Efficient Computation of Co-occurrence Statistics for Natural Language Processing Tasks

Balázs Kovács* BKOVACS@CS.CORNELL.EDU
Cornell University Gates Hall Ithaca, NY 14853

Jack Hessel* JHESSEL@CS.CORNELL.EDU
Cornell University Gates Hall Ithaca, NY 14853

Abstract

Co-occurrence matrices act as the input to many unsupervised learning algorithms, including those that learn word embeddings, and modern spectral topic models. However, the computation of these inputs often takes longer than the inference. While much thought has been given to implementing fast learning algorithms, there has been little consideration of preprocessing efficiency. Here, we compare state-of-the-art methods for co-occurrence matrix computation in both the context-level and document-level cases. We find that both of these tasks are well suited to GPU parallelization. In the context-level case, our implementation outperforms an existing optimized single-core version by 4 times, and scales better. In the document-level case, we achieve significantly more than 350 times speedup over a naive, single-core baseline.

1. Introduction

Many unsupervised natural language processing algorithms rely on heavy precomputation/preprocessing before learning may take place. In fact, these preprocessing steps often take longer than the learning itself. When such algorithms are considered in an end-to-end fashion, current efficiency bottlenecks are not problems often considered by machine learning researchers. While considerable effort has been spent to increase the efficiency of Markov chain sampling and gradient descent algorithms, for instance, very little has been addressed to current expensive preprocessing steps.

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

More specifically, there are several types of large matrices that act as inputs to unsupervised learning algorithms. Consider Pennington et al.'s (2014) popular algorithm GloVe for computing word embeddings. A word embedding is a low-dimensional (100, for example) vector representation of a given word. The goal is to map each word type to a vector in a relatively low dimensional space, such that the relationships between the meaning of the words roughly correspond to the spatial relationships of the vectors. Given a set of documents D, GloVe requires the construction of a word-by-word co-occurrence matrix X where X_{ij} is a statistic related to how many times word i and word j appear close together, averaged over all documents. According to their documentation, "populating this matrix requires a single pass through the entire corpus to collect the statistics. For large corpora, this pass can be computationally expensive, but it is a one-time up-front cost."

For many users of natural language processing, however, corpora are not static. Web search engines, for instance, are in constant flux, as webpages are added, deleted, and edited. Also, as users of social media react to world events, the language of social interaction can very wildly, even on a minute-by-minute basis. In cases like these, models trained on a cached matrix are sub-optimal at best, and misleading at worst. Furthermore, for very large document collections, the up front cost of precomputation can still be very high.

Even for static corpora, efficiently preprocessing text will likely become more important not only as datasets get larger, but also as information with strict privacy requirements becomes more available. The differential privacy framework (Dwork, 2011) gives a method for data scientists to run experiments on aggregate statistics computed from data without learning about individual records that comprise those datasets. Algorithms that preprocess by adding a small amount of privacy-preserving noise to

* indicates equal contribution

datasets (e.g. Zhu et al. (2010)) will allow experiments to be run on new types of datasets. To distribute privatized datasets, preprocessing will need to be executed many times, and so there will be more interest in efficient processing methods in the future.

To this point, we have referred only to word embeddings as a case where co-occurrence counting is applied in natural language processsing. However, this preprocessing step is ubiquitous in NLP and an efficient implementation would be relevant to other learning tasks as well. Arora et al. (2013) and Lee et al. (2015) require as input a word-word co-occurrence count matrix at a *document-level* (rather than a *context-level*, as with GloVe) for a spectral topic modeling algorithm, for instance.

In this paper, we describe and implement very fast text processing algorithms using GPUs. Our implementations will be made freely available. We evaluate our methods and other previous solutions for computing both co-occurrence matrices, and report significant improvements in both cases.

2. Related Work

Unsupervised natural language processing algorithms that are computable over large, unstructured corpora are increasingly prevalent. These methods allow practitioners to understand at a high level what is being said in a given set of text documents. In most cases, these algorithms take as input a matrix of co-occurrence statistics.

For example, topic modeling algorithms including LSA (Deerwester et al., 1990) pLSA (Hofmann, 1999) and LDA (Blei et al., 2003) cluster documents based on word co-occurrence statistics. These three algorithms operate on the *word-document* co-occurrence matrix, which counts the number of times each word appears in each document. This matrix is relatively easy to compute, even for large corpora. In a naive implementation, one could perform a linear scan through each document, and, upon encountering word i in document j simply increment X_{ij}.

Recently proposed algorithms rely on *word-word*, or co-occurrence matrices. These methods include a number of algorithms for computing low-dimensional neural word representations from large corpora (Mikolov et al., 2013; Levy & Goldberg, 2014; Pennington et al., 2014). Also, word-word co-occurrences have also been applied to more traditional topic modeling tasks (Lee et al., 2015; Arora et al., 2013). The word-word co-occurrence matrix is more difficult to compute than the word-document co-occurrence matrix. Consider a naive implementation that counts words co-occurring in a document: for each word in a document, an entire pass must be made over the rest document to update the overall statistics. This causes the complexity of the naive approach to be quadratic in the length of a document, rather than linear as in the word-document matrix's computation. Some of these algorithms don't require a full pass over the document for each word, instead focusing on small "context windows" around. However, a linear scaling of runtime with window size still causes the naive approach to be slow.

In this work, we consider efficient implementations to compute word-word co-occurrence matrices in the *document-level* and *context-level* cases.

State of the art approaches to compute these are complicated. Pennington et al. (2014) provide a context-level implementation based on pthreads that, in an on-line fashion, identifies a dense subregion of the co-occurrence matrix, and keeps this matrix in memory while writing the low frequency submatrix to the filesystem dynamically. Their approach is not designed to run on specialized hardware, and still represents a significant computational overhead.

Mikolov et al.'s (2013) context-level implementation of word2vec avoids this problem by computing the machine learning optimization in an on-line fashion, so the computation of the co-occurrence matrix is interleaved with the learning. This methods is fast, but has some downsides however. For one, learning must begin with only partial knowledge of the dataset. For normal stochastic optimization methods, this is not detrimental. However, it's not clear that word2vec's learning couldn't be significantly faster/better if the entire matrix was known a-priori. Also, without a co-occurrence matrix, learning could not be restarted with a different set of parameters without passing over the whole dataset again. Furthermore, it's less clear that the online learning approach would lead to speedups if the precomputed co-occurrence matrix is available. Finally, it might be harder to guarantee privacy for online learning algorithms like this.

In contrast, to our knowledge, there are no highly efficient document-level co-occurrence implementations available publicly. This is likely because spectral topic models are relatively new.

To our knowledge, GPUs or other specialized hardware, have never been used to explicitly compute word-to-word co-occurrence. In the past, they have been used to compute related statistics in medical contexts (Hartley et al., 2014) with elaborate hardware-aware optimizations using CUDA. An important difference between our goal and theirs is that they compute local co-occurrences of image pixel values, while we work with words and co-occurrences of a larger memory area – a whole document or a substantial part of the document. This fact changes the way the memory access is optimized, which is the usual bottleneck in CUDA applications (Hartley et al., 2014). Another

Number of CUDA cores	3072
Number of multiprocessors	24
Cores per multiprocessor	128
32-bit Registers per multiprocessor	64K
Shared memory per multiprocessor	96K
Threads per multiprocessor	2048

Table 1. CUDA Specifications for the Titan X GPU.

approach uses MapReduce to compute word-to-word co-occurrences (Lin, 2008), but does not leverage the parallel computation power of GPUs. Wittek and Darányi (2013) combines MapReduce and GPGPU technologies, but uses GPU only for computationally intensive tasks like the distributed self-organizing maps algorithm.

3. The Tools We Use

3.1. CUDA Libraries

For our algorithms, we rely on Nvidia's CUDA (NVIDIA Corporation, 2015) libraries. These libraries provide programmers with access to the parallel computing capabilities of graphics processing units. For the most part, we utilize the Titan X GPU, whose specifications are provided in Table 1.

We use the CUDA library in two ways. First, in the case of document-level co-occurrence, we are able to write the algorithm to compute normalized word-word co-occurrence in terms of sparse and dense matrix operations. To compute these matrix operations quickly, we make optimized calls to cuBLAS, CUDA's implementation of BLAS, and cuSPARSE, a set of functions for doing matrix algebra on sparse matrices. In both cases, the Nvidia libraries are substantially faster when compared to a multi-threaded CPU implementation. [1]

Because the GPU has a memory space separate from main memory, the onus is on the programmer to copy data back and forth between the GPU device and host via system calls. However, for simplicity, we utilize the Caffe library's "Blob" implementation (Jia et al., 2014) which provides a high-level, synchronized array abstraction, while efficiently handling memory copying behind the scenes.

3.2. CUDA Architecture

Before describing our approach, we will give a brief overview of the CUDA computing architecture for better understanding.

Nvidia's CUDA provides a convenient interface to execute a piece of code (called a *kernel*) on multiple GPU threads in parallel. However, because the hardware has it's own processors and memory units, there are a lot of important details which must be kept in mind to achieve peak performance (Hartley et al., 2014; NVIDIA Corporation, 2015):

1. The threads are decomposed into *blocks* defined by the programmer. Each block is run on exactly one multiprocessor, and threads in the block share a fast memory called *shared memory*, in addition to a texture cache (for graphics applications). The shared memory and texture cache share the same memory and the ratio between them can be changed programmatically.
2. The slower *global memory* is accessible by all threads running on the GPU.
3. Multiple blocks can be assigned to a multiprocessor by the scheduler. The threads corresponding to the blocks share the resources of the multiprocessor, specifically a certain number of registers. Thus, the number of registers our kernel uses limits the parallelism we can achieve.

cuBLAS and cuSPARSE are proprietary matrix libraries developed internally by Nvidia, so it is safe to assume these implementations keep these programming paradigms in mind. However, some of our approach involves writing our own kernels, and in those cases, we are careful to program with the hardware in mind.

3.3. Building a Vocabulary

In both the document-level and context-level cases, it is helpful to define a *vocabulary* of words of interest prior to any computation. We use Pennington et al.'s (2014) fast vocabulary finder prior to running our algorithms. Their approach can compute the most common words in a corpus of billions of tokens in a few seconds.

While this step is not a computational bottleneck, in the future, it might be sufficient to compute the vocabulary over only a random subset of the documents. Assuming documents/tokens are well-mixed, sampling a subset should produce a stable estimate of the most common words. Alternatively, the user may pre-specify their own vocabulary of interest, given our implementation.

3.4. Sparse Matrices on the GPU

Even though both the word-by-document matrix and the word-by-word co-occurrence matrix can be very large for usual datasets (which might consist of millions of documents and up to $100K$ vocabulary items), these matrices often have sparsity we can exploit.

For instance, it is well known that the word-document matrix is usually sparse (Sahlgren & universitet. Institutio-

[1] `http://tinyurl.com/cuda-benchmark`

Algorithm 1 Document-level Matrix Formulation

Input: Word-by-document count matrix D
Returns: Normalized word-word matrix X
$p = colsums(D)$
compute n such that $n_i = 1/(p_i \cdot (1 - p_i))$
$A = D \cdot diag(\sqrt{n})$
$M_1 = AA^T$
$M_2 = diag(X \cdot n)$
return $(M_1 - M_2)/(\#Docs)$

nen för lingvistik, 2006). This sparsity is a result of Zipf's law, which states that the i^{th} most common word in a language appears with frequency proportional to $1/i$ (Zipf, 1949). In other words: language contains lots of rare words. This means that any two given words are unlikely to co-occur. This sparsity helps us fit very large matrices into the limited GPU memory. We use a compressed row storage format (which is compatible with cuSPARSE) to store and operate on matrices.

4. Co-occurrence Implementations

4.1. Document-level Co-occurrence

In this case, we are interested in computing a word-to-word co-occurrence matrix X where X_{ij} is related to the number of times words i and j appear in the same document. The matrix we hope to end up with, however, is not a matrix of raw counts. Instead, the i, j^{th} entry of this matrix represents the probability that words i and j co-occur together, as in Arora et al. (2013) and Lee et al. (2015). This matrix must also normalize based on the length of the document, such that the matrix represents a joint distribution and sums to unity.

Let v_i be the vector (whose length is equal to the vocabulary size) of word counts for document i, and let $n_i = \sum_j v_{ij}$ be the length of document i. The co-occurrence matrix we are interested in is computed as a sum over all documents as

$$X = \sum_i \frac{v_i v_i^T - diag(v_i)}{n_i \cdot (n_i - 1)} \quad (1)$$

In this formulation, note that

$$X_{ij} = P(w_1 = i, w_2 = j | w_1 \text{ and } w_2 \text{ appeared together.})$$

We investigate three algorithms for the computation of this matrix. As a baseline, our first implementation uses a single-thread to compute each term in the sum in Equation 1 sequentially, on a document-by-document basis.

Next, we note that it is relatively straightforward to compute the word-by-document matrix, D, (potentially in parallel) on the CPU. Given this matrix, the expensive aspect of the computation of X (which can be thought of roughly as computing DD^T) can be decomposed to a small number of matrix operations. It can be shown that Algorithm 1 is equivalent to Equation 1. Our next implementation uses this formulation, making cuSPARSE calls directly.

In our experiments, we noted that the most computationally expensive part of Algorithm 1 was computing AA^T for a sparse matrix A. A has dimensions V_{count} by D_{count} where V_{count} is the size of the vocabulary and D_{count} is the number of documents in the corpus. While A is a sparse matrix, AA^T is not, and often contains over 85% nonzero elements. While there are some publicly available benchmarks, it's not clear how cuSPARSE performs when constructing a dense matrix in a sparse format, as we do in this case. Also, it's not clear how the speed of cuSPARSE's sparse to dense conversion depends on the sparsity of the input matrix.

As such, we implement a batched version of our word-word co-occurrence. This approach calls Algorithm 1 on smaller batches of 20K documents, and converts and accumulates the outputs on-line. By reducing the number of documents per call, we make the input matrix D skinnier, and, consequently, AA^T sparser. While the matrix remains relatively dense due to common words frequently co-occurring, in terms of nonzero elements, the density is decreased to around 1% for large vocabularies and 20% for small vocabularies. This implementation introduces extra overhead (memory copying, function calls, etc.) in favor of keeping AA^T sparse, so that cuSPARSE can operate exclusively on sparse matrices. The batch size is a parameter that could be optimized on a case-by-case basis.

4.2. Context-level Co-occurrence

For some NLP applications, the co-occurrence statistics required are not document-level statistics – if two words appear on the opposite ends of a text document, are they truly related?

Algorithms for computing word embeddings (Mikolov et al., 2013; Levy & Goldberg, 2014; Pennington et al., 2014), for instance, rely on a narrow, local neighborhood to be considered for co-occurrence and also might have some weighting function over this neighborhood (exactly neighboring words are likely more related than words a few places apart). Thus, we also implement a version of the algorithm which computes the co-occurrences over the local neighborhood, called the *context*, of each word.

Notice that the notion of a "document" is less important in the case of a sliding context window. For the purposes of this paper, in accordance with previous work, we will concatenate all documents together and disregard the small error introduced at the boundaries. These don't affect the

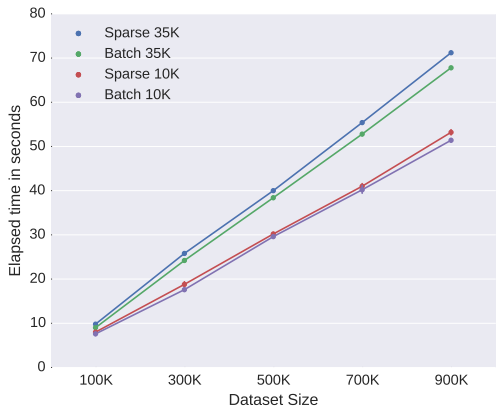

Figure 1. Dataset size (in documents) versus runtime for document-level word-word co-occurrence computation, for vocab sizes of 10K words and 35K words. "Sparse" is Algorithm 1 and "Batch" is the batched version of that approach. Note that batching is faster than not batching, and that this speedup might be more pronounced for larger vocabulary sizes.

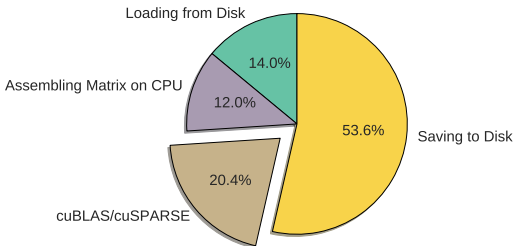

Figure 2. Running time of different parts of the processing for the document-level case on the 900K document dataset with vocabulary size 35K. Processing parts are roughly categorized into input/output, CPU computation, and GPU computation. The full processing time is 151 seconds, which includes saving.

quality of the final result substantially (see Mikolov et al.'s (2013) popular implementation).

Let us denote the window size as w and the associated word-to-word co-occurrence matrix as X. Like in the document-level case, we do not store raw word-counts in X; this is because word embedding algorithms assume preprocessed input. As a proof-of-concept and for easy comparison, we implement context weighting as described by Pennington et al. (2014), though this could be interchanged with any context-level statistic (e.g. pairwise mutual information) very easily.

We make two observations about X to decide what representation to use: (1) For large corpora it is not very sparse ($20\% - 90\%$ non-zero elements, depending on the vocabulary size) (2) To build a matrix with sparse representation on the GPU, we need to use locking, which would slow down the execution significantly. Therefore, we store X as a dense matrix and use atomic instructions to update its values from the kernel. Because of memory limitations and the limitations of the cuSPARSE library, the maximum vocabulary size we can use is 40K. While this size is relatively small for word-embedding applications (often a vocabulary of roughly 100K is used) it is usual for topic models to be computed with this many vocabulary items. We leave a larger vocabulary implementation to future work.

In our case, we make one pass through the whole corpus using the CPU and convert each word to an integer ID using the given vocabulary. This representation is more compact and facilitates efficient memory access on the GPU.

Second, we initialize X's values to zero in the GPU global memory. We launch a CUDA kernel, where each thread is responsible for computing co-occurrences for one word context window and adding these values to the corresponding elements of X using *atomicAdd*.

We can improve the performance of this algorithm if we use the shared memory of the multiprocessors to cache a chunk of the document. Each thread block first preloads a chunk in parallel and synchronizes the threads to ensure that every load operation has finished. Now we can count the co-occurrences and add them to X in the global memory. We report the performance of this final method in the evaluation section.

5. Evaluation

5.1. Hardware

For all experiments we use a workstation with one Nvidia Titan-X GPU, Intel Core i7-5820K 3.30 GHz CPU and three Seagate ST1000DM003 disks in RAID 5.

5.2. Datasets

To compare our approach to other methods, we compute document-level and context-level co-occurrence matrices for various datasets and vocabulary sizes. The corpora we use for document-level co-occurrence are subsets of 2M text posts made to `reddit.com` (Tan & Lee, 2015). The previous largest real dataset to be processed in this manner consisted of 300K documents (Sandhaus, 2008).

For the context-level experiments, we use the billion word language modeling benchmark of Chelba et al. (2013). This dataset consists of slightly less than 1B tokens (around 719M), and is commonly used to evaluate language models.

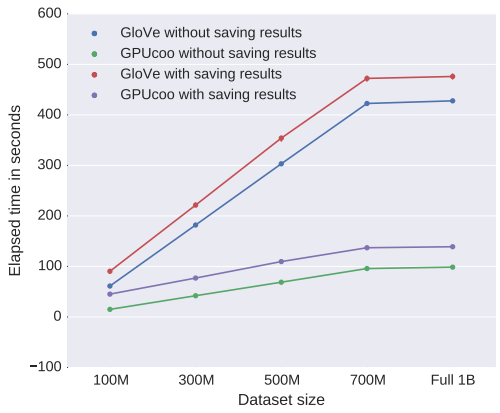

Figure 3. Context-level co-occurrence results; GPUcoo is our algorithm, and GloVe is an existing single-core baseline. Presented here are running times with and without saving the results to disk for increasing dataset sizes with vocabulary size 10K.

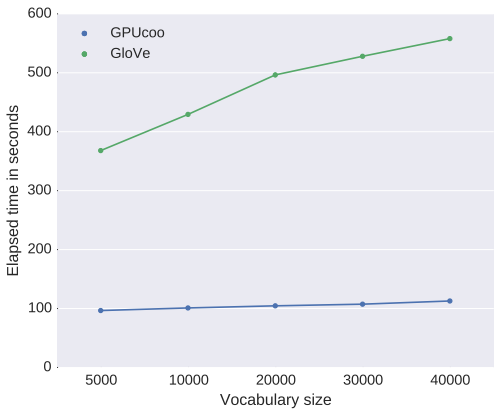

Figure 4. A demonstration of the effect of increasing vocabulary size. Here, we present running times for the context-level case with and without saving the results to disk. The x-axis represents increasing vocabulary size. The total number of tokens analyzed is around 719M.

5.3. Document-level experiments and results

Because the authors are not aware of any optimized single-core implementation, we compare the GPU implementations against a baseline that iterates Equation 1, one document at a time. We don't perform many experiments on this baseline approach because it is very slow. For reference, the baseline approach takes over 30 minutes to compute the document-level co-occurrence matrix in a 10K document/5K vocab toy case, compared to the GPU approach which takes about 6 seconds. This represents an over *350x* speedup. The naive version likely doesn't scale well either, so we would expect this speedup to be even greater for lager corpora/vocabularies. The main goal of the experiments in this section is to determine if and when the batched version of Algorithm 1 (which we call "Batch") outperforms the non-batched version (which we call "Sparse").

We run trials for the batched and non-batched versions of Algorithm 1 for corpora of different sizes (from 100K to 900K documents) and vocabulary of different sizes (from 10K to 35K items, in 5K increments). We summarize the results of these experiments in Figure 1; we only display results for 10K/35K vocab for clarity, and don't consider the time it takes to save the results because the two approaches we compare here use the same IO. In general, sparse and batching perform similarly, which indicates that any differences are minor. Batching is slightly faster in all cases, and this effect is perhaps more pronounced as the vocabulary size increases. This is possibly because the batch AA^T becomes much more sparse as the vocabulary size increases, making storage in cuSPARSE faster.

Figure 2 shows a breakdown of how much time time the implementation spends in each part of the computation (output time is included here). Saving accounts for over half of the processing time.

5.4. Context-level experiments and results

In our context-level experiments, we compare against Pennington et al.'s (2014) optimized single core implementation. We use context window size of 15 in all experiments. The first experiment compares the running time of the two algorithms for different corpus sizes using vocabulary size 10K. All running times include loading the corpus from disc and processing the data. For both algorithms, we measure the running time with and without saving the final word-word co-occurrence matrix to disk. As we can see in Figure 3, our algorithm outperforms the baseline by a significant margin, for the full 1B token dataset (which consists of closer to 719M tokens) it is *4x* faster.

Unfortunately we couldn't measure how much time the baseline spends separately on computing the co-occurrences, because that computation is interleaved with I/O processing. Our GPU implementation finishes the computation in less than 21 seconds in all cases, after having all data on the GPU. This suggests that we should concentrate on further optimizing the I/O processing in future work. Figure 5 shows a breakdown of the processing. Processing parts which take negligible time appear in parentheses and their processing time is added to the closest processing part.

In the second experiment we measure the running time for different vocabulary sizes on the 1B token dataset without saving the results to disk. We can see in Figure 4 that our algorithm scales very well for larger vocabulary sizes and is more than *4x* faster than the baseline. Only the cuSPARSE

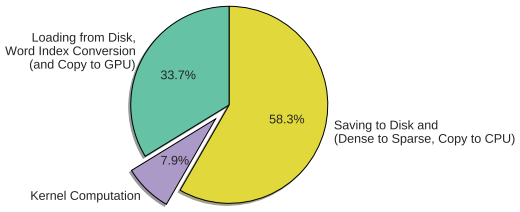

Figure 5. Running time of different parts of the processing for the context-level case on the 1B token dataset with vocabulary size 40K. Processing parts which take negligible time appear in parentheses and their processing time is added to the closest processing part. The full processing time is 264 seconds, which includes saving.

API and memory limits constrain us to stop at vocabulary size 40K, a concern we save for future work.

6. Conclusion and Future Work

Here, we have presented preliminary implementations to compute word-word co-occurrence matrices on a GPU for natural language processing tasks. In cases where the user is interested in computing co-occurrence on a *document-level* basis, we report greater than *350x* speedup over a naive, single-core implementation. In cases where the user is interested in computing co-occurrence on a *context-level* basis, we report *4x* over a highly optimized single-core implementation.

There are several directions we hope to take this work in the future. First, we aim to parallelize our file IO; this aspect of our computation is currently a bottleneck and the problem of quickly reading and writing to disk will increasingly effect overall runtime.

More broadly, however, we hope to use the insights we've gained creating these implementations to speed up machine learning algorithms. Memory bottlenecks of GPUs will continue to decline as technology advances, and so restating natural language processing algorithms to be GPU-friendly will be highly relevant in the future.

Co-occurrence counting has been a part of unsupervised natural language processing ever since large corpora have been treated as data (see Deerwester et al. (1990)). Even when algorithms are designed to go beyond counting, they often reduce to something very close to counting (Levy & Goldberg, 2014). A key advantage machine learning practitioners today have when compared to those of the past is specialized systems and hardware (e.g. high memory GPUs). These pieces of hardware allow for previously intractable learning algorithms to be run on larger and larger datasets. Most importantly, we hope this work serves as a motivating example for researchers interested in fast NLP.

7. Acknowledgments

We would like to thank David Mimno and all the participants in the Fall 2015 iteration of Cornell's Advanced Systems course for their helpful insights and feedback.

References

Arora, Sanjeev, Ge, Rong, Halpern, Yonatan, Mimno, David, Moitra, Ankur, Sontag, David, Wu, Yichen, and Zhu, Michael. A practical algorithm for topic modeling with provable guarantees. In *Proceedings of The 30th International Conference on Machine Learning*, pp. 280–288, 2013.

Blei, David M, Ng, Andrew Y, and Jordan, Michael I. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

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

Deerwester, Scott C., Dumais, Susan T, Landauer, Thomas K., Furnas, George W., and Harshman, Richard A. Indexing by latent semantic analysis. *JAsIs*, 41(6):391–407, 1990.

Dwork, Cynthia. Differential privacy. In *Encyclopedia of Cryptography and Security*, pp. 338–340. Springer, 2011.

Hartley, Timothy DR, Catalyurek, Umit, Ruiz, Antonio, Igual, Francisco, Mayo, Rafael, and Ujaldon, Manuel. Biomedical image analysis on a cooperative cluster of gpus and multicores. In *25th Anniversary International Conference on Supercomputing Anniversary Volume*, pp. 413–423. ACM, 2014.

Hofmann, Thomas. Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 50–57. ACM, 1999.

Jia, Yangqing, Shelhamer, Evan, Donahue, Jeff, Karayev, Sergey, Long, Jonathan, Girshick, Ross, Guadarrama, Sergio, and Darrell, Trevor. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Lee, Moontae, Mimno, David, and Bindel, David. Robust spectral inference for joint stochastic matrix factorization. In *Advances in neural information processing systems*, 2015.

Levy, Omer and Goldberg, Yoav. Neural word embedding as implicit matrix factorization. In *Advances in Neural Information Processing Systems*, pp. 2177–2185, 2014.

Lin, Jimmy. Scalable language processing algorithms for the masses: A case study in computing word co-occurrence matrices with mapreduce. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pp. 419–428, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics. URL http://dl.acm.org/citation.cfm?id=1613715.1613769.

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

NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide v7.5*. NVIDIA Corporation, 2015.

Pennington, Jeffrey, Socher, Richard, and Manning, Christopher D. Glove: Global vectors for word representation. *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*, 12:1532–1543, 2014.

Sahlgren, M. and universitet. Institutionen för lingvistik, Stockholms. *The Word-Space Model: Using Distributional Analysis to Represent Syntagmatic and Paradigmatic Relations Between Words in High-dimensional Vector Spaces*. SICS dissertation series. Department of Linguistics, Stockholm University, 2006. ISBN 9789171552815. URL https://books.google.com/books?id=swCOtgAACAAJ.

Sandhaus, Evan. The new york times annotated corpus. *Linguistic Data Consortium, Philadelphia*, 6(12): e26752, 2008.

Tan, Chenhao and Lee, Lillian. All who wander: On the prevalence and characteristics of multi-community engagement. In *Proceedings of the 24th International Conference on World Wide Web*, pp. 1056–1066. International World Wide Web Conferences Steering Committee, 2015.

Wittek, Peter and DaráNyi, SáNdor. Accelerating text mining workloads in a mapreduce-based distributed gpu environment. *Journal of Parallel and Distributed Computing*, 73(2):198–206, 2013.

Zhu, Yun, Xiong, Li, and Verdery, Christopher. Anonymizing user profiles for personalized web search. In *Proceedings of the 19th international conference on World wide web*, pp. 1225–1226. ACM, 2010.

Zipf, George Kingsley. Human behavior and the principle of least effort. 1949.