https://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html

[5]http://dengcai.zjulearning.org:8081/Data/FaceDataPIE.html

3.7.2.1 NMF Implementations Compared

We evaluated PL-NMF on CPUs and GPUs with the state-of-the art parallel NMF implementations such as planc[6] by Kannan et al. [19, 36] and bionmf-gpu[7] by Mejía-Roa et al. [58]. The four implementations used in our comparisons are as follows:

- **planc-MU-cpu**: planc's OpenMP-based MU
- **planc-HALS-cpu**: planc's OpenMP-based HALS
- **planc-BPP-cpu**: planc's OpenMP-based ANLS-BPP
- **bionmf-MU-gpu**: bionmf-gpu's GPU-based MU

All of the competing CPU implementations, including planc-MU-cpu, planc-HALS-cpu and planc-BPP-cpu, and our PL-NMF-cpu, used Intel's Math Kernel Library (MKL) for all BLAS (Basic Linear Algebra Subprograms) operations. Similarly, all GPU implementations, including bionmf-MU-gpu and our PL-NMF-gpu, used NVIDIA's cuBLAS library for all types of BLAS operations.

3.7.2.2 Evaluation Metric

In order to evaluate the accuracy of different NMF models, we used the relative objective function $\sqrt{\frac{\sum_{vd}(A_{vd}-(WH)_{vd})^2}{\sum_{vd}(A_{vd})^2}}$ suggested by Kim et al. [38], where A_{vd} and $(WH)_{vd}$ denote the values of each element in an input matrix $A \in \mathbb{R}_+^{V\times D}$ and an approximated matrix $(WH) \in \mathbb{R}_+^{V\times D}$. The capability of each NMF model in minimizing the objective function can be obtained by measuring relative changes of objective value over iterations.

[6] https://github.com/ramkikannan/planc

[7] https://github.com/bioinfo-cnb/bionmf-gpu

3.7.3 Performance Evaluation

3.7.3.1 Convergence

Figure 3.7 shows the relative error as a function of elapsed time of various NMF implementations for different K values. To ensure fairness, the number of threads in all CPU implementations were tuned per dataset and the best performing configuration was selected. For each dataset, the same randomly initialized non-negative matrices were used for all CPU and GPU implementations. Since the bionmf-MU-gpu implementation does not allow the input matrix to be sparse, we only compared our GPU implementation with bionmf-MU-gpu on AT&T and PIE dense image datasets. PL-NMF-cpu and PL-NMF-gpu consistently outperformed existing state-of-the-art CPU and GPU implementations on all datasets. As reported in previous studies, FAST-HALS produced a better convergence rate than other NMF variants. MU and ANLS-BPP algorithms suffered from a lower convergence rate on both sparse and dense matrices. As shown in Figure 3.8, planc-HALS-cpu was the only implementation which was able to maintain the same solution quality as ours. However, our implementation converged faster.

3.7.3.2 Speedup

Compared to the planc-HALS-cpu, our PL-NMF-cpu achieved 3.07×, 3.06×, 5.81×, 3.02× and 3.07× speedup per iteration on the 20 Newsgroups, TDT2, Reuters, AT&T and PIE datasets with $K = 240$, respectively. As the relative error reduction per iteration is vastly different between MU and FAST-HALS algorithms, measuring the speedup per iteration between bionmf-MU-gpu and PL-NMF-gpu is not a fair comparison.

Figure 3.9 depicts the speedup of our PL-NMF-gpu over all CPU implementations. The x-axis in Figure 3.9 is relative error, and the y-axis is the ratio of elapsed time for all CPU

implementations to reach a relative error to elapsed time for PL-NMF-gpu to approach the same relative error. All of the points in Figure 3.9 are greater than one. This indicates that PL-NMF-gpu is faster than all of the competing implementations. For example, when the compared models, i.e., PL-NMF-cpu, planc-HALS-cpu, bionmf-MU-gpu and planc-MU-cpu, converged to 0.12 relative error, the parallel PL-NMF-gpu achieved 3.49×, 9.74×, 26.41× and 287.1× speedup on PIE dataset, respectively.

Table 3.5: Breakdown of elapsed time in seconds for updating W on the 20 Newsgroups dataset. DMV: Iterative Dense Matrix-Vector Multiplications; DMM: Dense Matrix-Dense Matrix Multiplication; SpMM: Sparse Matrix-Dense Matrix Multiplication.

Sequential FAST-HALS NMF	**elapsed time (s)**	**PL-NMF-cpu**	**elapsed time (s)**
SpMM	0.048	SpMM	0.048
DMM	0.002	DMM	0.002
DMV	2.039	Phase 1	0.005
		Phase 2 & 3	0.026

Table 3.5 shows the breakdown of elapsed time for each step in updating W. Both sequential FAST-HALS NMF and PL-NMF-cpu implementations use the same mkl_dcsrmm() and cblas_dgemm() routines for SpMM and DMM operations. In Table 3.5, SpMM corresponds to line 10 in Algorithm 13 and line 1 in Algorithm 14, which computes the same AH^T. Similarly, DMM corresponds to line 11 in Algorithm 13 and line 2 in Algorithm 14, which performs the same HH^T. The difference of updating W is that PL-NMF-cpu performs phases 1, 2 and 3 instead of iteratively performing DMV computations. As expected, the updating time of W is considerably decreased in our PL-NMF-cpu algorithm, indicating that the reformulation of the core-computations to matrix-matrix multiplication shows the benefit of our approach.

3.7.3.3 Scalability with the large size of K

Figure 3.10a compares the training time of our PL-NMF-cpu and planc-HALS-cpu on the large TDT2 dataset for different sizes of K. The T values are chosen based on Equation 3.11. Figure 3.10b shows that as K increases our speedup also increases. Our PL-NMF-cpu achieved approximately 2.70×, 6.94×, 7.91×, 11.07× and 12.93× speedup over planc-HALS-cpu when K = 200, 400, 800, 1600 and 3200, respectively.

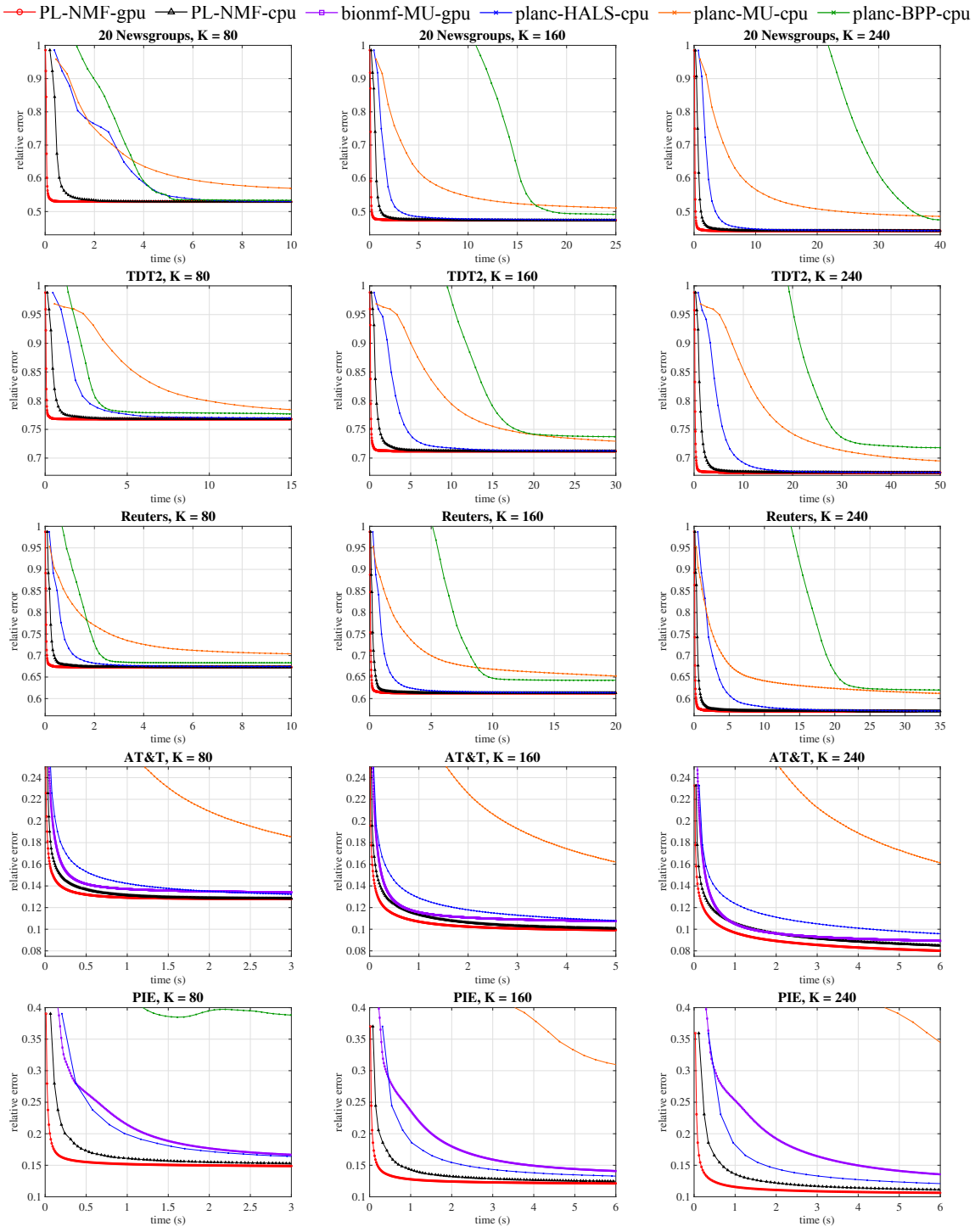


Figure 3.7: Relative objective value over time on five datasets. According to current model, the T values for $K = 80$, 160 and 240 are set to 10, 15 and 15, respectively. X-axis: elapsed time in seconds; Y-axis: relative error.

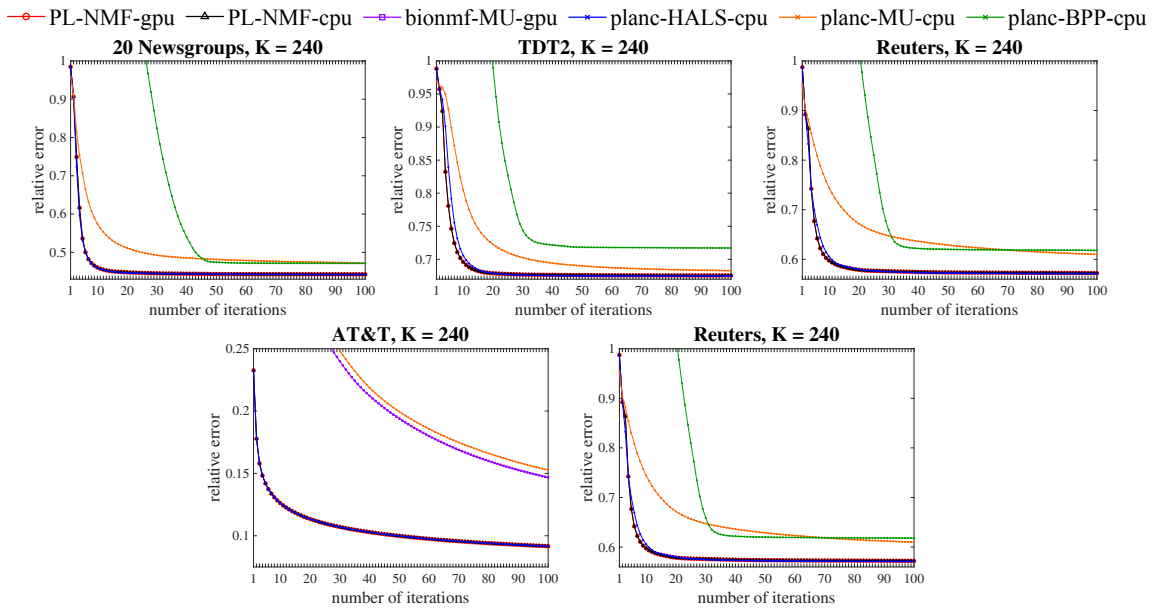


Figure 3.8: Comparison of convergence over iterations on five datasets, $K = 240$ and $T = 15$. X-axis: number of iterations; Y-axis: relative error.

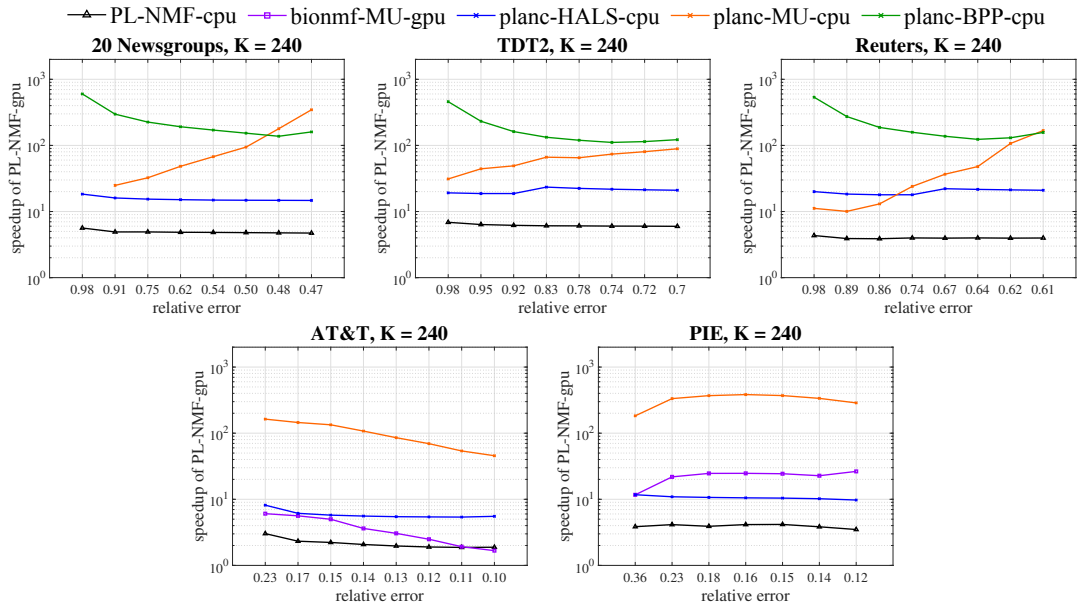
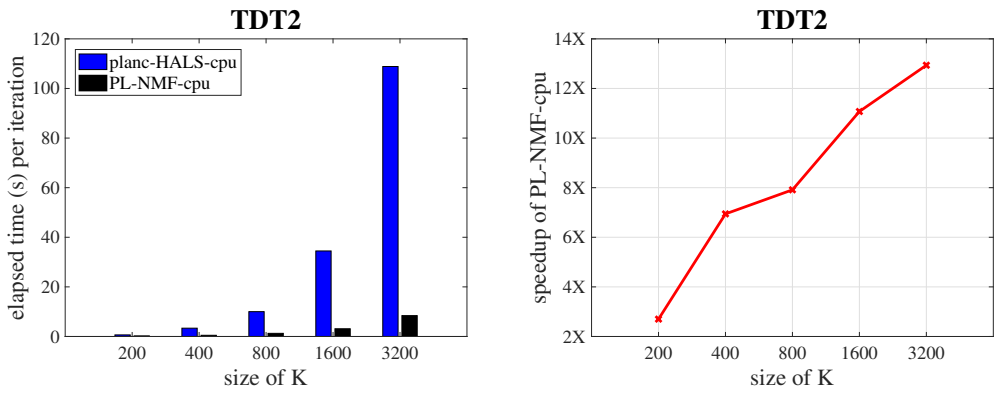


Figure 3.9: Speedup of PL-NMF-gpu over all CPU implementations on five datasets, $K = 240$ and $T = 15$.



(a) Time vs. Size of K (lower the better) (b) Speedup vs. Size of K (higher the better)

Figure 3.10: Speedup of PL-NMF-cpu over planc-HALS-cpu with the large size of K on TDT2 dataset. According to current model, the T values for $K = 200$, 400, 800, 1600 and 3200 are set to 10, 20, 25, 40 and 50, respectively.

3.8 Conclusion

In this chapter, we developed a HALS-based parallel NMF algorithm for multi-core CPUs and GPUs. The data movement overhead is a critical factor that affects performance. This chapter does a systematic analysis of data movement overheads associated with NMF algorithm to determine the bottlenecks. Our proposed approach alleviates the data movement overheads by enhancing data locality. Our experimental section shows that our parallel NMF achieves significant performance improvement over the existing state-of-the-art parallel implementations.

Chapter 4: Parallel Data-Local Training for Optimizing Word2Vec Embeddings for Word and Graph Embeddings

4.1 Introduction

The Word2Vec model proposed by Mikolov et al. [59, 60] belongs to the family of neural network-based static word embedding techniques that generate a dense vector in a low-dimensional embedding space for each word in a fixed vocabulary. The embeddings generated by Word2Vec include most of the important information about word similarity and word relatedness between other words in similar context. It can be utilized as a key feature in a wide range of applications such as natural language processing [63, 67, 106], bioinformatics [28, 61], and graph mining [26, 69].

The main focus of this chapter is the adaptation of the Word2Vec algorithm with a view towards reducing the amount of data movement from/to memory. Technology trends have made the cost of data movement increasingly dominant, both in terms of energy and time, over the cost of performing arithmetic operations in computer systems. However, the design and implementation of new algorithms in machine learning has been largely driven by a focus on the computational complexity. The inner core computation of the Word2Vec algorithm (explained in great detail later) involves a large number of dot-product operations between the embedding vectors representing different words in the corpus. The

dot-product computation is inherently memory-bandwidth limited because only two floating point operations are performed per pair of data elements read from memory. Since the computational peak performance of all current/emerging processors (multi-core CPUs, GPUs, FPGAs, etc.) greatly exceeds the peak memory bandwidth in words/second (often by a factor between ten and hundred), any algorithm that performs a large number of dot-products is inherently performance-handicapped, unless significant re-use of the processed vectors in caches can be achieved.

The Skip-gram based Word2Vec algorithm (described in detail later) processes words within small contiguous windows in the text, using dot-products of the center "focus" word with other words in the window in the process of trying to move the vector embedding of the word closer to that of the neighboring ones in the window (*Attraction*). Dot-products with a large number of randomly selected words in the sentence are also performed with the words in the window, seeking to move their vector embeddings further from them since they are not found co-located in the corpus (*Repulsion*). The random access to words in the latter negative-sampling process leads to very high data movement without any prospect of reuse of those randomly fetched vectors. We develop an adaptation to the way the negative sampling is performed that enables much higher data reuse for the fetched vectors. This is done by separating out the computations that seek to align a word closer to its neighbors in the windows from the negative-sampling computations, which are performed on mini-batches of words at a time, using a common set of randomly chosen words to move away from.

We develop efficient multi-core and GPU implementations of the adapted Word2Vec algorithm and perform extensive comparative evaluation of the new algorithms with a number of state-of-the-art implementations of Word2Vec on both platforms. We also show

the utility of the new Word2Vec implementation within the Node2Vec algorithm, which accelerates embedding learning for large graphs. We demonstrate significant reduction of data volume from/to main memory and improved performance over the current state-of-the-art. We also conduct extensive evaluation on the quality of the produced embeddings, for a number of application contexts. Overall, we find that we are able to achieve high performance with quality of results that are comparable or better than baselines.

Chapter 4 is organized as follows. In Section 4.2 we present the background on Word2Vec and Node2Vec. In Section 4.3, we present related prior works on parallelization of Word2Vec and graph embedding techniques. In Section 4.4, we present the high-level overview of skip-gram based Word2Vec algorithm with negative sampling method and conduct data movement analysis to identify the bottleneck. Sections 4.5 demonstrates details of our CPU and GPU implementations of PAR-Word2Vec. Also, we compare data movement costs of our PAR-Word2Vec with the original SG-NS based Word2Vec. In Section 4.6, we compare PAR-Word2Vec with existing state-of-the-art parallel Word2Vec implementations.

4.2 Background

4.2.1 Word2Vec

The basic intuition behind the Word2Vec model is that similar words tend to occur in similar contexts. Word2Vec is trained as a language model, in which the probability of a word in a text corpus depends on its surrounding context words. The Word2Vec model has two primary variants: (1) Continuous Bag-of-Words (CBOW), which predicts a center word using the average of the words in a fixed context window on either side, and (2) Skip-Gram (SG), which models pairwise probabilities of the center word with each of its context words individually. These two algorithms therefore result in different numbers of

parameter updates: CBOW only updates the center word once for a given context window, while SG updates the center word once for each context word. In practice the algorithms yield roughly equivalent performance ([81] find better results with CBOW; [48, 60] with SG); we follow prior work in focusing on the SG algorithm, which can (unlike CBOW) be interpreted as implicit factorization of the word co-occurrence matrix [47]. As defined by [59], the SG model seeks to maximize the average log probability $J(\theta)$ given a sequence of N word tokens in the text corpus.

$$J(\theta) = \frac{1}{N} \sum_{n=1}^{N} \sum_{-C \leq j \leq C, j \neq 0} \log p(w_{n+j}|w_n) \tag{4.1}$$

where C is the window size at the current center word w_i. Given by the softmax activation function, the probability of $p(w_{n+j}|w_n)$ is defined as:

$$p(w_{n+j}|w_n) = \frac{\exp(<\vec{\mathbf{w}}_{n+j}^{out}, \vec{\mathbf{w}}_n^{in}>)}{\sum_{v=1}^{V} \exp(<\vec{\mathbf{w}}_v^{out}, \vec{\mathbf{w}}_n^{in}>)} \tag{4.2}$$

where V is the size of the fixed vocabulary, $\vec{\mathbf{w}}_n^{in}$ and $\vec{\mathbf{w}}_v^{out}$ denote a word vector of a word w_n pulled from W_{in} matrix and a word vector of a word w_v pulled from W_{out} matrix, respectively. Furthermore, $<\vec{\mathbf{w}}_v^{out}, \vec{\mathbf{w}}_n^{in}>$ computes the inner product of word vectors $\vec{\mathbf{w}}_v^{out}$ and $\vec{\mathbf{w}}_n^{in}$.

SG and CBOW can be trained using one of two different objectives. Given a center word w_n and a true target word w_{n+j}, calculating the full softmax equation given in Equation 4.2 for every word update becomes rapidly intractable as the vocabulary size V increases. Instead of training probabilistically with the full softmax, the negative sampling (NS) method randomly chooses a small number of words in the vocabulary as negative targets, and trains using a log-bilinear objective derived from noise-contrastive estimation. The word vectors are therefore updated based on only the selected negative target words and a positive target word w_{n+j}, considerably reducing the number of computations while maintaining a good quality of solution [60]. Alternatively, the hierarchical softmax (HS) objective originally

proposed by Morin and Bengio [62], which can be viewed as an approximation of the full softmax function, can also be used to reduce the computational complexity of probabilistic training. Mikolov et al. [60] use the binary Huffman tree structure for hierarchical softmax so as to compute the probability distribution of a given word (root node) along with the path from root to vocabulary size of leaf nodes. Each path in the tree structure represents the relative probability to its child node. According to the nature of the binary Huffman tree, the root node can be rapidly reached to the leaf nodes when the words of root and leaf nodes are frequently used together in the corpus. This characteristic of hierarchical softmax makes neural network based models more efficient in terms of both elapsed time for training and accuracy. Typically, SG and CBOW are paired with each of the negative sampling (SG-NS) and hierarchical softmax (CBOW-HS), respectively. Hereafter, we concentrate on the skip-gram based Word2Vec with negative sampling (SG-NS) algorithm.

4.2.2 Node2Vec

The Node2Vec algorithm developed by Grover and Leskovec [26] is intended to learn vector-based representations of nodes in a graph, such that similar nodes (either nodes in similar subgraph structures or nearby nodes) have similar vectors. Algorithm 18 shows a pseudo-code for the Node2Vec algorithm. The Node2Vec consists of sampling R random walks of length L starting at each node in the graph; each of these walks ends up being a sentence for Word2Vec training. So for V unique nodes, Node2Vec generates a corpus of $V \times R$ sentences of L tokens (node IDs) each. Then Node2Vec runs off-the-shelf SG-NS based Word2Vec on a pre-processed corpus. In the pre-processed corpus, a single node and its neighbor node correspond to a word token w_n and w_{n+j} in Equation 4.1, respectively. In the Word2Vec algorithm, a node in the pre-generated graph dataset can be thought of as

Algorithm 18 Node2Vec algorithm

Input: Graph G: (V nodes, E edges and W: weights), p: return, q: in-out, R: number of random walks for each node, V: number of unique nodes, L: length of each walk, K: number of hidden units, C: window size
Output: W_{in}: $V \times K$ input node embedding matrix, W_{out}: $K \times V$ output node embedding matrix

$\pi \leftarrow$ Preprocess(G, p, q)
Generate new G': (V, E, π)
Initialize *random_walks*
for r $= 0$ **to** $R - 1$ **do**
  for v $= 0$ **to** $V - 1$ **do**
    Initialize *walk*
    for walk_iter $= 0$ **to** $L - 1$ **do**
      sample_node $\leftarrow$ Sampling(G', π)
      walk $\leftarrow$ [*walk*; sample_node]
    end for
    random_walks $\leftarrow$ [*random_walks*; *walk*]
  end for
end for
// *random_walks*: pre-generated dataset to use it as an input for Word2Vec
repeat
  W_{in},W_{out} $\leftarrow$ **SG-NS based Word2Vec**(*random_walks*, K, C)
until convergence

a word in the corpus. The effectiveness of the node embedding outputs, W_{in} and W_{out}, are then directly evaluated using the applications such as node classification and link prediction [26, 69, 91, 92]. Our contribution is to achieve a significant speedup on the embedding learning part in the Node2Vec algorithm by transforming the original Word2Vec into our new parallel Word2Vec algorithm. In fact, the Word2Vec embedding training portion of running Node2Vec on the BlogCatalog dataset accounts for $\frac{\text{training time of Word2Vec within the Node2Vec}}{\text{total training time of Node2Vec}} = \frac{72.635360(s)}{81.505412(s)} \approx 89.12\%$ of the total run time.

4.3 Related Work

4.3.1 Parallelization of Word2Vec Embeddings

Several efficient data-locality-enhanced schemes have recently been proposed for Word2Vec algorithms because the original Word2Vec algorithm imposes massive computations. As shown in Table 4.1, previous studies at parallelizing Word2Vec can be grouped by the types of machine, platform, and algorithm used in their implementations.

The original Word2Vec implementation released by Mikolov et al. [60] adopts HOGWILD! [75], a parallelization scheme where different word pairs are simultaneously processed across multiple threads. HOGWILD! partially overcomes the limitation of Stochastic Gradient Descent (SGD) optimization, which is inherently challenging to parallelize. However, adopting HOGWILD! in the Word2Vec algorithm has the inevitable limitation that it may lead to a race condition between different threads when updating the same word vector in the input and output matrices at the same time. In a shared-memory environment, Ji et al. [35] proposed a parallel Word2Vec (pWord2Vec) model that maximizes reuse of data structures by sharing negative samples within the same context window. The proposed scheme changes the original Level 1 BLAS operations into Level 3 BLAS operations;

Table 4.1: Previous studies on parallelization of Word2Vec. Context types – Skip-gram (SG) and Continuous Bag-of-Words (CBOW). Objective types – Negative Sampling (NS) and Hierarchical Softmax (HS)

Author	Machine	Platform	Algorithm
Mikolov et al. [60]	CPUs	Shared-memory	CBOW-NS, CBOW-HS, SG-NS, SG-HS
Ji et al. [35]	CPUs	Shared-memory	SG-NS
Vuurens et al. [96]	CPUs	Shared-memory	SG-HS
Simonton and Alaghband [86]	CPUs	Shared-memory	SG-NS, SG-HS
Rengasamy et al. [76]	CPUs	Shared-memory	SG-NS
Ji et al. [35]	CPUs	Distributed-memory	SG-NS
Ordentlich et al. [68]	CPUs	Distributed-memory	SG-NS
Simonton and Alaghband [86]	GPUs	Shared-memory	SG-NS, SG-HS
Bae and Yi [4]	GPUs	Shared-memory	CBOW-NS, CBOW-HS, SG-NS, SG-HS
Canny et al. [9]	GPUs	Shared-memory	SG-NS, SG-HS

thereby, the number of updates and communication cost between threads are considerably reduced. The main difference between pWord2Vec and our scheme is the maximum number of input words (columns in yellow colored matrix M_{in} in Figure 4.5) sharing the same negative samples. In pWord2Vec, the maximum number is restricted to the total number of input words in each context window, which is $2 \times window\ size + 1$. However, in our approach, the number of columns in matrix M_{in} is maximized using mini-batch size of all words within each sentence, regardless of context window, to share the same negative samples. Vuurens et al. [96] introduced an efficient caching strategy for updating vectors based on the SG-HS algorithm. They maintained local copies of the most frequently used inner nodes in the Huffman tree structure to maximize data reuse in cache and reduce the number of memory conflicts. Recently, Rengasamy et al. [76] introduced a context combining approach (pSGNScc) to further optimize the data reuse in pWord2Vec. In pSGNScc, multiple correlated context windows share not only negative samples but also positive samples. Given the

words in the current context window, pSGNScc utilizes a pre-generated inverse index table to find related windows according to the word occurrences in the entire corpus. In their experiments, pSGNScc achieved 1.28X speedup compared to pWord2Vec. In Ordentlich et al. [68], a distributed SG-NS based Word2Vec algorithm was proposed in order to reduce the high training latency and network bandwidth with large-scale datasets. On top of a Hadoop system, multiple servers learn the distributed word vectors to achieve the higher throughput in parallel. In the GPU platform, Simonton and Alaghband [86] developed both SG-HS and SG-NS based Word2Vec algorithms using shared memory registers and in-warp shuffle operations on GPUs. Within a thread block, the use of shared memory registers significantly reduces the data accessing time compared to the global memory access. Based on roofline design, Canny et al. [9] developed a system called BIDMach to improve the performance of different machine learning algorithms. As it was reported that their SG-NS based GPU implementation suffers from the quality of word embeddings (see Table 2 in [86]), their GPU implementation was not included as a competing model. Bae and Yi [4] implemented four variants of Word2Vec model using GPUs. Since the computations of nested loops which iterate over the number of hidden units are dominant in the original Word2Vec algorithm, the number of threads used in their GPU implementation is same as the number hidden units. One of their variants, SG-NS based GPU implementation, was used as a baseline for our new GPU implementation.

4.3.2 Word2Vec based Graph Embeddings

Recently, several graph embedding algorithms based on random walk sampling have been developed by utilizing Skip-gram based Word2Vec model to find a low-dimensional latent representation for each node on the graph [26, 69]. It is assumed that there exist

similarities between the nodes connected to each other on the random walks. DeepWalk[8] proposed by Perozzi et al. [69] samples random walks over the graph and maps them into the Skip-gram based Word2Vec model for training. To generate node embeddings, they used SG-HS algorithm instead of SG-NS algorithm used in the Node2Vec [26].

4.4 SG-NS based Word2Vec Algorithm

In this section, we describe the original SG-NS based Word2Vec algorithm [60] and main factors that limits the performance.

4.4.1 Overview of SG-NS Based Word2Vec

$$\log \sigma(<\vec{\mathbf{w}}^{out}_{n+j}, \vec{\mathbf{w}}^{in}_{n}>) + \sum_{t=1}^{T} \log \sigma(-<\vec{\mathbf{w}}^{out}_{t}, \vec{\mathbf{w}}^{in}_{n}>) \tag{4.3}$$

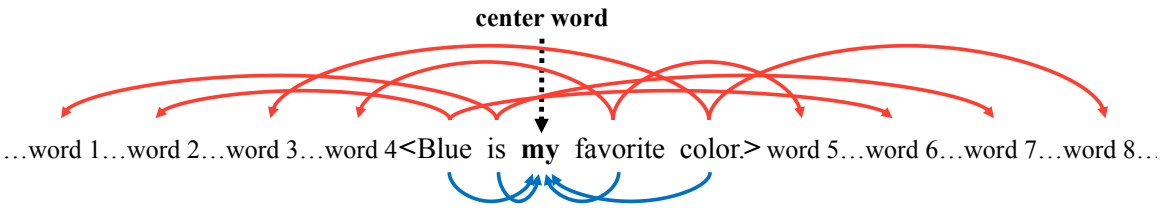
In the SG-NS based Word2Vec model proposed by Mikolov et al. [60], given an input word w_n, a positive target word w_{n+j}, and randomly sampled negative target words $w_{1:T}$, the log probability $p(w_{n+j}|w_n)$ is replaced by the loss calculation in Equation 4.3, where T is the number of negative samples (targets) and $\sigma(x)$ denotes the sigmoid logistic function defined as $\sigma(x) = \frac{1}{1+\exp(-x)}$. This term is maximized over every (w_n, w_{n+j}) pair in the corpus. Instead of using vocabulary size of V words as negative targets, randomly selecting only T words (usually $5 \leq T \leq 20$) as negative targets considerably reduces the computational complexity while maintaining good quality. Algorithm 19 shows pseudo-code for the original SG-NS based word2vec implementation [60]. It uses context words as inputs (line 11) and uses the word at the center of the context window as target (line 15), along with

[8]https://github.com/phanein/deepwalk

Algorithm 19 SG-NS based Word2Vec algorithm

Input: $corpus$: S sentences and a sequence of L word tokens in each sentence, V: the number of unique words, K: the number of hidden units, C: window size, T: the number of negative samples, α: learning rate, W_{in}: $(V \times K)$ input embedding matrix, W_{out}: $(V \times K)$ output embedding matrix
Output: W_{in}: $(V \times K)$ input embedding matrix

Initialize W_{in} and W_{out} with random numbers
repeat
  for sid $= 0$ **to** $S - 1$ **do**
    $L \leftarrow$ number of word tokens in sentence sid
    // Update W_{out} and W_{in} with both positive and negative samples
    for i $= 0$ **to** $L - 1$ **do**
      center_word $\leftarrow corpus$[sid][i]
      $C_{rand} \leftarrow random\_uniform()$ % C
      for j $= C_{rand}$ **to** $(2 \times C - C_{rand})$ **do**
        if j ! $= C$ **then**
          input $\leftarrow corpus$[sid][i$-C$+j]
          Initialize $temp$[0:$K-1$] to 0
          for t $= 0$ **to** T **do**
            if t $==$ 0 **then**
              target $\leftarrow$ center_word, label $\leftarrow 1$
            else
              target $\leftarrow random\_uniform()$ % V, label $\leftarrow 0$
            end if
            sum $\leftarrow 0$
            for k $= 0$ **to** $K - 1$ **do**
              sum $\leftarrow$ sum $+ W_{in}$[input][k] $\times W_{out}$[target][k]
            end for
            grad $\leftarrow$ (label $- sigmoid$(sum)) $\times \alpha$
            for k $= 0$ **to** $K - 1$ **do**
              $temp$[k] $\leftarrow temp$[k] $+$ grad $\times W_{out}$[target][k]
              W_{out}[target][k] $\leftarrow$
                W_{out}[target][k] $+$ grad $\times W_{in}$[input][k]
            end for
          end for
          for k $= 0$ **to** $K - 1$ **do**
            W_{in}[input][k] $\leftarrow W_{in}$[input][k] $+ temp$[k]
          end for
        end if
      end for
    end for
  end for
until convergence



Input	blue	blue	blue	is	is	is	favorite	favorite	favorite	color	color	color
Target	**my**	negative word 2	negative word 6	**my**	negative word 1	negative word 7	**my**	negative word 4	negative word 5	**my**	negative word 3	negative word 8
Update type	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**

Figure 4.1: An example of updates in the original Word2Vec for the sentence "blue is my favorite color," where "my" is the current center word, and window size and the number of negative samples are both 2. The update types of "1" and "0" indicate *Attraction* and *Repulsion* updates, respectively.

the negative samples (line 17)[9]. The input word vectors are pulled from the W_{in} embedding matrix that is kept after training, and the target word vectors are pulled from W_{out} embedding matrix, which is discarded after training. An input vector and a target vector are multiplied in lines 20 to 22, in order to compute the gradient in line 23. While training each epoch, based on this gradient, W_{out} and W_{in} matrices are updated (lines 24-28 and lines 30-32 for W_{out} and W_{in}, respectively).

Figure 4.1 depicts an example of sequential updates for SG-NS based Word2Vec with the sentence "blue is my favorite color," where "my" is the center word of the context window. When the center word "my" is used as a positive target word, its surrounding words "blue", "is", "favorite", and "color" are its input words. Then, for each pair of the center word "my" and one of its input words, two negative target words are randomly

[9] [59] defined skip-gram using context words as targets and center words as input; however, the reference Word2Vec implementation operates as we have described. This difference only changes the order of updates over the full sequence; in our experiments, Word2Vec had equivalent performance using both definitions of skip-gram.

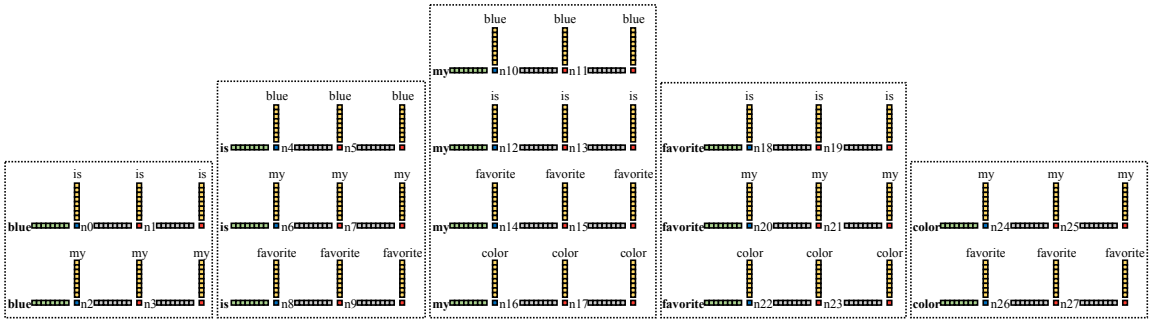


Figure 4.2: Full sequential updates required for the sentence "blue is my favorite color" in the original Word2Vec. In each vector-vector multiplication, the target word vectors on the left horizontal vectors (i.e., green and gray colored vectors) are pulled from W_{out} matrix, and the right vertical vectors (i.e., yellow colored vectors), input word vectors, are pulled from W_{in} matrix.

chosen to be negative samples (e.g., “word 2” and “word 6” are selected as negative targets where “blue” is the input word). Hereafter, an update with the center word (positive target word) and an input word is called an ***Attraction*** update (blue arrow in Figures 4.1 and 4.3), and an update with the negative target word and an input word is called ***Repulsion*** update (red arrow in Figures 4.1 and and 4.3). Figure 4.2 demonstrates the full sequential processes of the vector-vector multiplications for computing gradients, at each center word in a context window, required for both *Attraction* and *Repulsion* updates, where n0...n27 indicate randomly selected negative words.

4.4.2 Data Movement Analysis for SG-NS Based Word2Vec

$$S(L(1+2(C-C_{rand})(1+4K+8K(T+1)))) \tag{4.4}$$

$$S(L(2(C-C_{rand})(1+4K+8KT))) \tag{4.5}$$

To identify the data movement cost of the original SG-NS based Word2Vec algorithm, we individually analyzed each line in Algorithm 19. The outer loop in line 3 iterates over all the S sentences in the corpus, and the loop in line 6 iterates over all the L word tokens in each sentence. Each loop in line 6 reads a center word (positive target word) from $corpus$ (1 read). According to the randomly selected context window size, the loop in line 9 iterates over $2C - 2C_{rand}$ surrounding words. Each surrounding input word is read from $corpus$ (1 read), and K size of $temp$ array is initialized by zero (K writes). Then the loop in line 13 calculates vector updates by iterating over one positive and T negative samples ($T + 1$ iterations). After selecting a negative sample, the loop in line 20 multiplies the current input word vector and target word vector ($2K$ reads). The gradient is then computed in line 23. The loop in 24 updates W_{out} by accessing $temp$, W_{out} and W_{in} arrays ($4K$ reads and $2K$ writes). After the completion of the update calculation loop, the loop in line 30 applies the updates to W_{in} ($2K$ reads and K writes). Overall, the total data movement required for the original SG-NS based algorithm is shown in Equation 4.4. As shown in Equation 4.5, the data movement cost for only *Repulsion* updates can be simply obtained by excluding the data movement of line 7 and subtracting one iteration of the loop in line 13 from the total data movement. It is evident that the main bottleneck of SG-NS based Word2Vec algorithm is *Repulsion* updates. More specifically, the data movement overhead is closely associated with the several vector-vector multiplications within the loop in line 9. In the next section, we present a high-level overview and details of our new parallel Word2Vec algorithm called **P**arallel decoupled **A**ttraction-**R**epulsion based **Word2Vec** (**PAR-Word2Vec**) based on Skip-gram with a novel negative sampling method. In light of our main goal which is to enhance the algorithm in regard of data locality, our approach significantly alleviates data movement

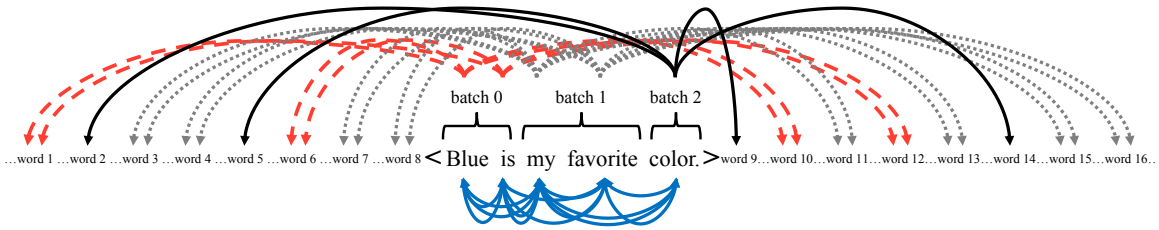
overhead involving *Repulsion* updates. We also compare the data movements required for our PAR-Word2Vec with original SG-NS based Word2Vec algorithms.

4.5 Parallel Decoupled Attraction-Repulsion based Word2Vec Algorithm

4.5.1 Overview of PAR-Word2Vec

In the original Word2Vec algorithm, majority of training time is spent on the process associated with negative sampling. Our main contribution is to improve the performance of negative sampling method by increasing data reuse. In order to use much larger size of matrix with negative sampling, we first decouple the full loss calculation in Equation 4.3 with positive samples, $\log \sigma(<\vec{\mathbf{w}}_{n+j}^{out}, \vec{\mathbf{w}}_{n}^{in}>)$, and negative samples, $\sum_{t=1}^{T} \log \sigma(-<\vec{\mathbf{w}}_{t}^{out}, \vec{\mathbf{w}}_{n}^{in}>)$, within each sentence. Thereafter, all of the *Attraction* updates required for each sentence are batched, and then followed by a batch of *Repulsion* updates. After the completion of *Attraction* updates, all words included in each sentence are divided into multiple mini-batches, and the words in the same mini-batch share the randomly chosen words for *Repulsion* updates. For example, in Figure 4.3, two input words in the same mini-batch (e.g., "blue" and "is" in batch 0) share the negative target words (e.g., "word 1", "word 6", "word 10" and "word 12"), when the mini-batch size is set to 2. If the mini-batch size is larger than or equals to the number of words in the sentence, there is only one mini-batch and all the words within the sentence share the same negative samples.

It is important to determine an appropriate mini-batch size to reduce training time without overly affecting the convergence rate. To analyze the impact of different mini-batch sizes on both convergence rate and training time, we conducted an empirical evaluation with the large text dataset. As depicted in Figures 4.4a and 4.4b, increasing the mini-batch size



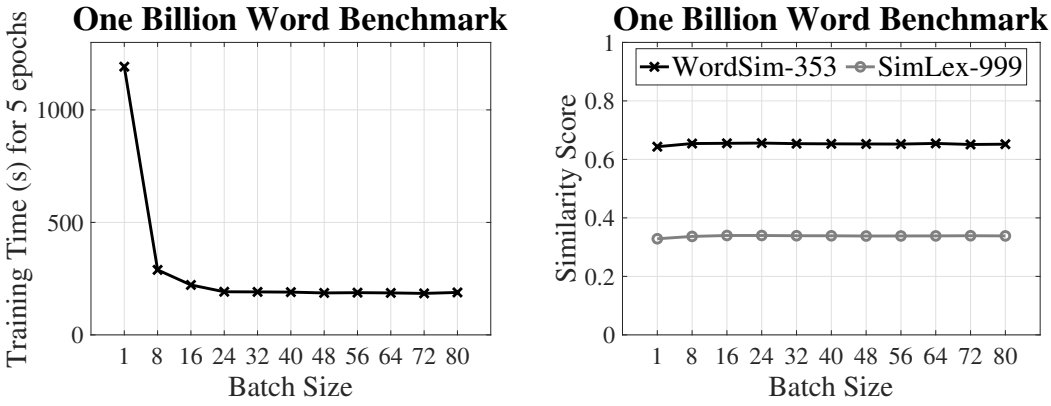
Input	is	my	blue	my	favorite	blue	is	favorite	color	is	my	color	my	favorite
Target	blue	blue	is	is	is	my	my	my	my	favorite	favorite	favorite	color	color
Update type	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(a) Update of *Attraction* phase

Batch index	batch 0								batch 1														batch 2			
Input	blue							is	my													favorite	color			
Target	4 shared negative words (word 1, 6, 10 and 12)								8 shared negative words (word 3, 4, 7, 8, 11, 13, 15 and 16)														4 shared negative words (word 2, 5, 9 and 14)			
Update type	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**

(b) Update of *Repulsion* phase

Figure 4.3: An example of full *Attraction* and *Repulsion* updates in the PAR-Word2Vec. In the sentence "blue is my favorite color," where the mini-batch size is set to 2, the input words in each mini-batch share the same negative target words. As marked by same colored arrows (red, gray and black), the words repelled by each mini-batch are shared negative target words (e.g., "word 1", "word 6", "word 10" and "word 12" are the shared negative targets for batch 0).



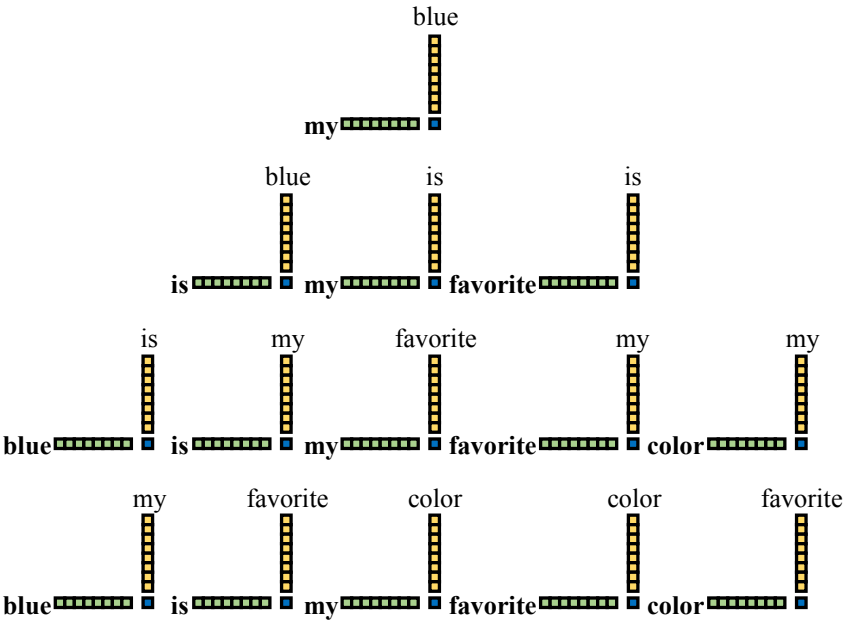
(a) Time vs. Batch size

(b) Quality vs. Batch size

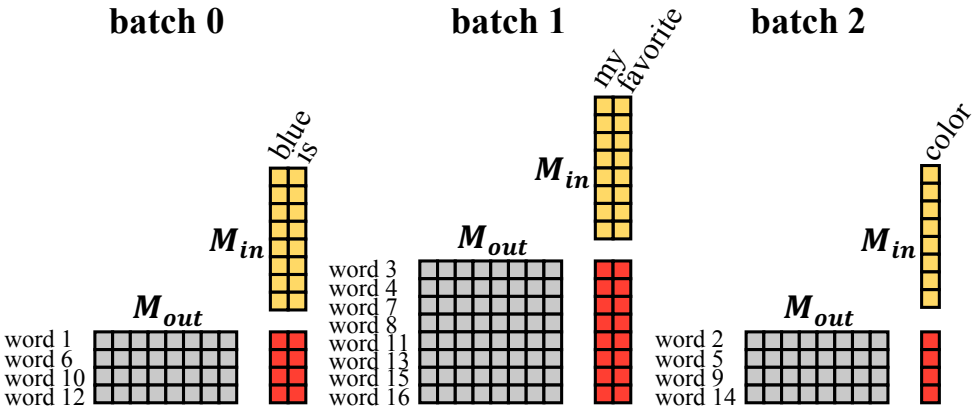
Figure 4.4: Comparison of averaged convergence and training time in second for 5 executions with different mini-batch sizes on PAR-Word2Vec-cpu on the One Billion Word Benchmark dataset, where K=128.

significantly reduces training time because of lower computational complexity without any loss of model quality. Furthermore, in order to perform the *Repulsion* phase as similar to the original SG-NS based algorithm, a different number of shared negative samples is drawn for each mini-batch according to the first word's sentence position in the mini-batch.

Figure 4.5 illustrates the full set of operations required for decoupled *Attraction* and *Repulsion* phases in PAR-Word2Vec. In Figure 4.5a, the vector-vector multiplications are performed for the *Attraction* phase; this is exactly the same computation as the original SG-NS algorithm. We maintain the *Attraction* phase as the original SG-NS algorithm, since the *Attraction* phase demands an extremely small computation compared to the *Repulsion* phase. Based on the mini-batch processing, however, the *Repulsion* phase is reformulated with matrix multiplications, as shown in Figure 4.5b. For each mini-batch, its shared negative target vectors pulled from the W_{out} matrix can be combined to form a temporary target matrix M_{out}. Likewise, input vectors pulled from W_{in} matrix are combined to generate a temporary



(a) Decoupled *Attraction* phase



(b) Decoupled *Repulsion* phase

Figure 4.5: Decoupled full *Attraction* and *Repulsion* updates required in the PAR-Word2Vec with the sentence "blue is my favorite color". In the decoupled *Repulsion* phase, the number of shared negative samples for the mini-batch (i.e., the number of rows in gray colored matrix) is determined by the first word's position in the mini-batch over the sentence. The mini-batches located at the start or end of the sentence (e.g., batch 0 or batch 2) have a less amount of shared negative samples, since the context window size of the input words contained in these mini-batches is smaller than that of the middle mini-batch (e.g., batch 1).

input matrix M_{in} for the current mini-batch. Then an efficient matrix multiplication of two matrices M_{out} and M_{in} is performed for computing the gradients. Given the result matrix of gradients, two additional matrix multiplications compute the update values for corresponding input and target vectors which will be accumulated to W_{out} and W_{in} matrices.

4.5.2 Details of Parallel CPU implementation

Algorithm 20 Parallel CPU implementation (**PAR-Word2Vec-cpu**)

Input: $corpus$: S sentences and a sequence of L word tokens in each sentence, V: the number of unique words, K: the number of hidden units, C: window size, T: the number of negative samples, B: mini-batch size, O: $2 \times C \times T$, α: learning rate, W_{in}: $(V \times K)$ input embedding matrix, W_{out}: $(V \times K)$ output embedding matrix, M_{in}: $(B \times K)$ input matrix for current batch, M_{out}: $(O \times K)$ output matrix for current batch, M_{grad}: $(O \times B)$ result matrix of $M_{out} \cdot M_{in}^{\mathrm{T}}$ for current batch, M_{in_update}: $(B \times K)$ updated input matrix for current batch, M_{out_update}: $(O \times K)$ updated output matrix for current batch, *shared_ns*: O size of shared negative samples array for current batch
Output: updated W_{in}: $(V \times K)$ input embedding matrix

1: Initialize W_{in} and W_{out} with random numbers
2: $numT \leftarrow$ number of threads
3: **#pragma omp parallel num_threads**($numT$)
4: Distribute S sentences of $corpus$ into $numT$ threads
5: tId $\leftarrow$ thread id
6: $S_{pt} \leftarrow \frac{S}{numT}$
7: **repeat**
8: **for** sid = tId $\times$ S_{pt} **to** (tId + 1) $\times$ $S_{pt} - 1$ **do**
9: $L \leftarrow$ number of word tokens in sentence sid
10: $W_{in}, W_{out} \leftarrow$ **Attraction**($corpus$, W_{in}, W_{out}, L, K, C)
11: W_{in}, $W_{out} \leftarrow$ **Repulsion**($corpus$, W_{in}, W_{out}, M_{in}, M_{out}, M_{grad}, M_{in_update}, M_{out_update}, *shared_ns*, L, K, C, T, B)
12: **end for**
13: **until** convergence

The pseudo-codes for our parallel CPU implementation are shown in Algorithm 20, 21 and 22. The sets of sentences in the corpus are disjointly distributed for processing

Algorithm 21 Attraction() phase on **PAR-Word2Vec-cpu**

```
1: // Update W_out and W_in with only positive samples
2: label ← 1
3: for i = 0 to L − 1 do
4:    center_word ← corpus[sid][i]
5:    C_rand ← random_uniform() % C
6:    for j = C_rand to (2 × C − C_rand) do
7:       if j != C then
8:          input ← corpus[sid][i−C+j]
9:          target ← center_word
10:         sum ← 0
11:         for k = 0 to K − 1 do
12:            sum ← sum + W_in[input][k] × W_out[target][k]
13:         end for
14:         grad ← (label − sigmoid(sum)) × α
15:         #pragma simd
16:         for k = 0 to K − 1 do
17:            temp ← grad × W_out[target][k]
18:            W_out[target][k] ← W_out[target][k] + grad × W_in[input][k]
19:            W_in[input][k] ← W_in[input][k] + temp
20:         end for
21:      end if
22:   end for
23: end for
```

Algorithm 22 Repulsion() phase on **PAR-Word2Vec-cpu**

1: **// Update W_{out} and W_{in} with only shared negative samples**
2: label $\leftarrow$ 0, num_batch $\leftarrow$ $(L+B-1)$ / B
3: **for** batch_id $=$ 0 **to** num_batch $-$ 1 **do**
4: min_pos $\leftarrow$ batch_id $\times$ B, max_pos $\leftarrow$ min(min_pos $+$ $B-1$, $L-1$)
5: current_batch_size $\leftarrow$ max_pos $-$ min_pos $+$ 1
6: **if** (min_pos $<=$ $C-1$) || ($L-1-$ min_pos $<=$ $C-1$) **then**
7: $C_{rep} \leftarrow random\_uniform()$ % C
8: **else**
9: $C_{rep} \leftarrow random\_uniform()$ % $(2 \times C)$
10: **end if**
11: num_shared_negatives $\leftarrow$ $C_{rep} \times T$
12: **for** n $=$ 0 **to** num_shared_negatives $-$ 1 **do**
13: $shared\_ns$[n] $\leftarrow$ $random\_uniform()$ % V
14: **end for**
15: **memcpy**(M_{in}[0:current_batch_size-1][0:K-1], $W_{in}[corpus$[sid][min_pos:min_pos+current_batch_size-1]][0:K-1])
16: **memcpy**(M_{out}[0:num_shared_negatives-1][0:K-1], $W_{out}[shared\_ns$[0:num_shared_negatives-1]][0:K-1])
17: **// Efficient matrix-matrix multiplication of M_{out} and M_{in}^{T}**
18: M_{grad}[0:num_shared_negatives$-$1][0:current_batch_size$-$1] $\leftarrow$ **sgemm**(M_{out}[0:num_shared_negatives$-$1][0:$K-1$], M_{in}^{T}[0:current_batch_size$-$1][0:$K-1$])
19: **for** n $=$ 0 **to** num_shared_negatives $-$ 1 **do**
20: **#pragma simd**
21: **for** b $=$ 0 **to** current_batch_size $-$ 1 **do**
22: output $\leftarrow$ M_{grad}[n][b]
23: grad $\leftarrow$ (label $-$ $sigmoid$(output)) $\times$ α
24: M_{grad}[n][b] $\leftarrow$ grad
25: **end for**
26: **end for**
27: **// Efficient matrix-matrix multiplication of M_{grad} and M_{in}**
28: M_{out_update}[0:num_shared_negatives$-$1][0:$K-1$] $\leftarrow$ **sgemm**(M_{grad}[0:num_shared_negatives$-$1][0:current_batch_size$-$1], M_{in}[0:current_batch_size$-$1][0:$K-1$])
29: **// Efficient matrix-matrix multiplication of M_{grad}^{T} and M_{out}**
30: M_{in_update}[0:current_batch_size$-$1][0:$K-1$] $\leftarrow$ **sgemm**(M_{grad}^{T}[0:num_shared_negatives$-$1][0:current_batch_size$-$1], M_{out}[0:num_shared_negatives$-$1][0:$K-1$])
31: **for** b $=$ 0 **to** current_batch_size $-$ 1 **do**
32: **#pragma simd**
33: **for** k $=$ 0 **to** K $-$ 1 **do**
34: $W_{in}[corpus$[sid][min_pos+b]][k] $+=$ M_{in_update}[b][k]
35: **end for**
36: **end for**
37: **for** n $=$ 0 **to** num_shared_negatives $-$ 1 **do**
38: **#pragma simd**
39: **for** k $=$ 0 **to** K $-$ 1 **do**
40: $W_{out}[shared\_ns$[n]][k] $+=$ M_{out_update}[n][k]
41: **end for**
42: **end for**
43: **end for**

by different threads (line 4 in Algorithm 20). For each sentence, a *Repulsion* phase starts to process after the completion of all *Attraction* updates. As shown in Algorithm 21, a decoupled *Attraction* phase is performed in the same way as the original Word2Vec algorithm. Algorithm 22 shows the pseudo-code for the decoupled *Repulsion* phase. In the *Repulsion* phase, the number of shared negative samples for the current mini-batch is determined by the first word's position in the mini-batch (lines 4-11). If the first word of the mini-batch is located at the start or end of the sentence, the corresponding mini-batch requires fewer shared negative samples, since the context window sizes of given words are small compared to the other words in the middle of the sentence. The shared negative targets are the randomly selected in lines 12-14. Next, the temporary matrices M_{in} and M_{out} for keeping input word vectors and shared negative target vectors are formed by pulling corresponding weight vectors from W_{in} and W_{out}, respectively (lines 15 and 16). Then all three matrix-matrix multiplications required for current mini-batch are efficiently performed using the **cblas_sgemm**() BLAS-3 routine provided in Intel's Math Kernel Library (MKL). In line 18 in Algorithm 22, the first matrix-matrix multiplication of two matrices, M_{out} and M_{in}^{T}, is performed to compute the gradients for the current mini-batch (lines 18-26). The outputs of $M_{out} \cdot M_{in}^{T}$ are stored in an additional matrix M_{grad}. The update values for W_{out} are then computed by performing the second matrix-matrix multiplications of matrices M_{grad} and M_{in} (line 28) and storing the results in M_{out_update}. Similarly, the update values for W_{in} can be obtained by performing the third matrix-matrix multiplication, with matrices M_{grad}^{T} and M_{out} (line 30). After completing all computations from the current mini-batch, the corresponding update values contained in M_{out_update} and M_{in_update} are accumulated to the main data structures, W_{out} and W_{in} (lines 31-42).

4.5.3 Details of Parallel GPU implementation

Algorithm 23 Parallel GPU implementation on host (**PAR-Word2Vec-gpu**)

Input: $corpus$: a sequence of N word tokens for all sentences, N: the total number of word tokens in corpus, S: the total number of sentences, V: the number of unique words, K: the number of hidden units, C: window size, T: the number of negative samples, B: batch size, O: $2 \times C \times T$, α: learning rate, γ: hyper-parameter for setting the number of thread blocks, sen_ptr: $S+1$ size of vector for holding the starting word index of each sentence over the $corpus$, W_{in}: $(V \times K)$ input embedding matrix, W_{out}: $(V \times K)$ output embedding matrix, M_{in}: $(B \times K \times$ num_blocks) input matrix for current batch, M_{out}: $(O \times K \times$ num_blocks) output matrix for current batch, M_{grad}: $(O \times B \times$ num_blocks) result matrix of $M_{out} \cdot M_{in}^{\mathrm{T}}$ for current batch, M_{in_update}: $(B \times K \times$ num_blocks) updated input matrix for current batch, M_{out_update}: $(O \times K \times$ num_blocks) updated output matrix for current batch, $shared\_C_{rand}$: (num_blocks) size of array for holding shared random window size for current batch, $shared\_ns$: $(O \times$ num_blocks) size of vector for holding shared negative samples for current batch
Output: W_{in}: $(V \times K)$ input embedding matrix

1: Allocate device memory for $corpus$, sen_ptr, W_{in}, W_{out}, M_{in}, M_{in_update}, M_{out}, M_{out_update}, M_{grad}, $shared\_C_{rand}$ and $shared\_ns$ using cudaMalloc
2: Copy host memory to device memory for $corpus$, sen_ptr, W_{in} and W_{out} using cudaMemcpy
3: num_blocks $\leftarrow \gamma \times 56$
4: num_threads $\leftarrow K$
5: num_sen_per_block $\leftarrow$ (S+num_blocks-1) / num_blocks
6: $\lambda \leftarrow \alpha /$total number of epochs
7: **repeat**
8: SG_Shared_NS<<<num_blocks, num_threads>>>
(num_sen_per_block, $corpus$, sen_ptr, W_{in}, W_{out}, M_{in}, M_{in_update}, M_{out}, M_{out_update}, M_{grad}, $shared\_C_{rand}$, $shared\_ns$, K, C, T, B, O)
9: $\alpha \leftarrow \alpha - \lambda$
10: **until** convergence
11: Copy device memory to host memory for W_{in} using cudaMemcpy

Algorithm 23, 24, 25 and 26 show the pseudo-code for GPU implementation. In order to achieve massive parallelism on GPUs, we divide the corpus of sentences into different thread blocks (line 5 in Algorithm 23). Hence, the thread blocks are processed in parallel,

Algorithm 24 SG_Shared_NS() kernel on **PAR-Word2Vec-gpu**

```
1: __shared__ shared_vector[1024]
2: start_sen_id_per_block ← blockIdx.x × num_sen_per_block
3: end_sen_id_per_block ← min(start_sen_id_per_block + num_sen_per_block, total_num_sen)
4: for sid = start_sen_id_per_block to end_sen_id_per_block do
5:     start_idx ← sen_ptr[sid]
6:     end_idx ← sen_ptr[sid+1]
7:     L ← end_idx - start_idx // L: length of sentence
8:     Attraction()
9:     __syncthreads()
10:    Repulsion()
11: end for
```

whereas multiple sentences are processed sequentially within each thread block (lines 4-11 in Algorithm 24). In line 3 in Algorithm 23, we launch $\gamma \times 56$ thread blocks as there are 56 Streaming Multiprocessors (SMs) on an NVIDIA Pascal P100 GPU. γ is the overbooking factor used to maintain good load balance. Note that the parameter γ is varied according to the total number of sentences over the corpus. In order to simultaneously update K dimensions of each word vector involved in the *Attraction* and *Repulsion* phases, K threads are selected within a thread block (line 4 in Algorithm 23). In *Attraction* phase, to obtain the gradient of vector-vector multiplications of W_{in} and W_{out}, a warp-level reduction is performed by warp shuffling primitives (lines 10-21 in Algorithm 25). All the threads in the warp read $32/K$ of the word vectors (K dimensions) from W_{in} and W_{out} and perform multiplications. Then the computed gradients are stored in a single space of shared-memory (shared_vector[0]) since all the threads in each thread block must use the same gradient (lines 22-24) before computing the update values. The update values for each word vector are then accumulated to W_{in} and W_{out} in global memory using atomic operations, because multiple thread blocks can update the same word vector at the same time (lines 27 and

Algorithm 25 Attraction() device function on **PAR-Word2Vec-gpu**

label $\leftarrow$ 1, f $\leftarrow$ 0
for sen_pos = start_idx **to** end_idx **do**
  center_word $\leftarrow$ $corpus$[sen_pos]
  $C_{rand} \leftarrow curand\_uniform() \times C$
  for j = C_{rand} **to** $(2 \times C - C_{rand})$ **do**
    if j ! = C **then**
      input_word $\leftarrow$ $corpus$[sen_pos$-C+$j], in $\leftarrow$ input_word $\times K$
      target_word $\leftarrow$ center_word, out $\leftarrow$ target_word $\times K$
      temp $\leftarrow$ 0
      if threadIdx.x / 32 == 0 **then**
        f $\leftarrow$ W_{in}[threadIdx.x+in] $\times$ W_{out}[threadIdx.x+out]
        f2 $\leftarrow$ W_{in}[threadIdx.x+in+32] $\times$ W_{out}[threadIdx.x+out+32]
        f3 $\leftarrow$ W_{in}[threadIdx.x+in+64] $\times$ W_{out}[threadIdx.x+out+64]
        f4 $\leftarrow$ W_{in}[threadIdx.x+in+96] $\times$ W_{out}[threadIdx.x+out+96]
        f $\leftarrow$ f + f2 + f3 + f4
        f $\leftarrow$ f + __shfl_down(f, 16)
        f $\leftarrow$ f + __shfl_down(f, 8)
        f $\leftarrow$ f + __shfl_down(f, 4)
        f $\leftarrow$ f + __shfl_down(f, 2)
        f $\leftarrow$ f + __shfl_down(f, 1)
      end if
      if threadIdx.x == 0 **then**
        shared_vector[0] $\leftarrow$ (label $-$ $sigmoid$(f)) $\times$ α
      end if
      __syncthreads()
      temp $\leftarrow$ temp + shared_vector[0] $\times$ W_{out}[threadIdx.x+out]
      atomicAdd(W_{out}[threadIdx.x+out],
      shared_vector[0] $\times$ W_{in}[threadIdx.x+in])
      atomicAdd(W_{in}[threadIdx.x+in], temp)
    end if
  end for
end for

Algorithm 26 Repulsion() device function on **PAR-Word2Vec-gpu**

label ← 0, num_batch ← $(L + B - 1) / B$
for batch_id = 0 **to** num_batch − 1 **do**
  min_pos ← start_idx + (batch_id × B), max_pos ← min(min_pos + B, end_idx)
  in_size ← max_pos - min_pos
  if threadIdx.x == 0 **then**
    if (min_pos − start_idx <= $C - 1$) || ($L - 1$ + start_idx − min_pos <= $C - 1$) **then**
      $C_{rep} \leftarrow curand\_uniform() \times C$
    else
      $C_{rep} \leftarrow curand\_uniform() \times (2 \times C)$
    end if
    $shared\_C_{rand}$[blockIdx.x] ← C_{rep}
  end if
  __syncthreads()
  out_size ← $shared\_C_{rand}$[blockIdx.x] × T
  for i = threadIdx.x **to** out_size − 1 **step** blockDim.x **do**
    $shared\_ns$[blockIdx.x×O+i] ← $curand\_uniform()$ × V
  end for
  __syncthreads()
  for sen_pos = min_pos **to** max_pos − 1 **do**
    $M_{in}[blockIdx.x \times B \times K + ((sen\_pos - min\_pos) \times K) + threadIdx.x]$ ← $W_{in}[corpus$[sen_pos]×K+threadIdx.x]
  end for
  for ns = 0 **to** out_size − 1 **do**
    M_{out}[blockIdx.x×$O \times K$+ns×K+threadIdx.x] ← $W_{out}[shared\_ns$[blockIdx.x×O+ns] ×K+threadIdx.x]
  end for
  __syncthreads()
  // 2D tiled matrix multiplication of M_{out} and M_{in}^{T}
  num_RBA ← (out_size + 16 − 1) / 16, num_CBA ← K / 32
  num_CBB ← (in_size + 16 −1) / 16
  thread_row ← threadIdx.x / 16, thread_col ← threadIdx.x % 16
  reg_tiles[8]
  for rb = 0 **to** num_RBA − 1 **do**
    for cb = 0 **to** num_CBB − 1 **do**
      memset(reg_tiles, 0)
      for m = 0 **to** num_CBA − 1 **do**
        memcpy(shared_vector[0:16×32-1], corresponding each block of M_{out})
        memcpy(shared_vector[16×32:1023], corresponding each block of M_{in})
        __syncthreads()
        if thread_col < 16 **then**
          for j = 0 **to** 32 − 1 **do**
            element_2nd ← shared_vector[16×32+thread_col×32+j]
            reg_tiles[0] ← reg_tiles[0] + shared_vector[thread_row×32+j] × element_2nd
            reg_tiles[1] ← reg_tiles[1] + shared_vector[(thread_row+8)×32+j] × element_2nd
          end for
        end if
        __syncthreads()
      end for
      if thread_col < 16 **then**
        if thread_row < 16 **then**
          $M_{grad}[blockIdx.x \times O \times B + (rb \times 16 + thread\_row) \times B + (cb \times 16) + thread\_col]$ ← reg_tiles[0]
        end if
        if thread_row + 8 < 16 **then**
          $M_{grad}[blockIdx.x \times O \times B + (rb \times 16 + thread\_row + 8) \times B + (cb \times 16) + thread\_col]$ ← reg_tiles[1]
        end if
      end if
    end for
  end for
  __syncthreads()
  for z = threadIdx.x **to** out_size × B − 1 **step** blockDim.x **do**
    f ← M_{grad}[blockIdx.x×$O \times B$+z]
    grad ← (label - $sigmoid$(f)) × α
    M_{grad}[blockIdx.x×$O \times B$+z] ← grad
  end for
  __syncthreads()
  // 2D tiled matrix multiplication of M_{grad} and M_{in}
  M_{out_update} ← **MatrixMultiplication**(M_{grad}, M_{in}) **like lines 27-56**
  __syncthreads()
  // 2D tiled matrix multiplication of M_{grad}^{T} and M_{out}
  M_{in_update} ← **MatrixMultiplication**(M_{grad}^{T}, M_{out}) **like lines 27-56**
  __syncthreads()
  for sen_pos = min_pos **to** max_pos − 1 **do**
    atomicAdd($W_{in}[corpus$[sen_pos]×K+threadIdx.x], $M_{in\_update}[blockIdx.x \times B \times K + ((sen\_pos - min\_pos) \times K) + threadIdx.x]$)
  end for
  __syncthreads()
  for ns = 0 **to** out_size − 1 **do**
    atomicAdd($W_{out}[shared\_ns$[blockIdx.x×O+ns] ×K+threadIdx.x], M_{out_update}[blockIdx.x×$O \times K$+ns×K+threadIdx.x])
  end for
  __syncthreads()
end for

28). At the end of the *Attraction* phase, all threads are synchronized before the start of the *Repulsion* phase (line 9 in Algorithm 24). Similar to the *Repulsion* phase in our parallel CPU implementation, the number of shared negative samples O are chosen by the first word's position in the mini-batch (lines 3-14 in Algorithm 26). After randomly selecting shared negative samples (lines 15-17) and forming the input and output matrices by copying the corresponding word vectors from W_{in} and W_{out} for the current batch (lines 19-24), three matrix-matrix multiplications are performed using a 2D register tiling strategy along with the use of shared-memory. To achieve good performance on GPUs, the prudent use of shared-memory is crucial. Due to the limited amount of shared-memory per SM, we use 1024 size of shared-memory, which provides the best performing occupancy in the current NVIDIA Pascal GPU (line 1 in Algorithm 24). In order take advantage of warp execution, the two sub-matrices involved in each 2D-tiled multiplication are carefully partitioned into allocated shared memory space (e.g., M_{out} and M_{in} matrices involved in the first matrix multiplication are divided into ($O \times K$) / (16 $\times$ 32) and ($B \times K$) / (16 $\times$ 32) tiles, respectively; this allows the first half of the shared-memory space to hold each tile from M_{out} (line 35); the remaining space is used to maintain each tile from M_{in} (line 36). Similarly, M_{grad} and M_{in} are partitioned into ($O \times B$) / (32 $\times$ 16) and ($B \times K$) / (16 $\times$ 32) for performing the second matrix-matrix multiplication, and M_{grad} and M_{out} are divided into ($O \times B$) / (20 $\times$ 8) and (O $\times$ K) / (20 $\times$ 32) to compute the third matrix-matrix multiplication.). The shared-memory and register tile sizes have an impact on both data reuse and concurrency. The higher the tile sizes the higher the data reuse. However, higher tile sizes demand more resources and thus limit the number of concurrently active threads (occupancy). The tile sizes were chosen such that the data movement was minimized while maintaining good concurrency.

4.5.4 Data Movement Analysis for PAR-Word2Vec

$$S(L(Attractions)+\frac{L}{B}(Repulsions))= \\ S(L(1+2(C-C_{rand})(1+8K))+\frac{L}{B}(\frac{6OKB}{\sqrt{\tau}}+O+4KB+4KO+2OB)) \tag{4.6}$$

$$S(\frac{L}{B}(\frac{6OKB}{\sqrt{\tau}}+O+4KB+4KO+2OB)) \tag{4.7}$$

We analyzed the data movement of our PAR-Word2Vec algorithm based on Algorithms 20, 21 and 22. Our algorithm iterates over all S sentences, and *Attraction* and *Repulsion* phases are separately performed within each sentence (line 8-12 in Algorithm 20). The data movement cost of the decoupled *Attraction* phase is similar to the original SG-NS algorithm without a negative sampling loop. The loop in line 3 of Algorithm 21 iterates over all L word tokens in each sentence and has an associated data movement of $1+(2C-2C_{rand})(1+2K+4K+2K)$ for the *Attraction* phase. In the *Repulsion* phase, the loop in line 3 in Algorithm 22 iterates over $\frac{L}{B}$ mini-batches for the current sentence. First the O shared negative samples are randomly chosen (line 11 in Algorithm 22) and kept in the *shared_ns* array through the loop in line 12 (O writes). Then the shared negative word vectors and input word vectors pulled from W_{out} and W_{in} matrices are copied to M_{out} and M_{in} temporary matrices in lines 15 and 16 ($KB+KO$ reads and $KB+KO$ writes). Given M_{out} and M_{in} matrices, three matrix multiplications are required to perform *Repulsion* updates. It is well known that the highest order term in the number of data elements moved (between main memory and a cache of size τ words) for efficient tiled matrix multiplication of two matrices A, $(M \times K)$ and B $(K \times N)$ is $\frac{2MNK}{\sqrt{\tau}}$ (An extensive discussion of both lower bounds and data movement volume for several tiling schemes may be found in the recent work of Smith [87]). Hence, the data movement costs associated with the three matrix multiplications in lines 18, 28 and 30 are $\frac{2OKB}{\sqrt{\tau}}$, $\frac{2OBK}{\sqrt{\tau}}$ and $\frac{2BOK}{\sqrt{\tau}}$, respectively. The loop in lines 19-26 computes the gradients and

has an associated data movement cost of OB reads and OB writes. At the end of *Repulsion* phase (lines 31-42), the update vectors in M_{out} and M_{in} matrices are copied back to W_{out} and W_{in} ($KB + KO$ reads and $KB + KO$ writes). In total, Equation 4.6 shows the data movement cost for our PAR-Word2Vec algorithm, where τ is the cache size. Also, the data movement required for only *Repulsion* phase of PAR-Word2Vec is also shown in Equation 4.7.

4.5.5 Data Movement Analysis Comparison

For the One Billion Word Benchmark dataset (S=30,607,741 and L=$\frac{N}{S}$=$\frac{804,269,958}{30,607,741}$) with K=128, C=8, T=5, C_{rand}=0, C_{rep}=16, O=TC_{rep}=80, and B=24 on a machine with 35 MB cache[10], based on Equation 4.4, the total data movement cost of original SG-NS based Word2Vec algorithm is 85665204×10^6 bytes. Whereas, based on Equation 4.6, the total data movement cost for our PAR-Word2Vec algorithm is only 15109321×10^6 bytes which is approximately 5.67$\times$ lower than the original SG-NS algorithm. Moreover, based on the Equation 4.5 and 4.7, the data movement associated with only *Repulsion* updates is greatly reduced by $(1-\frac{19239278\times10^5}{72487241\times10^6}) \approx 97.3\%$. On the other hand, both algorithms must have the same amount of data movements for *Attraction* updates. The data movement improvement of our approach can be clearly proven by the fact that the difference between Equation 4.4 and 4.5 (*Attraction* updates for original SG-NS based Word2Vec), and the difference between Equation 4.6 and 4.7 (*Attraction* updates for PAR-Word2Vec) are exactly matched.

[10] C_{rand}=0 makes the maximum size of context window at current center word according to the lines 8-9 in Algorithm 19. C_{rep}=16 corresponds to the maximum number of C_{rep}, when C=8 according to the lines 6-10 in Algorithm 22. B=24 is chosen to be used for all the experiments in Section 6.

4.6 Experimental Evaluation

This section provides both performance and quality assessments for the Word2Vec and Node2Vec algorithms. Our PAR-Word2Vec implementations on multi-core CPUs and GPUs are compared with various state-of-the-art implementations.

4.6.1 Benchmarking Machines

The detailed configuration of the benchmarking machines used for experiments is shown in Table 4.2. All the CPU experiments were run on an Intel Xeon CPU E5-2680 v4 running at 2.4 GHz with 128GB RAM. The GPU experiments were run on an NVIDIA Tesla P100 PCIE GPU with 16GB global memory.

Table 4.2: Machine configuration

Machine	Details
CPU	Intel(R) Xeon(R) CPU E5-2680 v4 (28 cores), 128GB; ICC 18.0.3
GPU	Tesla P100 PCIE; CUDA 9.2.88 (56 SMs, 64 cores/MP, 16GB Global Memory, 4 MB L2 cache)

4.6.2 Datasets

4.6.2.1 Text Datasets for Word2Vec Evaluations

For the direct Word2Vec evaluations, we used two publicly available real-world text datasets – text8[11] and One Billion Word Benchmark (1B-Word)[12].

- text8: The dataset contains approximately 17 million word tokens collected from Wikipedia.

[11] http://mattmahoney.net/dc/text8.zip

[12] http://www.statmt.org/lm-benchmark/

- One Billion Word Benchmark (1B-Word): This corpus released by Chelba et al. [12] contains approximately 0.8 billion word tokens produced from the WMT 2011 News Crawl data.

4.6.2.2 Graph Datasets for Node2Vec Evaluations

In order to evaluate the Node2Vec algorithm, we also used five publicly available graph datasets: three labeled datasets – BlogCatalog[13], PPI[14] and Wikipedia-2006[15] – and two unlabeled datasets – Facebook[16] and arXiv ASTRO-PH (ASTRO-PH)[17].

- BlogCatalog: The dataset released by Zafarani and Liu [105] includes the friendship network and group membership information crawled from the BlogCatalog website[18]. Each node represents a blogger, and the nodes connected to each edge denote that the bloggers are friends with each other.

- Protein-Protein Interactions (PPI): This dataset is a subgraph of the PPI network for Homo Sapiens [8]. The preprocessed subgraph and labeled nodes are available from the Node2Vec repository[19].

- Wikipedia-2006: The dataset contains the first 10^9 bytes of the English Wikipedia dump on March 3, 2006 [57]. Each node represents a word, and the co-occurred words are connected through edge. This preprocessed dataset with the nodes labeled with

[13]http://socialcomputing.asu.edu/datasets/BlogCatalog3

[14]https://downloads.thebiogrid.org/BioGRID

[15]http://www.mattmahoney.net/dc/textdata

[16]http://snap.stanford.edu/data/egonets-Facebook.html

[17]http://snap.stanford.edu/data/ca-AstroPh.html

[18]http://www.blogcatalog.com

[19]https://snap.stanford.edu/node2vec/#datasets

the Stanford Log-linear Part-Of-Speech Tagger [93] are available from the Node2Vec repository[19].

- Facebook: This network includes the friendship relation between Facebook users [46]. Each node denotes a user, and two users are connected with an edge if they are friends with each other.

- arXiv ASTRO-PH (ASTRO-PH): This dataset is a collaboration network of Astro Physics related papers in the e-print arXiv between January 1993 to April 2003 [46]. Each node represents an author, and any co-authors of the same paper are linked by an edge.

Table 4.3 and 4.4 show the characteristics of each text and graph dataset and the details of graph datasets, respectively. Given the directed/undirected graph which represents the connections between nodes with/without weights, random walks are pre-generated with the original Node2Vec implementation released by SNAP (Stanford Network Analysis Platform)[20]. Thereafter, the pre-generated random walks are used as inputs for all the Word2Vec variants. Each random walk is comprised of a sequence of $N/S = 81$ nodes.

4.6.3 Evaluation Metrics

4.6.3.1 Word2Vec Evaluation Metrics

We used standard word similarity and relatedness evaluations [5, 82] to compare word embeddings learned from PAR-Word2Vec and other methods. This task involves inventories of word pairs that have been assigned a similarity or relatedness score by human annotators (e.g., (tiger, cat, 7.35), (stock, life, 0.92) [80]). For each word pair, the cosine similarity of the corresponding embeddings is calculated. These cosine similarities are then rank-ordered,

[20]https://github.com/snap-stanford/snap/tree/master/examples/node2vec

Table 4.3: Statistics of text datasets, and graph datasets pre-generated by Node2Vec. V is the number of unique words/the number of unique nodes, S is the total number of sentences over the corpus/the total number of random walks over the graph, and N is the total number of word tokens over the corpus/the sum of the length of the walks in S.

Dataset		V	S	N
Text Dataset	text8	71,291	9,385	16,718,843
	1B-Word	555,514	30,607,741	804,269,958
Graph Dataset	BlogCatalog	10,313	103,120	8,352,720
	PPI	3,891	38,900	3,150,900
	Wikipedia-2006	4,778	47,770	3,869,370
	Facebook	4,040	40,390	3,271,590
	ASTRO-PH	18,773	187,720	15,205,320

Table 4.4: Details of graph datasets. E is the total number of edges and A is the number of different labels for nodes in the graph.

Dataset	**Graph type**	E	A
BlogCatalog	undirected/unweighted	333,983	39
PPI	undirected/unweighted	76,584	50
Wikipedia-2006	undirected/weighted	184,812	40
Facebook	undirected/unweighted	88,234	N/A
ASTRO-PH	directed/unweighted	198,110	N/A

and the rankings compared to rank-ordering of the human judgments using Pearson's ρ (-1 to 1, higher is better). We evaluated on the following datasets:

– **WordSim-353:** 353 pairs rated for similarity of meaning [34].

– **SimLex-999:** 999 pairs rated specifically for similarity, and not relatedness [34].

These evaluations are referred to as "intrinsic", since they do not use any learned parameters beyond the word embeddings. Analogy completion tasks [59] have often been used as another "intrinsic" evaluation, together with similarity/relatedness. However, these tasks

have well-documented issues that limit their value as an evaluation metric [52, 65, 78]. We therefore follow prior work [13, 79, 100] by augmenting our intrinsic evaluation with "extrinsic" evaluations that measure the quality of word embeddings by plugging them into another machine learning model that learns to use them as features. We evaluated on the following tasks:

– **Relation extraction:** SemEval-2010 shared task 8 [32], using a CNN model with word and distance embeddings [107] for nine-way relation classification.

– **Sentiment analysis:** positive/negative binary classification of IMDB movie reviews [56], using a single 100-dimensional LSTM.

4.6.3.2 Node2Vec Evaluation Metrics

The quality of the trained node embeddings out of Word2Vec can be evaluated through such applications as multi-label classification (e.g., classifying bloggers into categories in BlogCatalog dataset) and link prediction tasks. [26,69,91,92]. The node embeddings coming out of the various Word2Vec implementations can just be fed into the same classification models to evaluate them.

– **Multi-label classification**: For the evaluations with labeled graph datasets, the logistic regression model takes the node embeddings as input and predicts the probability of each label based on a logistic function. We used the same train/test split across all datasets. The train set consists of 90% of labeled node embeddings and the remaining 10% is used as the test set. Additionally, the hyperparameters of logistic regression model were tuned using GridSearchCV[21], which seeks for the best parameters of logistic regression through Grid-Search [27].

[21] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

– **Link prediction**: We conducted link prediction task with unlabeled graph datasets based on the edge information. Link prediction can be considered as a binary classification by predicting the connection between two nodes. Where two nodes are connected in the graph, they were labeled as a positive example. Negative examples were generated by randomly sampling pairs of nodes not connected to each other. For a fair evaluation, the numbers of positive and negative examples used in our experiments were balanced and we used the same negative samples across all variants of Word2Vec. Features were generated for each sample by concatenating the learned embeddings for each pair of nodes; these features were then plugged into Support-Vector Machine (SVM) model [16] for training and testing.

4.6.4 Word2Vec Implementations Compared

We evaluated PAR-Word2Vec on multi-core CPUs and GPUs with state-of-the art parallel Word2Vec implementations. The seven implementations used in our comparisons are as follows:

- **Word2Vec-cpu**[22]: The original SG-NS based parallel CPU implementation by Mikolov et al. [60]
- **pWord2Vec-cpu**[23]: SG-NS based parallel CPU implementation by Ji et al. [35]
- **wombatSGNS-cpu**[24]: SG-NS based parallel CPU implementation by Simonton and Alaghband [86]

[22]https://code.google.com/archive/p/word2vec/

[23]https://github.com/IntelLabs/pWord2Vec

[24]https://github.com/tmsimont/wombat

- **pSGNScc-cpu**[25]: SG-NS based parallel CPU implementation by Rengasamy et al. [76]
- **PAR-Word2Vec-cpu**: Our SG-NS based parallel CPU implementation
- **accSGNS-gpu**[26]: SG-NS based parallel GPU implementation by Bae and Yi [4]
- **PAR-Word2Vec-gpu**: Our SG-NS based parallel GPU implementation

Note that, all compared models are based on SG-NS algorithm. While Word2Vec-cpu uses Pthreads API, all other CPU implementations, including pWord2Vec-cpu, wombatSGNS-cpu, pSGNScc-cpu, and our PAR-Word2Vec-cpu, use OpenMP API and the same Intel's Math Kernel Library (MKL) for all BLAS (Basic Linear Algebra Subprograms) operations. It is obvious that annealing the learning rate is a critical part of having good quality of embeddings. For all CPU implementations, during training the model for each epoch, the corpus is read sentence-by-sentence and the learning rate is reduced based on this reading progress through the corpus. Whereas, our PAR-Word2Vec-gpu completes the reading of the entire corpus and copies it into GPU memory once prior to the start of training and gradually decreases the learning rate at the end of each epoch. As shown in line 9 in Algorithm 23, the learning rate is decreased by $\alpha = \alpha - \frac{\text{initial } \alpha}{\text{total number of epochs}}$ at the end of each epoch.

4.6.5 Performance Evaluation

The hyper-parameters used in all experiments are provided in Table 4.5. For all datasets, including text and graph, we used the same number of negative samples $T = 5$ and same size of embedding vector $K = 128$. For text and graph datasets, respectively, we set the window size C to 8 and 10. We then trained all variants of Word2Vec over 10 epochs with all datasets. To ensure fairness, 28 threads were used in all CPU experiments. In addition,

[25]https://github.com/vasupsu/pWord2Vec

[26]https://github.com/kinchi22/word2vec-acc-cuda

Table 4.5: Details of hyper-parameters used in the experiments. I is the total number of training epochs, T is the number of negative samples, C is the window size, K is the embedding vector size as well as p and q are (return and in-out) parameters for pre-generating graph datasets.

Dataset		I	T	C	K	p	q
Text Datasets	text8	10	5	8	128	N/A	N/A
	1B-Word	10	5	8	128	N/A	N/A
Graph Dataset	BlogCatalog	10	5	10	128	0.25	0.25
	PPI	10	5	10	128	4	1
	Wikipedia-2006	10	5	10	128	4	0.5
	Facebook	10	5	10	128	1	1
	ASTRO-PH	10	5	10	128	1	1

all the parameters in all CPU and GPU implementations were tuned for each dataset and the best performing configurations were selected. For both PAR-Word2Vec-cpu and PAR-Word2Vec-gpu, we used $B = 24$ for all the experiments. However, the number of thread blocks were varied according to the total number of sentences in each dataset; $\gamma = 64$ for text8, BlogCatalog, PPI, Wikipedia-2006, Facebook and ASTRO-PH, and $\gamma = 1024$ for 1B-Word datasets. As suggested in the running scripts of pWord2Vec-cpu and pSGNScc-cpu, the batch size of both pWord2Vec-cpu and pSGNScc-cpu were set to $2 \times C + 1$. In the process of pre-generating graph datasets, we used the same p and q values for BlogCatalog, PPI and Wikipedia datasets as suggested in [26]. Note that the window size C is randomly selected for each inner loop in Word2Vec algorithm (e.g., Line 8 in Algorithm 19). Furthermore, all negative target words are randomly chosen from the vocabulary (e.g., Line 17 in Algorithm 19). Therefore, all the experiment results presented in this section are averaged over 5 different executions.

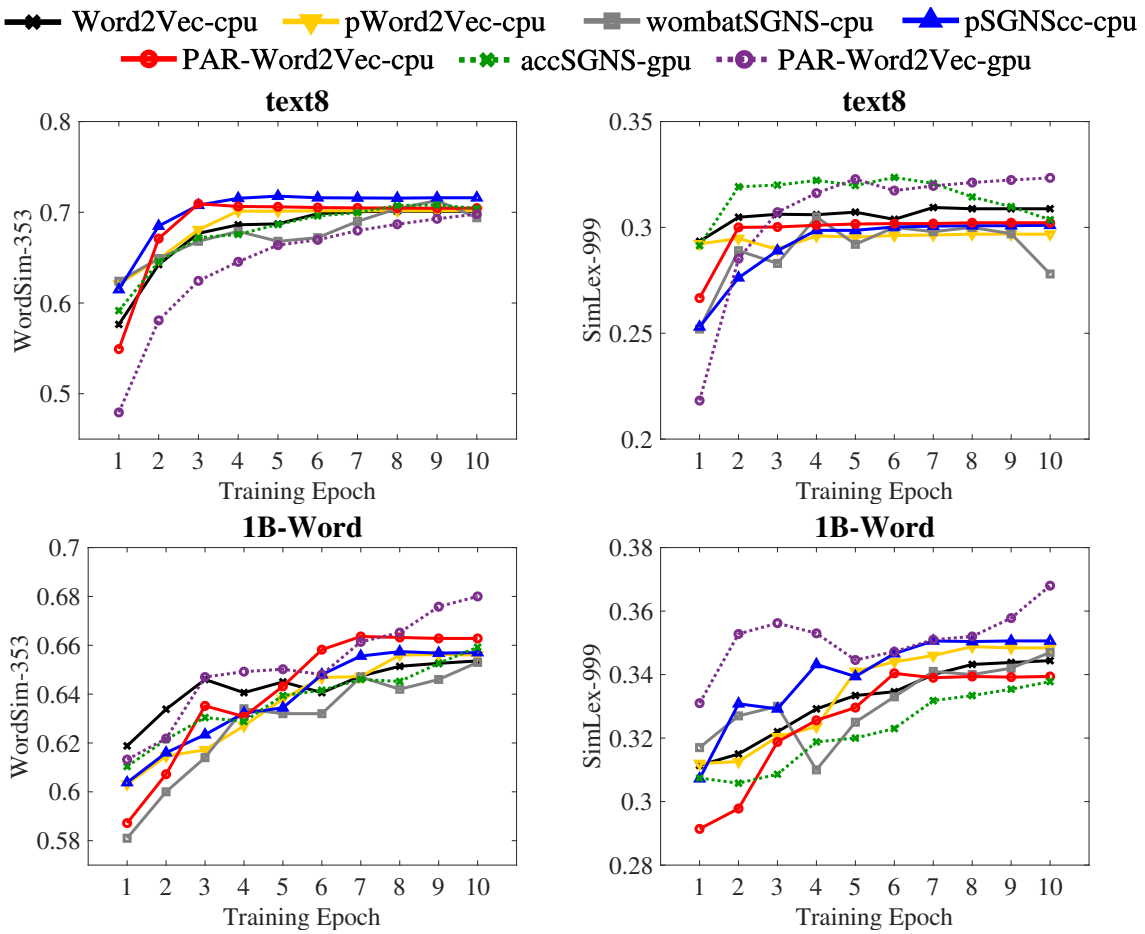


Figure 4.6: Comparison of word similarity scores over training epoch on text datasets, $K = 128$. Each point is averaged over five executions. X-axis: number of training epochs; Y-axis: word similarity scores.

4.6.5.1 Model Quality

Intrinsic Evaluations of Word Embeddings. Figure 4.6 shows word similarity scores over training epochs, comparing all variants of Word2Vec implementations on text datasets. As shown in Table 4.6, the difference between our PAR-Word2Vec models and the baselines is not statistically significant in terms of the converged word similarity scores after training for 10 epochs.

Table 4.6: Mean and standard deviation of converged word similarity scores over 5 different executions on text datasets.

Model	text8		1B-Word	
	WordSim-353	SimLex-999	WordSim-353	SimLex-999
Word2Vec-cpu	0.701 ($\pm$0.010)	0.308 ($\pm$0.014)	0.653 ($\pm$0.004)	0.344 ($\pm$0.007)
pWord2Vec-cpu	0.701 ($\pm$0.008)	0.296 ($\pm$0.008)	0.656 ($\pm$0.003)	0.348 ($\pm$0.002)
wombatSGNS-cpu	0.694 ($\pm$0.013)	0.278 ($\pm$0.009)	0.653 ($\pm$0.001)	0.350 ($\pm$0.002)
pSGNScc-cpu	0.716 ($\pm$0.008)	0.301 ($\pm$0.009)	0.657 ($\pm$0.003)	0.350 ($\pm$0.001)
PAR-Word2Vec-cpu	0.705 ($\pm$0.008)	0.302 ($\pm$0.003)	0.663 ($\pm$0.002)	0.341 ($\pm$0.003)
accSGNS-gpu	0.704 ($\pm$0.004)	0.303 ($\pm$0.002)	0.659 ($\pm$0.003)	0.337 ($\pm$0.002)
PAR-Word2Vec-gpu	0.698 ($\pm$0.008)	0.323 ($\pm$0.005)	0.680 ($\pm$0.004)	0.368 ($\pm$0.004)

For the large 1B-Word dataset, our PAR-Word2Vec-cpu and PAR-Word2Vec-gpu consistently produced high word similarity scores on WordSim-353 and SimLex-999 tasks. This result demonstrate that our sentence-wise decoupled Attraction-Repulsion based approach is highly beneficial to performance improvement in terms of both model quality and speedup. In the large 1B-Word dataset, $\frac{27{,}994{,}959}{30{,}607{,}741} \approx 91.46$ % of sentences over the corpus include less than 25 words. The mini-batch size $B = 24$ that we used for the experiment with 1B-Word dataset indicates that sharing the same negative samples for all words within a sentence would not affect the quality of model at all.

On the small text8 dataset, however, it is interesting point to see that PAR-Word2Vec-gpu has some variability; it produces high averaged score for the SimLex-999 similarity task, but low averaged score for the WordSim-353 task as shown in Table 4.6. However, PAR-Word2Vec-gpu has standard deviations that overlap in the ranges (e.g., $0.698 + 0.008 = 0.706$ on WordSim-353 task), implying that PAR-Word2Vec-gpu yields comparable results on all intrinsic evaluations. We suspect that issue is with the number of thread blocks. As seen in Algorithm 23, each thread block is processing multiple sentences and the global W_{in} and W_{out} matrices are updated after each sentence is processed (local updates within

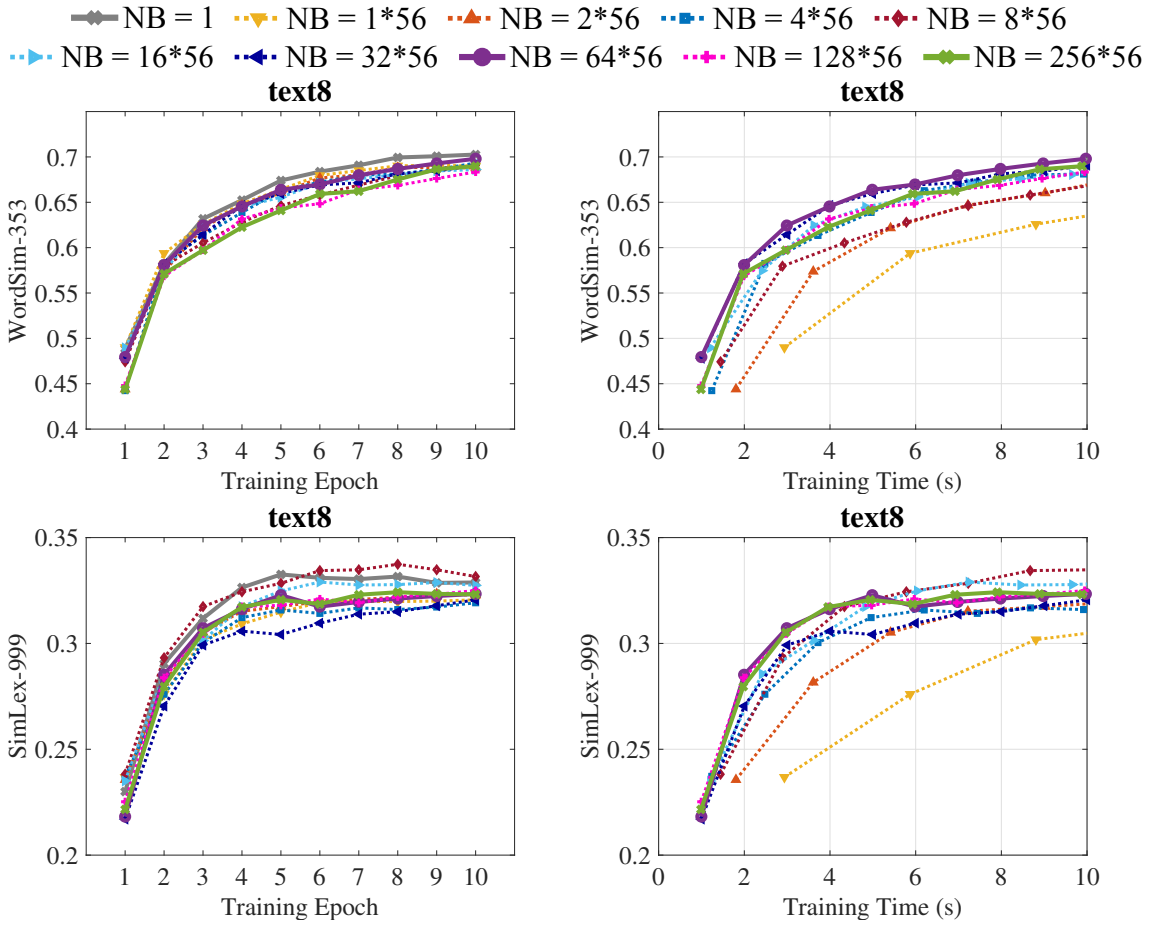


Figure 4.7: Word similarity scores over training epoch and time across different number of thread blocks used in PAR-Word2Vec-gpu on text8 dataset, $K = 128$. Each point is averaged over five executions. NB: the total number of thread blocks; X-axis: number of training epochs and training time in seconds; Y-axis: word similarity scores.

a sentence; global updates across sentences). Because of that, different thread blocks are not able to see each others update until the entire sentences are processed as shown in Algorithm 26. Assume that both sentence 1 and sentence 100 have a common word, word 1. If we processed the sentences sequentially, it is guaranteed that the updates to common word 1 performed while processing sentence 1 is visible before sentence 100 reads the corresponding embedding vector of word 1. Whereas, when we consider sentence 1 was

processed by thread block 1 and sentence 100 by thread block 5 in parallel, then both of the thread blocks will read the same initial embedding vector of word 1. Hence, the updates would not be the same as sequential. The more the number of thread blocks the higher the chance of this incoherence. Especially for the small text8 dataset, this is an issue as average sentence length is around 1000 words (higher the sentence length higher the chance of this incoherence). For the large 1B-Word dataset, the average sentence length is less than 25 (hence this effect may not be visible for 1B-Word dataset). In order to verify this issue, we conducted an experiment with varying the number of thread blocks in text8 dataset. As shown in Figure 4.7, the results were mostly matched as we expected: the smaller number of thread blocks tends to provide a better quality, but slower training time. Another possible reason is that the method of annealing learning rate in PAR-Word2Vec-gpu is different from all other variants (see Section 4.6.4). Accordingly, to achieve high performance in terms of both quality and training time, we had chosen to use 64×56 thread blocks and 1024×56 thread blocks for text8 and 1B-Word datasets, respectively.

Table 4.7: Mean and standard deviation of Macro F_1 scores for relation extraction (Rel. Ext.) task and Accuracy for sentiment analysis (Sent. Analysis) task over 5 different runs each for 5 different embedding executions, by Word2Vec variants.

Model	text8		1B-Word	
	Rel. Ext.	Sent. Analysis	Rel. Ext.	Sent. Analysis
Word2Vec-cpu	0.671 ($\pm$0.010)	0.795 ($\pm$0.006)	0.689 ($\pm$0.009)	0.782 ($\pm$0.008)
pWord2Vec-cpu	0.669 ($\pm$0.006)	0.791 ($\pm$0.004)	0.686 ($\pm$0.008)	0.779 ($\pm$0.007)
wombatSGNS-cpu	0.666 ($\pm$0.007)	0.776 ($\pm$0.005)	0.691 ($\pm$0.010)	0.783 ($\pm$0.005)
pSGNScc-cpu	0.666 ($\pm$0.009)	0.790 ($\pm$0.005)	0.685 ($\pm$0.010)	0.784 ($\pm$0.006)
PAR-Word2Vec-cpu	0.665 ($\pm$0.010)	0.783 ($\pm$0.008)	0.691 ($\pm$0.008)	0.780 ($\pm$0.004)
accSGNS-gpu	0.680 ($\pm$0.010)	0.796 ($\pm$0.007)	0.689 ($\pm$0.006)	0.787 ($\pm$0.006)
PAR-Word2Vec-gpu	0.663 ($\pm$0.010)	0.807 ($\pm$0.004)	0.623 ($\pm$0.009)	0.780 ($\pm$0.004)

Extrinsic Evaluations of Word Embedding. Table 4.7 shows performance on the extrinsic relation extraction and sentiment classification tasks after training for 10 epochs; performance numbers were averaged over five replicates each of five embedding runs, to control for random initialization effects in both embedding learning and the models used for extrinsic evaluations. PAR-Word2Vec-cpu matches the evaluation quality of other CPU implementations, using both text8 and 1B-word datasets. PAR-Word2Vec-gpu yields comparable or superior performance on sentiment classification, but surprisingly low evaluation quality on the relation extraction task with the larger 1B-word dataset; quality with text8 is on par with other implementations. Taken together with the intrinsic evaluation results, this suggests that the massive parallelization of our PAR-Word2Vec-gpu algorithm may be exploring the limits of HOGWILD!-style data parallelization in word embedding training.

Table 4.8: Mean and standard deviation of Micro F_1 and Macro F_1 scores for multi-label classification, and Micro F_1 score for link prediction task over 5 different runs each for 5 different embedding executions, by Word2Vec variants.

Model	BlogCatalog		PPI		Wikipedia-2006		Facebook	ASTRO-PH
	Micro	Macro	Micro	Macro	Micro	Macro	Micro	Micro
Word2Vec-cpu	0.429	0.306	0.213	0.188	0.461	0.081	0.699	0.723
pWord2Vec-cpu	0.422	0.304	0.211	0.187	0.478	0.088	0.691	0.721
wombatSGNS-cpu	0.429	0.310	0.211	0.184	0.464	0.079	0.692	0.718
pSGNScc-cpu	0.422	0.301	0.211	0.184	0.442	0.070	0.686	0.692
PAR-Word2Vec-cpu	0.425	0.304	0.223	0.194	0.480	0.101	0.687	0.723
accSGNS-gpu	0.423	0.297	0.215	0.190	0.460	0.082	0.698	0.721
PAR-Word2Vec-gpu	0.420	0.291	0.217	0.184	0.456	0.071	0.672	0.720
Avg. standard dev.	±0.006	±0.011	±0.006	±0.006	±0.006	±0.008	±0.001	±0.002

Extrinsic Evaluations of Graph Embedding. To evaluate the multi-label classification task with labeled graph datasets, we measured the average of Micro F_1 and Macro F_1 scores through 5-fold cross-validation across multiple shuffles of the datasets. For the link prediction task with unlabeled graph datasets, we only measured Micro F_1 score

since the numbers of distinct positive edges and distinct negative edges used for the SVM training were the identical. As shown in Table 4.8, all variants including our CPU and GPU implementations maintain mostly the same quality of node embeddings.

4.6.5.2 Speedup

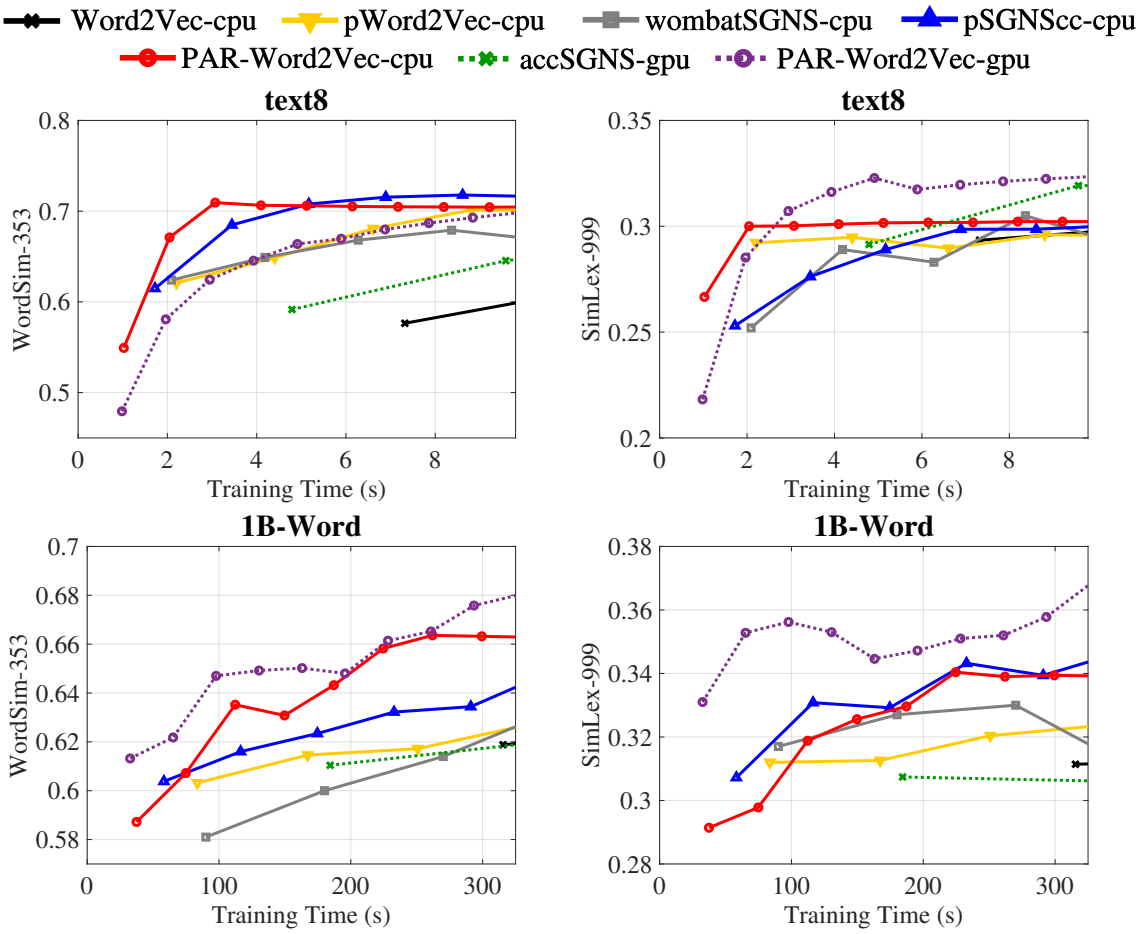


Figure 4.8: Comparison of word similarity scores over training time on text datasets, $K = 128$. Each point is averaged over five executions. X-axis: training time in seconds; Y-axis: word similarity scores.

Figure 4.8 shows the word similarity scores over elapsed training time in text datasets. Our PAR-Word2Vec-cpu and PAR-Word2Vec-gpu achieved significant improvement in performance over existing state-of-the-art parallel Word2Vec implementations while maintaining the same evaluation quality. As the results in Table 4.9 show, PAR-Word2Vec-cpu

Table 4.9: Comparison of the training time in seconds per epoch on text and graph datasets.

Model	Text Dataset		Labeled Graph Dataset			Unlabeled Graph Dataset	
	text8	1B -Word	Blog Catalog	PPI	Wikipedia -2006	Facebook	ASTRO -PH
Word2Vec-cpu	7.32	315.46	6.43	3.13	2.95	2.55	12.54
pWord2Vec-cpu	2.20	86.63	1.56	0.45	0.55	0.53	2.66
wombatSGNS-cpu	2.09	90.04	1.43	0.47	0.58	0.71	2.88
pSGNScc-cpu	1.72	58.20	1.46	0.70	0.75	0.84	2.77
PAR-Word2Vec-cpu	**1.02**	**37.43**	**0.83**	**0.33**	**0.28**	**0.31**	**1.43**
accSGNS-gpu	4.79	185.31	2.23	0.66	0.62	1.37	6.44
PAR-Word2Vec-gpu	**0.98**	**32.60**	**0.72**	**0.20**	**0.21**	**0.27**	**1.08**

achieved approximately 9.01×, 2.41×, 2.51×, and 1.62× speedup on the large 1B-Word dataset compared to Word2Vec-cpu, pWord2Vec-cpu, wombatSGNS-cpu, and pSGNScc-cpu, respectively. For GPU implementations, our parallel PAR-Word2Vec-gpu launches a kernel with only $\gamma \times 56$ thread blocks with K threads and accSGNS-gpu launches a large amount of S thread blocks along with K threads. Although it uses a relatively fewer thread blocks compared to accSGNS-gpu, PAR-Word2Vec-gpu achieved 5.96× speedup over accSGNS-gpu while maintaining same or better quality. With the graph datasets, the range of performance improvements of our CPU and GPU implementations is almost the same as the performance improvement with text datasets. On the large ASTRO-PH graph dataset, PAR-Word2Vec-cpu achieved 8.77×, 1.86×, 2.01×, and 1.93× speedup compared

to Word2Vec-cpu, pWord2Vec-cpu, wombatSGNS-cpu, and pSGNScc-cpu, respectively. Our PAR-Word2Vec-gpu also consistently outperformed accSGNS-gpu.

4.7 Discussion

One of the major factors that limits our single GPU implementation compared to our single CPU implementation is synchronization overheads. Our GPU implementation uses shared-memory to keep a slice of W_{in}, W_{out} and temporary results. This requires multiple synchronizations (one synchronization per load, store and update of each data structure). In contrast, our CPU implementation does not require any synchronization as we are using the implicit cache to buffer data. Another factor that affects the GPU performance is atomic operations. Since the amount of parallelism in GPUs is much higher than CPUs, we had to use atomic updates to maintain consistency of our data structures and this negatively impacted performance. We also found that intra thread-block load imbalance also limited the GPU performance. Techniques like binning can be employed to reduce load imbalance.

4.8 Conclusion

In this chapter, we built a parallel word embedding algorithm to enhance data locality, focusing on reduction of data movement. To achieve high performance, minimizing data movement is a critical factor, since data movement is much more expensive than arithmetic operations. We found the main bottleneck of the original Word2Vec algorithm by conducting a systematic analysis of data movement. The rearrangement of data computation enables our proposed algorithm to greatly reduce the data movement overheads. Experiments on the

large datasets show that our algorithm achieves superior performance over the existing state-of-the-art implementations. We also presented insights into parallelism versus data-locality trade-offs as well as performance versus quality trends.

Chapter 5: Conclusion and Future Work

5.1 Conclusion

In order for current machine learning algorithms to achieve high performance on multi-core CPUs and GPUs platforms, minimizing the amount of data movement is increasingly critical. However, the design and implementation of new algorithms in machine learning have been largely driven by a focus on computational complexity. In this dissertation, we explored an exercise in different ways of implementing data-aware and architecture-aware algorithms for state-of-the-art machine learning algorithms.

For an efficient parallel LDA algorithm on multi-core CPUs, 2D-tiling strategy and over-decomposition technique are used to avoid use of atomic operations and ensure good load balancing. A high-performance LDA algorithm for GPUs is further proposed based on approximated Collapsed Gibbs Sampling. The proposed LDA algorithms are designed to achieve high performance by systematically analyzing the data access patterns for LDA and devising suitable algorithmic adaptations and parallelization strategies for multi-core CPUs and GPUs.

For an efficient parallel Non-negative Matrix Factorization algorithm on multi-core CPUs and GPUs, a HALS-based parallel NMF algorithm is developed by incorporating algorithmic transformations to enhance data locality. A systematic analysis of data movement overheads

associated with NMF algorithm is conducted to determine the bottlenecks. The new parallel NMF algorithm alleviates the data movement overheads by enhancing data locality.

For an efficient parallel Word2Vec algorithm on mutli-core CPUs and GPUs, a data-locality-enhanced Word2Vec algorithm based on Skip-gram with a novel negative sampling method is developed. The main bottleneck of the original Word2Vec algorithm is identified by conducting a systematic analysis of data movement. The rearrangement of data computation is performed to greatly reduce the data movement overheads incurred from a large number of dot-product operations. The utility of the new Word2Vec implementation within the Node2Vec algorithm is also shown for accelerating embedding learning for large graphs. We also presented insights into parallelism versus data-locality trade-offs as well as performance versus quality trends.

Experimental results with large datasets demonstrate the effectiveness of our new approaches.

5.2 Future Research

NMF versus SG-NS based Word2Vec: Levy and Goldberg [47] presented the neural network-based SG-NS word embedding algorithm that can be viewed as implicit factorization of the word co-occurrence matrix. More specifically, they showed the objective of SG-NS is to implicitly factorize a shift Pointwise Mutual Information (PMI) matrix. The problem of PMI matrix is that the zero values represent unobserved word-context pairs, whereas the negative values indicate uncorrelated word-context pairs. In order to avoid this inconsistency problem, an approximation of PMI matrix called the Positive PMI (PPMI) matrix can be adopted with a non-negativity constraint (replacing all negative values to zero). As an alternative to the original Word2Vec algorithm and the basic PMI-based approaches,

they also showed Shift PPMI (SPPMI) and Singular Value Decomposition (SVD) methods to decompose the word co-occurrence matrix. They compared the original SG-NS algorithm with the matrix-based algorithms, SPPMI and SVD in terms of the quality of solutions using word similarity and word analogy tasks. Experimental results showed that SPPMI and SVD yield better results on word similarity task compared to SG-NS algorithm. However, SG-NS produced comparable or better results on word analogy task over SPPMI and SVD algorithms. They suspected that issue is related to the nature of SG-NS's training method in which weighted matrix factorization provides more influence to frequently appearing word-context pairs. By bridging my in-depth knowledge of NMF and SG-NS algorithms, I would like to investigate the quality of word embeddings generated by NMF, and present insights into NMF versus SG-NS trade-offs.

Recently, several graph embedding algorithms based on matrix factorization [10, 73, 74] and neural word embedding techniques [26, 69] have been proposed to learn vector-based network representations of nodes in a graph. Gurukar et al. [27] described various graph embedding algorithms in detail, and extensively evaluated the quality of their embeddings on link prediction and node classification tasks. I would like to utilize our optimized NMF algorithm (described in Chapter 3) to enable acceleration of network representation learning for large graphs.

Parallel Dynamic Word Embeddings: While embeddings are often trained once on a large corpus and re-used for a variety of applications, several results have demonstrated that for tasks in specific domains, such as biomedicine, embeddings trained *de novo* on domain-specific corpora outperform general-purpose embeddings [45, 66]. Additionally, recent work has shown that training many sets of embeddings on specific sub-corpora can be used to model language change over time [29], even for specific tasks like tracking

armed conflicts [43]. The time cost to train a large number of embedding models limits the expanded use of embeddings for detailed analysis of multiple datasets.

Hence, in order to reduce the time required to train high-quality word embeddings, I would like to apply our new Word2Vec training approach (data-first orientation) to neural network-based dynamic word embedding techniques such as ELMo (Embeddings from Language Models) [70] and BERT (Bidirectional Encoder Representations from Transformers) [17] models that the NLP community has moved to. Different from the Word2Vec model, the ELMo/BERT models inherently encode a sequence of words and therefore an entire sentence needs to be captured in sequence. The goal is to enable acceleration of ELMo/BERT models which might be constrained by their excessive training time.

Parallel Non-negative Tensor Factorization: Based on my previous experience with the parallelization of NMF, I would like to further develop an efficient parallel algorithm for Non-negative Tensor Factorization (NTF). NTF is an unsupervised dimension reduction technique widely used in the fields of bioinformatics and image processing [14,83,90]. Given a non-negative 3D tensor $T \in \mathbb{R}_{+}^{X_1 \times X_2 \times X_3}$, a CANDECOMP/PARAFAC (CP) decomposition [11,30] of T can be formulated as follows:

$$T \approx \sum_{k=1}^{K} w_k \otimes s_k \otimes h_k \tag{5.1}$$

where $w_k \in \mathbb{R}_{+}^{X_1}$, $s_k \in \mathbb{R}_{+}^{X_2}$ and $h_k \in \mathbb{R}_{+}^{X_3}$ represent the rank-K approximation of the input tensor T, and $\otimes$ denotes an outer product of vectors. Further, Equation 5.2 can be also written in matrix format as follows:

$$T \approx W, S, H \tag{5.2}$$

Hence, NTF estimates W, S and H matrices, such that $\sum_{k=1}^{K} w_k \otimes s_k \otimes h_k$ approximates T. NTF minimizes the reconstruction error between T and the reconstructed tensor from

W, S, H. The NTF problem known as non-negative CP (NNCP) can be defined as follows:

$$\min_{W,S,H} ||T - W, S, H||_F^2 \quad \text{subject to } W, S, H \geq 0 \tag{5.3}$$

In order to minimize the cost functions, such as Frobenius norm and Kullback-Leibler divergence, several variants of NTF algorithms have been proposed. Since NTF can be viewed as an extension of NMF, the existing three NMF algorithms including *Multiplicative Update (MU)* by Lee et al. [44], *Alternating Least Squares (ALS)* and *Hierarchical Alternating Least Squares (HALS)* by Cichocki et al. [14] can be directly employed to the NTF algorithm.

Prior Efforts on Sequential and Parallel NTF Algorithms

Several studies solve the NNCP problem in Equation 5.3 using MU-based NTF algorithm for many different applications, such as image decomposition and sound source separation [20, 83, 84, 90, 99]. However, some studies have reported that the use of MU algorithm suffers from slower convergence and lower convergence rate [23, 37, 51].

Alternating Least Squares (ALS) approach is a special type of block coordinate Gauss-Seidel method [25] to solve non-linear optimization problems. For the ALS-based NMF algorithm, ALS alternately updates a factor matrix where the other matrix is fixed. Thus, ALS-based NMF algorithm converts the non-convex problem into a set of convex least squares subproblems. The original ALS algorithm could be extended to the NTF problem through updating a subset of matrices where the rest of the matrices are fixed. Therefore, the original ALS-based NTF problem solves a sequence of ALS-based NMF subproblems [21, 109]. In other words, the ALS-based NTF algorithm divides the original problem in

Equation 5.3 into three sequential subproblems as follows:

$$\min_{H} ||T_h - (W \odot S)H||_F^2 \qquad \text{where } W \text{ and } S \text{ are fixed}$$

$$\min_{S} ||T_s - (W \odot H)S||_F^2 \qquad \text{where } W \text{ and } H \text{ are fixed}$$

$$\min_{W} ||T_w - (H \odot S)W||_F^2 \qquad \text{where } H \text{ and } S \text{ are fixed}$$

where $\odot$ is the Khatri-Rao product of the two matrices. In each subproblem, T_h, T_s and T_w represent the unfolded tensors T along the h, s and w dimensions. Recently, Kim et al. [39] propose an efficient Alternating Non-negative Least Squares (ANLS) based NTF algorithm that solves non-negativity constrained least squares (NNLS) problems using Block Principle Pivoting algorithm. Under the Karush-Kuhn-Tucker (KTT) conditions, their algorithm iteratively finds the indices of non-zero elements (passive set) and zero elements (active set) in the optimal matrices until KTT conditions are satisfied. The values of indices that correspond to the active set will become zero, and the values of passive set are approximated by solving a standard least squares problem.

As an alternative to the ALS approach, Cichocki et al. [14] extend their FAST HALS based NMF algorithm to FAST HALS based NTF algorithm using squared euclidean distance. The FAST HALS based NTF algorithm hierarchically updates each column vector of factor matrices at a time and then uses it to update a corresponding row vector of other matrices. The algorithm minimizes a set of local cost functions. As described in Algorithm 3 in [14], if the number of factor matrices equals to 2, then the FAST HALS based NTF corresponds to the FAST HALS based NMF.

Compared to the NMF, a limited amount of effort has been conducted on parallelization of NTF. Zhang et al. [109] propose an efficient data partition scheme for parallel ALS-based NTF algorithm. They considered the subproblem-specific data partitions to minimize memory use, maintaining a large scale global climate dataset. For each ALS subproblem, 3D tensor and three factor matrices are distributed to independent processors and divided into row blocks. Antikainen et al. [2] presents a parallel GPU implementation of the NTF algorithm based on the Block Gauss-Seidel (BGS) method [25] and Jacobi iterative method proposed by Hazan et al. [31].

Bibliography

[1] Mehdi Hosseinzadeh Aghdam, Morteza Analoui, and Peyman Kabiri. A novel non-negative matrix factorization method for recommender systems. *Applied Mathematics & Information Sciences*, 9(5):2721, 2015.

[2] Jukka Antikainen, Jiri Havel, Radovan Josth, Adam Herout, Pavel Zemcik, and Markku Hauta-Kasari. Nonnegative tensor factorization accelerated using gpgpu. *IEEE Transactions on Parallel and Distributed Systems*, 22(7):1135–1141, 2011.

[3] Arthur Asuncion, Max Welling, Padhraic Smyth, and Yee Whye Teh. On smoothing and inference for topic models. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 27–34. AUAI Press, 2009.

[4] Seulki Bae and Youngmin Yi. Acceleration of word2vec using gpus. In *International Conference on Neural Information Processing*, pages 269–279. Springer, 2016.

[5] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 238–247. Association for Computational Linguistics, 2014.

[6] Eric Battenberg and David Wessel. Accelerating non-negative matrix factorization for audio source separation on multi-core and many-core architectures. In *ISMIR*, pages 501–506, 2009.

[7] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *JMLR*, 2003.

[8] Bobby-Joe Breitkreutz, Chris Stark, Teresa Reguly, Lorrie Boucher, Ashton Breitkreutz, Michael Livstone, Rose Oughtred, Daniel H Lackner, Jürg Bähler, Valerie Wood, et al. The biogrid interaction database: 2008 update. *Nucleic acids research*, 36(suppl_1):D637–D640, 2007.

[9] John Canny, Huasha Zhao, Bobby Jaros, Ye Chen, and Jiangchang Mao. Machine learning at the limit. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 233–242. IEEE, 2015.

[10] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, pages 891–900. ACM, 2015.

[11] J Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika*, 35(3):283–319, 1970.

[12] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

[13] Billy Chiu, Anna Korhonen, and Sampo Pyysalo. Intrinsic Evaluation of Word Vectors Fails to Predict Extrinsic Performance. *Proceedings of the 1st Workshop on Evaluating Vector Space Representations for NLP*, pages 1–6, 2016.

[14] Andrzej Cichocki and Anh-Huy Phan. Fast local algorithms for large scale nonnegative matrix and tensor factorizations. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 92(3):708–721, 2009.

[15] Andrzej Cichocki, Rafal Zdunek, and Shun-ichi Amari. Hierarchical als algorithms for nonnegative matrix and 3d tensor factorization. In *International Conference on Independent Component Analysis and Signal Separation*, pages 169–176. Springer, 2007.

[16] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Chao Dong, Huijie Zhao, and Wei Wang. Parallel nonnegative matrix factorization algorithm on the distributed memory platform. *International journal of parallel programming*, 38(2):117–137, 2010.

[19] James P Fairbanks, Ramakrishnan Kannan, Haesun Park, and David A Bader. Behavioral clusters in dynamic graphs. *Parallel Computing*, 47:38–50, 2015.

[20] Cédric Févotte and Alexey Ozerov. Notes on nonnegative tensor factorization of the spectrogram for audio source separation: statistical insights and towards self-clustering of the spatial cues. In *International Symposium on Computer Music Modeling and Retrieval*, pages 102–115. Springer, 2010.

[21] Michael P Friedlander and Kathrin Hatz. Computing non-negative tensor factorizations. *Optimisation Methods and Software*, 23(4):631–647, 2008.

[22] Nicolas Gillis. The why and how of nonnegative matrix factorization. *Regularization, Optimization, Kernels, and Support Vector Machines*, 12(257), 2014.

[23] Edward F Gonzalez and Yin Zhang. Accelerating the lee-seung algorithm for non-negative matrix factorization. *Dept. Comput. & Appl. Math., Rice Univ., Houston, TX, Tech. Rep. TR-05-02*, pages 1–13, 2005.

[24] Thomas L Griffiths and Mark Steyvers. Finding scientific topics. *Proceedings of the National academy of Sciences*, 101(suppl 1):5228–5235, 2004.

[25] Luigi Grippo and Marco Sciandrone. On the convergence of the block nonlinear gauss–seidel method under convex constraints. *Operations research letters*, 26(3):127–136, 2000.

[26] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[27] Saket Gurukar, Priyesh Vijayan, Aakash Srinivasan, Goonmeet Bajaj, Chen Cai, Moniba Keymanesh, Saravana Kumar, Pranav Maneriker, Anasua Mitra, Vedang Patel, et al. Network representation learning: Consolidation and renewed bearing. *arXiv preprint arXiv:1905.00987*, 2019.

[28] Maryam Habibi, Leon Weber, Mariana Neves, David Luis Wiegandt, and Ulf Leser. Deep learning with word embeddings improves biomedical named entity recognition. *Bioinformatics*, 33(14):i37–i48, 2017.

[29] William L. Hamilton, Jure Leskovec, and Dan Jurafsky. Diachronic Word Embeddings Reveal Statistical Laws of Semantic Change. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1489–1501, Berlin, Germany, aug 2016. Association for Computational Linguistics.

[30] Richard A Harshman. Foundations of the parafac procedure: Models and conditions for an" explanatory" multimodal factor analysis. 1970.

[31] Tamir Hazan, Simon Polak, and Amnon Shashua. Sparse image coding using a 3d non-negative tensor factorization. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 1, pages 50–57. IEEE, 2005.

[32] Iris Hendrickx, Su Nam Kim, Zornitsa Kozareva, Preslav Nakov, Diarmuid Ó Séaghdha, Sebastian Padó, Marco Pennacchiotti, Lorenza Romano, and Stan Szpakowicz. SemEval-2010 Task 8: Multi-Way Classification of Semantic Relations between Pairs of Nominals. In *Proceedings of the 5th International Workshop on Semantic Evaluation*, pages 33–38. Association for Computational Linguistics, 2010.

[33] Antonio Hernando, Jesús Bobadilla, and Fernando Ortega. A non negative matrix factorization for collaborative filtering recommender systems based on a bayesian probabilistic model. *Knowledge-Based Systems*, 97:188–202, 2016.

[34] Felix Hill, Roi Reichart, and Anna Korhonen. Simlex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, 41(4):665–695, 2015.

[35] Shihao Ji, Nadathur Satish, Sheng Li, and Pradeep Dubey. Parallelizing word2vec in shared and distributed memory. *arXiv preprint arXiv:1604.04661*, 2016.

[36] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. A high-performance parallel algorithm for nonnegative matrix factorization. In *ACM SIGPLAN Notices*, volume 51, page 9. ACM, 2016.

[37] Hyunsoo Kim and Haesun Park. Nonnegative matrix factorization based on alternating nonnegativity constrained least squares and active set method. *SIAM journal on matrix analysis and applications*, 30(2):713–730, 2008.

[38] Jingu Kim and Haesun Park. Fast nonnegative matrix factorization: An active-set-like method and comparisons. *SIAM Journal on Scientific Computing*, 33(6):3261–3281, 2011.

[39] Jingu Kim and Haesun Park. Fast nonnegative tensor factorization with an active-set-like method. In *High-Performance Scientific Computing*, pages 311–326. Springer, 2012.

[40] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Stanford, CA, 1995.

[41] Sven Koitka and Christoph M Friedrich. nmfgpu4r: Gpu-accelerated computation of the non-negative matrix factorization (nmf) using cuda capable hardware. *R JOURNAL*, 8(2):382–392, 2016.

[42] Da Kuang, Jaegul Choo, and Haesun Park. Nonnegative matrix factorization for interactive topic modeling and document clustering. In *Partitional Clustering Algorithms*, pages 215–243. Springer, 2015.

[43] Andrey Kutuzov, Erik Velldal, and Lilja Øvrelid. Tracing armed conflicts with diachronic word embedding models. In *Proceedings of the Events and Stories in the News Workshop*, pages 31–36, Vancouver, Canada, aug 2017. Association for Computational Linguistics.

[44] Daniel D Lee and H Sebastian Seung. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562, 2001.

[45] K Lee, S A Hasan, O Farri, A Choudhary, and A Agrawal. Medical Concept Normalization for Online User-Generated Texts. In *2017 IEEE International Conference on Healthcare Informatics (ICHI)*, pages 462–469, aug 2017.

[46] Jure Leskovec and Andrej Krevl. {SNAP Datasets}:{Stanford} large network dataset collection. 2015.

[47] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2177–2185. Curran Associates, Inc., 2014.

[48] Omer Levy, Yoav Goldberg, and Ido Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225, 2015.

[49] Ruiqi Liao, Yifan Zhang, Jihong Guan, and Shuigeng Zhou. Cloudnmf: a mapreduce implementation of nonnegative matrix factorization for large-scale biological datasets. *Genomics, proteomics & bioinformatics*, 12(1):48–51, 2014.

[50] M. Lichman. UCI machine learning repository, 2013.

[51] Chih-Jen Lin. Projected gradient methods for nonnegative matrix factorization. *Neural computation*, 19(10):2756–2779, 2007.

[52] Tal Linzen. Issues in evaluating semantic spaces using word analogies. In *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, pages 13–18, 2016.

[53] Chao Liu, Hung-chih Yang, Jinliang Fan, Li-Wei He, and Yi-Min Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proceedings of the 19th international conference on World wide web*, pages 681–690. ACM, 2010.

[54] Noel Lopes and Bernardete Ribeiro. Non-negative matrix factorization implementation using graphic processing units. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 275–283. Springer, 2010.

[55] Mian Lu, Ge Bai, Qiong Luo, Jie Tang, and Jiuxin Zhao. Accelerating topic model training on a single machine. In *APWeb*. Springer, 2013.

[56] Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150. Association for Computational Linguistics, 2011.

[57] Matt Mahoney. Large text compression benchmark. *URL: http://www. mattmahoney. net/text/text. html*, 2011.

[58] Edgardo Mejía-Roa, Daniel Tabas-Madrid, Javier Setoain, Carlos García, Francisco Tirado, and Alberto Pascual-Montano. Nmf-mgpu: non-negative matrix factorization on multi-gpu systems. *BMC bioinformatics*, 16(1):43, 2015.

[59] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[60] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[61] SPFGH Moen and Tapio Salakoski2 Sophia Ananiadou. Distributional semantics resources for biomedical text processing. *Proceedings of LBM*, pages 39–44, 2013.

[62] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer, 2005.

[63] Preslav Nakov, Alan Ritter, Sara Rosenthal, Fabrizio Sebastiani, and Veselin Stoyanov. Semeval-2016 task 4: Sentiment analysis in twitter. In *Proceedings of the 10th international workshop on semantic evaluation (semeval-2016)*, pages 1–18, 2016.

[64] David Newman, Arthur Asuncion, Padhraic Smyth, and Max Welling. Distributed algorithms for topic models. *JMLR*, 2009.

[65] Denis Newman-Griffis, Albert M Lai, and Eric Fosler-Lussier. Insights into Analogy Completion from the Biomedical Domain. In *BioNLP 2017*, pages 19–28, Vancouver, Canada, aug 2017. Association for Computational Linguistics.

[66] Denis Newman-Griffis and Ayah Zirikly. Embedding Transfer for Low-Resource Medical Named Entity Recognition: A Case Study on Patient Mobility. In *Proceedings of the BioNLP 2018 workshop*, pages 1–11, Melbourne, Australia, jul 2018. Association for Computational Linguistics.

[67] Thien Huu Nguyen and Ralph Grishman. Relation extraction: Perspective from convolutional neural networks. In *Proceedings of the 1st Workshop on Vector Space Modeling for Natural Language Processing*, pages 39–48, 2015.

[68] Erik Ordentlich, Lee Yang, Andy Feng, Peter Cnudde, Mihajlo Grbovic, Nemanja Djuric, Vladan Radosavljevic, and Gavin Owens. Network-efficient distributed word2vec training system for large vocabularies. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1139–1148. ACM, 2016.

[69] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[70] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[71] Xuan-Hieu Phan and Cam-Tu Nguyen. Gibbslda++: A c/c++ implementation of latent dirichlet allocation (lda), 2007.

[72] Ian Porteous, David Newman, Alexander Ihler, Arthur Asuncion, Padhraic Smyth, and Max Welling. Fast collapsed gibbs sampling for latent dirichlet allocation. In *SIGKDD*. ACM, 2008.

[73] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Chi Wang, Kuansan Wang, and Jie Tang. Netsmf: Large-scale network embedding as sparse matrix factorization. In *The World Wide Web Conference*, pages 1509–1520. ACM, 2019.

[74] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 459–467. ACM, 2018.

[75] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[76] Vasudevan Rengasamy, Tao-Yang Fu, Wang-Chien Lee, and Kamesh Madduri. Optimizing word2vec performance on multicore systems. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, page 3. ACM, 2017.

[77] Stefan A Robila and Lukasz G Maciak. A parallel unmixing algorithm for hyperspectral images. In *Intelligent Robots and Computer Vision XXIV: Algorithms, Techniques, and Active Vision*, volume 6384, page 63840F. International Society for Optics and Photonics, 2006.

[78] Anna Rogers, Aleksandr Drozd, and Bofang Li. The (too Many) Problems of Analogical Reasoning with Word Vectors. *Proceedings of the 6th Joint Conference on Lexical and Computational Semantics (*SEM 2017)*, pages 135–148, 2017.

[79] Anna Rogers, Shashwath Hosur Ananthakrishna, and Anna Rumshisky. What's in Your Embedding, And How It Predicts Task Performance. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 2690–2703, Santa Fe, NM, USA, 2018. Association for Computational Linguistics.

[80] Herbert Rubenstein and John B. Goodenough. Contextual correlates of synonymy. *Communications of the ACM*, 8(10):627–633, 1965.

[81] Magnus Sahlgren and Alessandro Lenci. The effects of data size and frequency range on distributional semantic models. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 975–980. Association for Computational Linguistics, 2016.

[82] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 298–307. Association for Computational Linguistics, 2015.

[83] Amnon Shashua and Tamir Hazan. Non-negative tensor factorization with applications to statistics and computer vision. In *Proceedings of the 22nd international conference on Machine learning*, pages 792–799. ACM, 2005.

[84] Amnon Shashua, Ron Zass, and Tamir Hazan. Multi-way clustering using super-symmetric non-negative tensor factorization. In *European conference on computer vision*, pages 595–608. Springer, 2006.

[85] Tian Shi, Kyeongpil Kang, Jaegul Choo, and Chandan K Reddy. Short-text topic modeling via non-negative matrix factorization enriched with local word-context correlations. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1105–1114. International World Wide Web Conferences Steering Committee, 2018.

[86] Trevor M Simonton and Gita Alaghband. Efficient and accurate word2vec implementations in gpu and shared-memory multicore architectures. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

[87] Tyler Michael Smith et al. *Theory and practice of classical matrix-matrix multiplication for hierarchical memory architectures*. PhD thesis, 2018.

[88] Padhraic Smyth, Max Welling, and Arthur U Asuncion. Asynchronous distributed learning of topic models. In *NIPS*, 2009.

[89] Sangho Suh, Jaegul Choo, Joonseok Lee, and Chandan K Reddy. Local topic discovery via boosted ensemble of nonnegative matrix factorization. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 4944–4948. AAAI Press, 2017.

[90] Koh Takeuchi, Ryota Tomioka, Katsuhiko Ishiguro, Akisato Kimura, and Hiroshi Sawada. Non-negative multiple tensor factorization. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 1199–1204. IEEE, 2013.

[91] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015.

[92] Lei Tang and Huan Liu. Leveraging social media networks for classification. *Data Mining and Knowledge Discovery*, 23(3):447–478, 2011.

[93] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 conference of the North American chapter of the association for computational linguistics on human language technology-volume 1*, pages 173–180. Association for Computational Linguistics, 2003.

[94] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C Pocock, Stephen Green, and Guy L Steele. Augur: Data-parallel probabilistic modeling. In *NIPS*, 2014.

[95] Jean-Baptiste Tristan, Joseph Tassarotti, and Guy Steele. Efficient training of lda on a gpu by mean-for-mode estimation. In *ICML*, 2015.

[96] Jeroen BP Vuurens, Carsten Eickhoff, and Arjen P de Vries. Efficient parallel learning of word2vec. *arXiv preprint arXiv:1606.07822*, 2016.

[97] Jim Jing-Yan Wang, Xiaolei Wang, and Xin Gao. Non-negative matrix factorization by maximizing correntropy for cancer clustering. *BMC bioinformatics*, 14(1):107, 2013.

[98] Yi Wang, Hongjie Bai, Matt Stanton, Wen-Yen Chen, and Edward Y Chang. Plda: Parallel latent dirichlet allocation for large-scale applications. *AAIM*, 2009.

[99] Max Welling and Markus Weber. Positive tensor factorization. *Pattern Recognition Letters*, 22(12):1255–1261, 2001.

[100] Brendan Whitaker, Denis Newman-Griffis, Aparajita Haldar, Hakan Ferhatosmanoglu, and Eric Fosler-Lussier. Characterizing the impact of geometric properties of word embeddings on task performance. In *Proceedings of the Third Workshop on Evaluating Vector Space Representations for NLP (RepEval)*, Minneapolis, MN, 2019. Association for Computational Linguistics.

[101] Han Xiao and Thomas Stibor. Efficient collapsed gibbs sampling for latent dirichlet allocation. In *ACML*, 2010.

[102] Pei Xue, Tao Li, Kezhao Zhao, Qiankun Dong, and Wenjing Ma. Glda: Parallel gibbs sampling for latent dirichlet allocation on gpu. In *ACA*. Springer, 2016.

[103] Feng Yan, Ningyi Xu, and Yuan Qi. Parallel inference for latent dirichlet allocation on graphics processing units. In *NIPS*, 2009.

[104] Zi Yang and George Michailidis. A non-negative matrix factorization method for detecting modules in heterogeneous omics multi-modal data. *Bioinformatics*, 32(1):1–8, 2015.

[105] R. Zafarani and H. Liu. Social computing data repository at ASU, 2009.

[106] Daojian Zeng, Kang Liu, Yubo Chen, and Jun Zhao. Distant supervision for relation extraction via piecewise convolutional neural networks. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1753–1762, 2015.

[107] Daojian Zeng, Kang Liu, Siwei Lai, Guangyou Zhou, and Jun Zhao. Relation Classification via Convolutional Deep Neural Network. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 2335–2344. Dublin City University and Association for Computational Linguistics, 2014.

[108] Bingjing Zhang, Bo Peng, and Judy Qiu. High performance lda through collective model communication optimization. *Procedia Computer Science*, 2016.

[109] Qiang Zhang, Michael W Berry, Brian T Lamb, and Tabitha Samuel. A parallel nonnegative tensor factorization algorithm for mining global climate data. In *International Conference on Computational Science*, pages 405–415. Springer, 2009.

[110] Sheng Zhang, Weihong Wang, James Ford, and Fillia Makedon. Learning from incomplete ratings using non-negative matrix factorization. In *Proceedings of the 2006 SIAM international conference on data mining*, pages 549–553. SIAM, 2006.

[111] Huasha Zhao, Biye Jiang, John F Canny, and Bobby Jaros. Same but different: Fast and high quality gibbs parameter estimation. In *SIGKDD*. ACM, 2015.