BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs

Saurabh Gupta
Amazon Web Services
gsaur@amazon.com

Vineet Khare
Amazon Web Services
vkhare@amazon.com

ABSTRACT

Word2Vec is a popular algorithm used for generating dense vector representations of words in large corpora using unsupervised learning. The resulting vectors have been shown to capture semantic relationships between the corresponding words and are used extensively for many downstream natural language processing (NLP) tasks like sentiment analysis, named entity recognition and machine translation. Most open-source implementations of the algorithm have been parallelized for multi-core CPU architectures including the original C implementation by Mikolov et al. [1] and FastText [2] by Facebook. A few other implementations have attempted to leverage GPU parallelization but at the cost of accuracy and scalability. In this work, we present BlazingText, a highly optimized implementation of word2vec in CUDA, that can leverage multiple GPUs for training. BlazingText can achieve a training speed of up to 43M words/sec on 8 GPUs, which is a 9x speedup over 8-threaded CPU implementations, with minimal effect on the quality of the embeddings.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Natural language processing**;

KEYWORDS

Word embeddings, Word2Vec, Natural Language Processing, Machine Learning, CUDA, GPU

ACM Reference format:
Saurabh Gupta and Vineet Khare. 2017. BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs. In *Proceedings of MLHPC'17: Machine Learning in HPC Environments, Denver, CO, USA, November 12–17, 2017,* 5 pages.
https://doi.org/10.1145/3146347.3146354

1 INTRODUCTION

Word2Vec aims to represent each word as a vector in a low-dimensional embedding space such that the geometry of resulting vectors captures word semantic similarity through the cosine similarity of corresponding vectors as well as more complex relationships through vector subtractions, such as vec("King") - vec("Queen") + vec("Woman") ≈ vec("Man"). This idea has enabled many Natural Language Processing (NLP) algorithms to achieve better performance [3, 4].

The optimization in word2vec is done using Stochastic Gradient Descent (SGD), which solves the problem iteratively; at each step, it picks a pair of words: an input word and a target word either from its window or a random negative sample. It then computes the gradients of the objective function with respect to the two chosen words, and updates the word representations of the two words based on the gradient values. The algorithm then proceeds to the next iteration with a different word pair being chosen.

One of the main issues with SGD is that it is inherently sequential; since there is a dependency between the update from one iteration and the computation in the next iteration (they may happen to touch the same word representations), each iteration must potentially wait for the update from the previous iteration to complete. This does not allow us to use the parallel resources of the hardware.

However, to solve the above issue, word2vec uses Hogwild [5], a scheme where different threads process different word pairs in parallel and ignore any conflicts that may arise in the model update phases. In theory, this can reduce the rate of convergence of algorithm as compared to a sequential run. However, the Hogwild approach has been shown to work well in the case updates across threads are unlikely to be to the same word; and indeed for large vocabulary sizes, conflicts are relatively rare and convergence is not typically affected.

The success of Hogwild approach for Word2Vec in case of multi-core architectures makes this algorithm a good candidate for exploiting GPU, which provides orders of magnitude more parallelism than a CPU. In this paper, we propose an efficient parallelization technique for accelerating word2vec using GPUs.

GPU acceleration using deep learning frameworks is not a good choice for accelerating word2vec [6]. These frameworks are often suitable for "deep networks" where the computation is dominated by heavy operations like convolutions and large matrix multiplications. On the other hand, word2vec is a relatively shallow network, as each training step consists of an embedding lookup, gradient computation and finally weight updates for the word pair under consideration. The gradient computation and updates involve small dot products and thus don't benefit from the use of cuDNN [7] or cuBLAS [8] libraries.

The limitations of deep learning frameworks led us to explore the CUDA C++ API. We design the training algorithm from scratch, to utilize CUDA multi-threading capabilities optimally, without hurting the output accuracy by *over-exploiting* GPU parallelism.

Finally, to scale out BlazingText to process text corpus at several million words/sec, we demonstrate the possibility of using multiple GPUs to perform data parallelism based training, which is one of the main contributions of our work. We benchmark BlazingText against

MLHPC'17: Machine Learning in HPC Environments, November 12–17, 2017, Denver, CO, USA
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5137-9/17/11.
https://doi.org/10.1145/3146347.3146354

commonly available open-source tools to illustrate its superior performance.

The rest of the paper is organized as follows. In Sec. 2, we describe the original word2vec model. In Sec. 3, we review the existing approaches to accelerate word2vec using GPUs or multi-node CPUs. We describe our efforts using CUDA C++ API in Sec. 4. Experimental results on Text8 and One Billion Words Benchmark dataset are presented in Sec. 5, followed by conclusion and future work in Sec. 6.

2 WORD2VEC MODEL

Word2vec represents each word w in a vocabulary V of size T as a low-dimensional dense vector v_w in an embedding space $\mathbb{R}^D$. It attempts to learn the continuous word vectors $v_w, \forall w \in V$, from a training corpus such that the spatial distance between words then describes the similarity between words, e.g., the closer two words are in the embedding space, the more similar they are semantically and syntactically. Inspired by the distributional hypothesis (Harris, 1954), these representations are trained to predict words appearing in the context of a given word. Under this hypothesis, two distinct model architectures: Contextual Bag-Of-Words (CBOW) and Skip-Gram with Negative Sampling (SGNS) are proposed in word2vec [1]. The objective of CBOW is to predict a word given its context, whereas Skipgram tries to predict the context given a word. In practice, Skipgram gives better performance and is described below.

More formally, given a large training corpus represented as a sequence of words $w_1, w_2 ... w_T$, the objective of the skipgram model is to maximize the log-likelihood

$$\sum_{t=1}^{T} \sum_{c \in C_t} \log p(w_c | w_t)$$

where T is the vocabulary size and the context C_t is the set of indices of words surrounding word w_t . The probability of observing a context word w_c given w_t will be parameterized using the aforementioned word vectors. For now, let us consider that we are given a scoring function s which maps pairs of (word, context) to scores in $\mathbb{R}$. One possible choice to define the probability of a context word is the softmax:

$$p(w_c | w_t) = \frac{\exp(s(w_t, w_c))}{\sum_{j=1}^{W} \exp(s(w_t, j))}$$

However, such a model is not adapted to our case as it implies that, given a word w_t, we only predict one context word w_c.

The problem of predicting context words can instead be framed as a set of independent binary classification tasks. Then the goal is to independently predict the presence (or absence) of context words. For the word at position t we consider all context words as positive examples and sample negatives at random from the dictionary. For a chosen context position c, using the binary logistic loss, we obtain the following negative log-likelihood:

$$\log(1 + e^{-s(w_t, w_c)}) + \sum_{n \in \mathcal{N}_{t,c}} \log(1 + e^{s(w_t, n)})$$

where $\mathcal{N}_{t,c}$ is a set of negative examples sampled from the vocabulary. By denoting the logistic loss function $l : x \mapsto \log(1 + e^{-x})$, we can re-write the objective as:

$$\sum_{t=1}^{T} \sum_{c \in C_t} \left[l(s(w_t, w_c)) + \sum_{n \in \mathcal{N}_{t,c}} l(-s(w_t, n)) \right]$$

A natural parameterization for the scoring function s between a word w_t and a context word w_c is to use word vectors. Let us define for each word w in the vocabulary two vectors u_w and v_w in $\mathbb{R}^D$. These two vectors are sometimes referred to as *input* and *output* vectors in the literature. In particular, we have vectors u_{w_t} and v_{w_c} , corresponding to words w_t and w_c respectively. Then the score can be computed as the dot product between word and context vectors as $s(w_t, w_c) = u_{w_t}^T v_{w_c}$.

3 RELATED WORK

While some existing word2vec systems are limited to running on a single machine CPU, there have been some efforts in accelerating word2vec using multi-core CPU nodes for distributed data parallel training. These include the training systems available in Apache Spark MLLib [9] and Deeplearning4j [10]. These systems rely on the *reduce* operation to synchronize the model between all executors after every iteration. Broadcasting all the word vectors across nodes limits the scalability as typical network bandwidths are an order of magnitude lower than CPU memory bandwidths. Moreover, the model accuracy plummets as more nodes are employed to increase the throughput.

The work done by Ji et al. [11] demonstrates strong scalability on CPU nodes by using a scheme based on minibatching and shared negative samples. Their approach converts level-1 BLAS operations into level-3 BLAS matrix multiply operations, hence efficiently leveraging the vectorized multiply-add instructions of modern architectures. However, they still don't leverage GPUs and their implementation scales well on Intel BDW and Intel KNL processors, which can be much more expensive than GPUs and are not yet provided by the major cloud services platforms. Using their idea, we shared the negative samples across a minibatch and used the highly optimized cuBLAS level-3 matrix multiplication kernels, but due to the small size of matrices being multiplied, the overhead of CUDA kernel-launches drastically reduced the performance and scalability.

Several other works [12, 13] have tried to utilize deep learning libraries like TensorFlow, Keras and Theano, but show even slower performance compared to the FastText CPU implementation. Since an iteration of word2vec is not very compute intensive, a large batch size is needed to fully utilize the GPU. However, mini-batching of training data reduces the rate of convergence dramatically and with a batch size of one, training becomes extremely slow.

4 GPU PARALLELIZATION USING CUDA

Since the GPU architecture provides orders of magnitude more parallelism than CPU cores, word2vec seems to be a good fit for acceleration using GPU as the algorithm itself exhibits good amount of parallelism as exploited by asynchronous SGD or Hogwild. However, as more parallelism is used, different threads might conflict with each other at the time of reading and updating word vectors, resulting in a huge accuracy drop. Thus, a careful consideration should be given to manage the trade-off between level of parallelism and synchronization.

Deep learning frameworks don't provide a fine-grained control over the scalability and parallelism of the GPU, as provided by the CUDA C++ API. This instills the need of re-factoring the algorithm design significantly, enabling maximum parallel throughput while forcing synchronization to prevent the slump in accuracy. We explored the following two approaches for implementing the algorithm using CUDA.

4.1 One Thread Block per word

The original word2vec implementation processes a sentence sequentially i.e. for each center word w_t, it considers all words in the window of size *ws* of that center word as target words, which means that all the words vectors in $[w_{t-ws}, w_{t+ws}]$ will get updated in one step. Similarly in the next step, for the center word $w_t + 1$, all the vectors in the range $[w_{t-ws+1}, w_{t+ws+1}]$ will be updated. This implies that when processing a sentence, a word vector can be modified upto $2ws + 1$ times and ideally, each consecutive step should use the updated vectors that were modified by the previous step.

Designing a CUDA program from scratch requires us to make decisions about the structure of grid and thread blocks. Threads within the same thread block can synchronize with each other, but threads belonging to different thread blocks can not communicate with each other, making the thread blocks independent of each other. In this approach, we choose to allocate a thread block per word, with the number of threads within a block to be a multiple of 32, for warp related efficiency. As the number of threads within a block is close to the vector dimension (usually 100), each thread maps to a vector dimension and does element-wise multiplication. These individual products are then efficiently summed using a *reduce* kernel to compute the dot-product between any 2 given vectors. Several parallel reduction techniques were explored and finally a highly optimized *completely unrolled* reduce kernel was employed. This algorithm design exploits the parallelism to its peak, as each word is processed independently by a thread block, and the threads within a thread block synchronize only when executing the reduce operation. However, the approach suffers from a major drawback that significantly undermines the quality of embeddings. Different thread blocks can modify the word vectors independently with no synchronization which can be very detrimental to the accuracy, since each vector can be updated upto $2w + 1$ times as the window slides over the sentence. This results in a large number of thread overwrites and stale reads.

4.2 One Thread Block per sentence

As discussed in the previous section, updating word vectors without any synchronization makes full use of CUDA parallelism but degrades the accuracy of embeddings due to race conditions. As the window slides over the sentence, *updated* previous word vectors should be used when processing the following word. Thus, to address the sequential dependency, this approach maps each sentence to a CUDA thread block and like the previous approach, it maps each dimension of the vector to a thread. So, different thread blocks process the sentences in parallel to each other, while within a sentence, the thread block loops over the words to update the vectors serially. This approach may still lead to some race conditions, but since different sentences do not have a lot of words in common, it does not cause much convergence problems in practice.

Since the text corpus can be too large to reside in GPU memory, data is streamed from disk to the GPU and several sentences are batched to amortize the cost of data transfer between CPU and GPU. To decide the trade-off between accuracy and throughput, the optimum number of thread blocks or sentences that should be processed in parallel are chosen empirically. Having more sentences to be processed concurrently increases the throughput but results in accuracy drop as the probability of different thread blocks updating the same word vector increases.

Several other optimizations are used to make this approach more efficient. If the kernel execution time is less than the data transfer time, then the GPU will sit idle waiting for the next batch of sentences. To avoid this, we try to overlap data transfer and kernel execution on GPU by using multiple CPU threads and CUDA streams, which allow data transfer to the GPU while it is busy running kernels, thus leaving no scope for idle GPU time. We use multiple CPU threads to read data from disk and prepare the next batch of sentences, which is concurrently transferred to the GPU using multiple CUDA streams.

We use this approach for all our experiments described in Section 5.

4.3 Distributed training on Multiple GPUs

Scaling BlazingText on a multi-GPU distributed system is critical because it can still take a couple of days on a single GPU to train on some of the biggest datasets in the industry, which can be of the order of several tera-bytes [14]. To scale out BlazingText, we explore different paradigms of distributed training - model parallelism and data parallelism. Since the vector dimensions are usually small, the scale of dot products is not that big and thus distributing different dimensions of the vectors will not provide too much performance gain. Hence, we use data parallelism, where we divide the dataset equally into N shards when using N GPUs. The model parameters - Input and Output vectors for all the words in vocabulary, are replicated on each GPU; each device then independently processes the data partition it owns and updates its local model, periodically synchronizing the local model with all other $N - 1$ GPUs.

The main issue to be addressed in data parallelism is efficient model synchronization between the GPUs. This is efficiently handled by using NVIDIA's NCCL library [15], which provides an AllReduce method that handles the peer-to-peer data transfer between the GPUs in an optimized way based on the topology of GPU network. If GPUs are not connected via the same PCIe switch or same IOH chip, then data is transferred through the host CPU. As the model parameters' size can be of the order of hundreds of MBs, synchronization can be costly and thus it very important to determine the right synchronization interval. Frequent synchronizations will result in better convergence but will slow down training and vice-versa. For simplicity, we choose to synchronize after every epoch and leave it for future work to explore more efficient synchronization schemes.

5 EXPERIMENTS

We optimize BlazingText with the techniques described above for single GPU and multiple-GPU systems. In all our experiments, we map each sentence to a thread block. In this section, we report the throughput(in million words/sec) and accuracy of learned embeddings on standard word similarity and word analogy test sets. We benchmark BlazingText against FastText CPU implementation **(without subword embeddings)**.

Hardware: All our experiments are performed on AWS p2.8xlarge GPU instance which has 8 NVIDIA K80 GPUs and Intel Xeon CPU E5-2686 v4 @ 2.30GHz with 16 cores (32 threads).

Software: BlazingText has been written in C++ using CUDA compiler - NVCC v8.0

Training corpora: We train our models on two different corpora: (1) Text8 dataset [16] of 17 million words from Wikipedia that is widely used for word embedding demos, (2) The One Billion Words benchmark dataset [17].

Test sets: The learned embeddings are evaluated on word similarity and word analogy tasks. For word similarity, we use WS-353 [18] which is one of the most popular test datasets used for this purpose. It contains word pairs together with human-assigned similarity judgments. The word representations are evaluated by ranking the pairs according to their cosine similarities, and measuring the Spearman's rank correlation coefficient with the human judgments. For word analogy, we use the Google analogy dataset [1] which contains word analogy questions. A question is correctly answered only if the algorithm selects the word that is exactly the same as the correct word in the question.

Hyperparameters: For all our experiments, we report the results using both CBOW and Skipgram algorithms (with negative sampling) and **FastText's default parameter settings** (dim = 100, window size = 5, sampling threshold = 1e-4, initial learning rate = 0.05). We use 20 epochs for Text8 dataset and 10 epochs for One Billion Words benchmark.

5.1 Throughput

Figures 1 and 2 show the throughput measured as million words/sec of BlazingText on GPU and FastText on CPU, scaling across multiple GPUs and CPU cores, for Skipgram and CBOW respectively. When scaling to multiple GPUs, our implementation achieves near linear speedup. Using 8 GPUs, Skipgram achieves 13.2 million words/sec while CBOW delivers about 42.5 million words/sec, which is more than 3x speedup over 32 threaded FastText. As evident from Table 1, scaling across multiple GPUs has minimal effect on accuracy, thus highlighting the effectiveness of our implementation and efficient use of multi-GPU architecture. As there is a trade-off between throughput and accuracy, we can further increase the throughput by lowering synchronization frequency, as long as the drop in accuracy is acceptable.

5.2 Accuracy

We evaluate the models trained from FastText CPU implementation and our implementation (with varying number of GPUs), and report their predictive performances on the word similarity and word analogy tasks in Table 1. To make sure that our implementation generalizes well, we used 2 different corpora for training the models - Text8 and 1B words benchmark.

To nullify the effects of randomness due to GPU parallelism, we run each experiment multiple times (n=10) and report the mean results. As can be seen from Table 1, BlazingText shows similar performance on mulitple GPUs as FastText CPU. When using upto 4 GPUs, the performance is almost identical, in some cases even better than the CPU version. As expected, the predictive performance

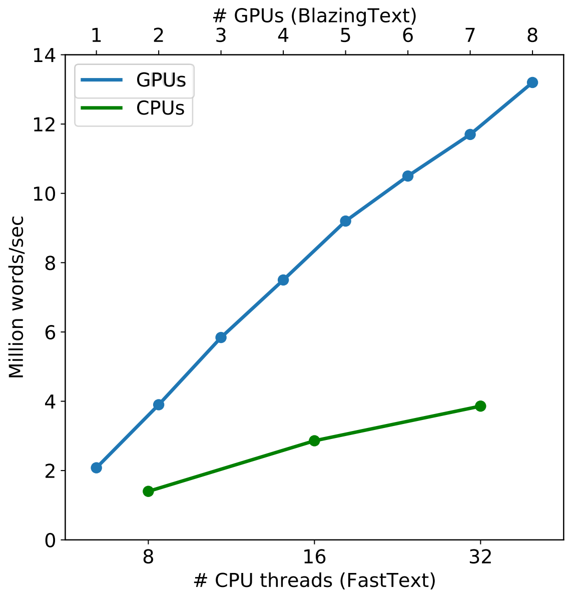

Figure 1: Skipgram throughput

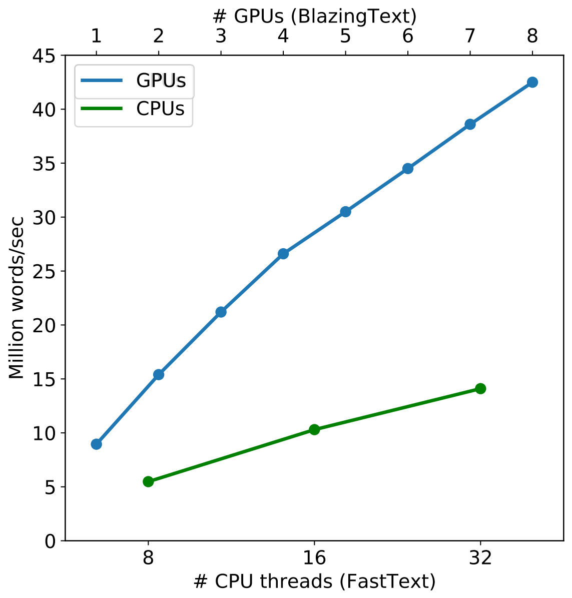

Figure 2: CBOW throughput

Table 1: Spearman's rank correlation coefficient between model scores and human judgement on WS-353 dataset for word similarity. For word analogy task, we report the accuracies on Google analogy dataset.

Training Corpus			# GPUs (BlazingText)								# CPUs (FastText)		
			1	2	3	4	5	6	7	8	8	16	32
Text8	Similarity	Skipgram	.716	.713	.710	.707	.703	.699	.695	.694	.707	.706	.7
		CBOW	.711	.708	.705	.689	.683	.681	.679	.675	.694	.689	.69
	Analogy	Skipgram	.327	.324	.321	.311	.299	.297	.296	.295	.329	.330	.326
		CBOW	.321	.329	.32	.299	.295	.289	.285	.281	.323	.326	.325
1 Billion word benchmark	Similarity	Skipgram	.659	.658	.656	.653	.655	.659	.651	.650	.660	.659	.656
		CBOW	.609	.607	.599	.598	.601	.604	.598	.597	.610	.607	.608
	Analogy	Skipgram	.301	.305	.299	.299	.298	.297	.295	.289	.300	.302	.301
		CBOW	.299	.296	.295	.29	.289	.289	.287	.288	.311	.314	.312

decreases as more GPUs are used, but within acceptable limits. For similarity based evaluation, using 8 GPUs give upto just 2% worse accuracy, but more than 3x speedup over 32 CPU threads.

For analogy based evaluation, the differences between multiple GPUs and CPUs are more conspicuous. In our experiments, our primary focus was on scalability and thus we did not increase the synchronization frequency when scaling across more GPUs. Increment in synchronization frequency can maintain a comparable accuracy, but will take a toll on scalability, leading to a sub-linear scaling. However, depending on the end application, one can decide the trade-off and tune the frequency accordingly.

6 CONCLUSION

In this work, we present BlazingText: a high performance distributed word2vec implementation that leverages massive parallelism provided by modern GPUs. We carefully exploit GPU parallelism to strike the right balance between throughput and accuracy. The proposed implementation achieves near linear scalability across multiple GPUs and can process tens of millions of words per second while maintaining a comparable accuracy with open source CPU based implementations like FastText. Our experiments on different datasets demonstrate good predictive performance and generalizations of our techniques.

As for future work, we plan to improve our model synchronization strategy and learning rate scheduling. As the vectors associated with more frequent words are updated more often, we want to match model synchronization frequency to word frequency, thus syncing the vectors of common words more frequently by using a sub-model synchronization scheme. For learning rate scheduling, we want to explore techniques like Adam [19], to improve the convergence rate when scaling across multiple GPUs.

REFERENCES

[1] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Neural and Information Processing System (NIPS)*, 2013.

[2] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[3] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. In *HLT-NAACL*, 2016.

[4] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[5] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[6] Simon Pavlik. Gensim word2vec on cpu faster than word2vec keras on gpu. https://rare-technologies.com/gensim-word2vec-on-cpu-faster-than-word2veckeras-on-gpu-incubator-student-blog/, 2016. Accessed: 2017-08-01.

[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[8] cublas: Dense linear algebra on gpus. https://developer.nvidia.com/cublas, 2015. Accessed: 2017-08-01.

[9] Spark mllib word2vec. https://spark.apache.org/docs/latest/mllib-feature-extraction.html, 2016. Accessed: 2017-08-01.

[10] Deeplearning4j: Introduction to word2vec. https://deeplearning4j.org/word2vec, 2015. Accessed: 2017-08-01.

[11] Shihao Ji, Nadathur Satish, Sheng Li, and Pradeep Dubey. Parallelizing word2vec in multi-core and many-core architectures. Nov 2016.

[12] Word2vec using tensorflow. https://github.com/tensorflow/models/tree/master/tutorials/embedding, 2015. Accessed: 2017-08-01.

[13] Word2vec-keras-in-gensim. https://github.com/niitsuma/word2vec-keras-in-gensim, 2016. Accessed: 2017-08-01.

[14] Icwsm 2011 spinn3r dataset. http://www.icwsm.org/2011/data.php. Accessed: 2017-08-01.

[15] Fast multi-gpu collectives with nccl. https://devblogs.nvidia.com/parallelforall/fast-multi-gpu-collectives-nccl/, 2017. Accessed: 2017-08-01.

[16] Text8 dataset. http://mattmahoney.net/dc/text8.zip. Accessed: 2017-08-01.

[17] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, and Phillipp Koehn. One billion word benchmark for measuring progress in statistical language modeling. *CoRR*, abs/1312.3005, 2013.

[18] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. Placing search in context: The concept revisited. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 406–414, New York, NY, USA, 2001. ACM.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.