Graphics Processing Units (GPUs): Architecture and Programming Final Project Report

Title: GPU implementation of Word2vec (Continuous Bag of Words) with negative sampling
Author: Pranav Dhar
Date: 12/13/2016
NYU-ID: N15699519

1 Abstract

The following report describes the GPU implementation of the Word2Vec algorithm et al [1] using the CBOW (Continuous Bag of Words) model with negative sampling. The primary objective is to investigate the parts of the algorithm that can be made GPU friendly and analyse the performance of the GPU implementation with that of the multicore version.

2 Introduction

Word2vec, specifically CBOW (Continuous Bag of Words) model is used to produce word embeddings from a corpus of textual data. This word embeddings represent the semantic relationship between context and target words within sentences that are generally obtained by computing their cosine similarity. This model is widely used in the realm of natural language processing and is quite helpful for use cases such as: sentiment analysis, recommendation engines and relationship mapping in social graphs etc. The word2vec paper includes various techniques/models such as hierarchical softmax, CBOW, negative sampling, clustering using k-means, down-sampling and various parameters which result in different types of word embeddings. In this paper we will focus on the Word2vec CBOW model and use Negative sampling and compare its performance of the to its implementation developed on the GPU.

3 Background Information

The Word2vec algorithm makes use of a shallow two layer neural network where the input layer consists of a one-hot encoded vector that represents the context word and the output layer similarly contains a target word.

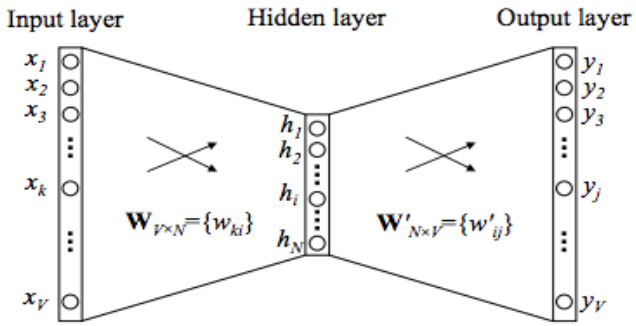

Figure 1: CBOW model with 1 word context and 1 word target.[3]

Figure 1 shows a simple CBOW model with 1 hot encoded vectors of context and target words. The steps in this particular algorithm involve reading in a large text corpus and create a vocabulary of unique words with a certain minimum occurrence which are then subsequently indexed. The words are represented as 1 hot encoded vectors which means that the index represented by a word in the vocabulary is set to 1 and all other elements are 0. The other main components are the weight matrices W and W' which are initialized to random values before the training begins. The Word2Vec CBOW algorithm can be broken down into the following steps:

1. If the context word is represented by the index K in the vocabulary, copy the corresponding row from the input weights matrix W into the hidden layer.
2. Propagate the hidden layer weights to the output layer by computing the inner product of the Jth column of the output weights, where J is the index of the target word.

3. Compute the errors for the target and the negative samples. Calculate the gradient and back propagate the values to the input layer by updating the input weights.

4 Literature Survey

In the word2vec et al [1] paper techniques such as CBOW are recommended for large datasets as it is much faster than Skip-gram. However the Skip-gram technique is regarded as being better for producing more accurate word embeddings for less frequent words. In another paper et al [2] various techniques are described to Parallelize the Word2vec algorithm by using mini-batching, by using multiple batches of context/target pairs resulting in fewer updates to global memory. In the original word2vec algorithm the Hogwild SGD update is made in parallel and any conflict are ignored, this approach is similar to the GPU implementation described below. The notion is that since the vectors are sparse it is unlikely that there will be collisions, however in et al [2] this approach is said to reduce the quality of the word embeddings generated.

5 Proposed Solution

In contrast to the multi-core implementation, the GPU implementation requires additional pre-processing of the text data before being sent transferred to the GPU's device memory. In the original word2vec algorithm the vocabulary is constructed in a sequential manner creating an indexed hash map of all the unique words with a certain minimum count. In the GPU implementation this hash table setup is similar and is performed on the host. However in the multi-core version the File I/O is distributed between n threads that read and train the CBOW model independently. Since the File I/O API is not accessible in the GPU kernel an array is created containing the context/target word pairs as indices and transferred into the device memory.

Subsequently the training of the model occurs in the GPU kernel where each thread handles a given context/target pair. The intermediate calculations, error/gradient vectors are saved in the local memory and are propagated back to the global memory in the final step of the thread. Also negative sampling is performed on the output weights by selecting a few random words assumed not to be in the context and adjusting the weights accordingly. Finally some of the conditional statements had to be removed to avoid branch divergence in the threads while being careful not to affect accuracy of the word embeddings.

A few techniques used in the multi-core implementation where also applicable for the GPU version, for example the exponents tables was precomputed to avoid using the exp() method which is computationally expensive. Similarly the next_random variable was given a psuedo-random value using the threadId multiplied with a large value(long long int) instead of using the rand() method.

Once all the threads complete running the output weight are copied back to the host and written into a word embeddings file in the binary format.

5.1 Experimental Setup

For running our experiment we used the following setup for the multicore CPU results: Intel Xeon E5-2680 (2.50 Ghz) (24 cores), and GeForce GTX TITAN Black (Compute capability 3.5 & total global memory: 6 GB) for the GPU results.

5.2 Experimental Results & Analysis

Following table contains information about time take by CPU & GPU for 15 iterations, negative samples = 25, window size = 1, hidden layer size = 5, alpha (learning rate) = 0.05:

Vocab Size	# Words	CPU - 10 threads (sec)	CPU - 20 threads (sec)	GPU Time (sec)
62	738	0.61	0.632	0.297
4294	174295	1.112	1.288	1.21
9096	496980	2.52	2.169	2.097
13172	741888	3.571	3.107	3.144
22323	1374002	5.995	5.394	5.816

Figure 2: Multicore CPU vs GPU

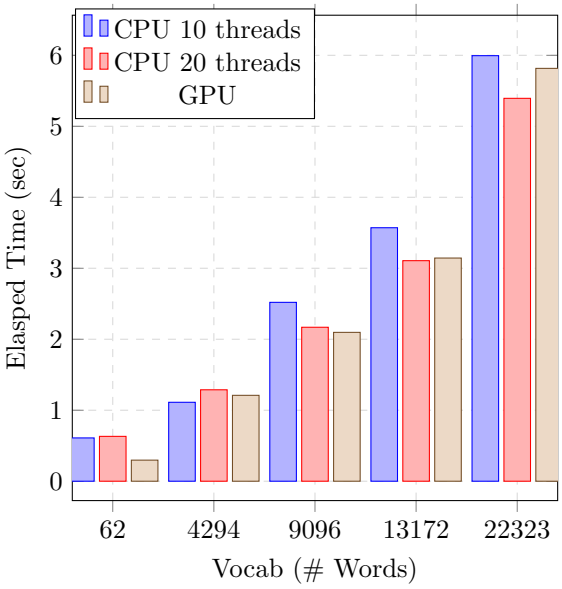

Figure 3: A figure

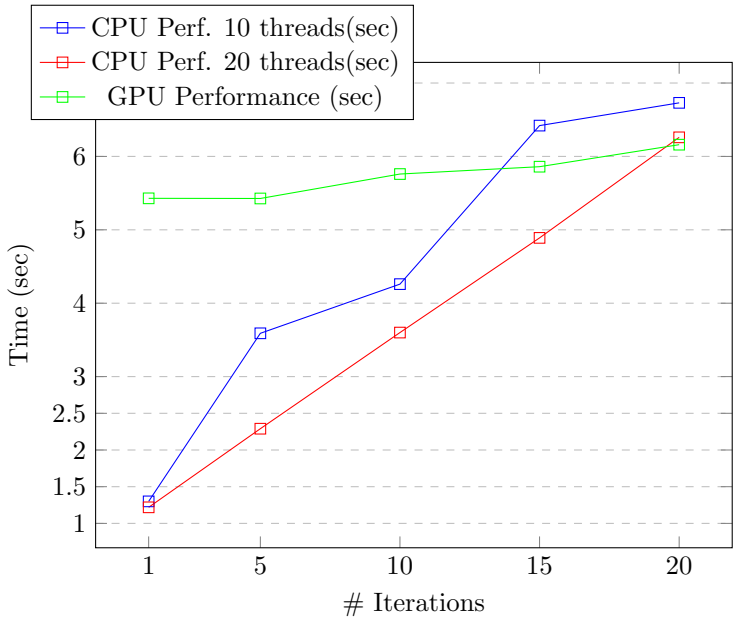

Figure 4: Iterations vs Performance (1.3M words, Vocab = 22323)

For the experiments to evaluate the performance of multi-core vs GPU Word2vec CBOW model we took the following parameters as constant values: negative samples = 25, window size = 1, hidden layer size = 5, alpha (learning rate) = 0.05. The models were then run on text corpora of various vocabulary sizes and word counts and their elapsed times were noted down. As we can see from Figure 3 the GPU implementation performs better than the CPU version running on 10 threads, while it performs worse than the version running on 20 threads. Also we can see from the number of iterations vs performance in Figure 4 that the GPU gives similar times as the number of iterations increase while the CPU version its increases linearly.

5.3 Conclusions

From the experiments conducted on the GPU implementation of Word2Vec CBOW model it is evident that this problem scales well on a GPU. The GPU also shows less performance degradation when the number of iterations is increased with respect to the CPU. There are additional optimizations that can yield a higher GPU performance by parallelizing the initial Vocabulary generation setup using multiple threads on the host. Also when analysing the word embeddings generated by the GPU version there is a loss of accuracy when compared to the multi-core version which could be caused by not grouping multiple context/target pairs per kernel thread. Further improvements could be made by increasing the context window size and adjusting the learning rate within the kernel. Another technique which would benefit the GPU implementation is to sub-sample the most frequent terms as described in the original word2vec paper.

References

[1] M. et al. Distributed representations of words and phrases and their compositionality, . URL https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-composi

[2] S. J. et al. Parallelizing word2vec in shared and distributed memory, . URL https://arxiv.org/pdf/1604.04661.pdf.

[3] X. Rong. word2vec parameter learning explained. URL https://arxiv.org/pdf/1411.2738v4.pdf.