# Parallel Implementation of Global Vector for Word Representation

E4750_2017Fall_VRFW_report
Zixiao Zhang zz2500, Di Zuo dz2357
*Columbia University*

## Abstract

*GloVe model can embed words into vector space, but the first stage of it, populating a word-word co-occurrence matrix, consumes much more time than other stages, which directly limits the efficiency of this model. Our project focused on accelerating this stage by implementing a parallelizable (by using GPU) version of it. We balanced the workload for each block in order to let GPU run efficiently for different sizes of inputs. And finally we can produce the same results as the CPU version but with dramatical reduction of time consumed.*

## 1. Overview

### 1.1 Problem in a Nutshell

Representing words with real-valued dense vectors rather than one-hot encoding can improve both the efficiency and accuracy of various natural language processing applications, such as information retrieval, named entity recognition, machine translation and so on. One of the most famous models, which can embed words into semantic vector space, is known as Global Vectors for Word Representation (GloVe) [1], proposed by Jeffrey Pennington and his colleagues in Stanford University. This model utilizes both the global information from word-word co-occurrence statistics and local information obtained by context window method.

In order to count the word-word co-occurrence, it needs to go through the entire corpus and check whether a target word is located within the context window of another target word, then add values at the corresponding position in the co-occurrence matrix. Generally, only when the size of corpus is large enough (in [1] the smallest corpus has size of 1 billion tokens), can we obtain adequate information to conduct accurate embeddings. On the other hand, iterating through a large corpus with only one thread will definitely take a great deal of time, which is far beyond the amount of time consumed by training this model, and this becomes the bottleneck for efficiently building the word vectors.

Our main goal is to implement a parallelizable version of this populating stage, which can execute on GPU by using CUDA. For achieving our goal, we are faced with several challenges including word indexing and performance concern. And we will illustrate our implementations in details in the following parts.

### 1.2 Prior Work

Jeffrey Pennington and his colleagues proposed GloVe model in [1], where they also analyzed the run-time of their implementation. They used a single thread of a dual 2.1GHz Intel Xeon E5-2658 machine, populating X with a 10 word symmetric context window, a 400,000 word vocabulary, and a 6 billion token corpus took about 85 minutes. Besides, they trained a model with vector size of 300 using above settings, which took 14 minutes for a single iteration. According to this comparison, it is obvious that accelerating the populating process is truly helpful in reducing time cost when dealing with much larger corpus.

## 2. Description

In this section, we go through the details of our GPU implementation of GloVe. First, we talk about the challenges we are faced with in handling documents; then we formulate the entire project and give step-by-step illustration; finally, software design architecture and pseudo code will be explicitly given at the end of this section.

### 2.1 Objectives and Technical Challenges

As mentioned before, our main goal is to implement a GPU version algorithm which can accelerating the process of populating the word-word co-occurrence matrix, and undoubtedly we also need to implement a CPU version as reference for both correctness and time comparison. After we obtain this co-occurrence matrix, we train the GloVe model and embed 10,000 frequent words into both 100 and 300 dimensional vector space. Last but not least, we do some tests to prove the word vectors we construct are useful in some situations. So, explicitly we have following four goals:

a) Implement GPU version of populating algorithm
b) Implement CPU version of populating algorithm
c) Build a GloVe model and train it with word-word co-occurrence matrix
d) Test the word vectors we construct

After setting up these goals, the most challenging part is text indexing when implementing GPU algorithm,

which means how to distribute different articles to different threads. Such task can be easily implemented without concern for performance. For example, make every thread deal with a fixed number of articles without caring about the total number of words processed every time. But this naive approach will lead to the divergence of time consumed by different threads, then constrain the overall efficiency of GPU code. So, we decided to balance the workload by evenly splitting whole corpora. On the other hand, word indexing is also something requires attention. For example, "for" is mapped to integer 79479 in our frequent word list, but then mapped to 10 in output matrix. Infrequent words are also mapped to certain indexes in frequent word list, but are all mapped to 0 and will not be counted in output matrix. If we ignore the special case in word indexing, we will easily add values in wrong positions in output matrix.

## 2.2 Problem Formulation and Design

The project mainly consists of four steps: preprocessing of original wiki text dataset, populating word-word co-occurrence matrix, training GloVe model and testing the results.

**2.2.1** Preprocessing

We need to tokenize text documents, that means eliminating symbols, turning uppercase letter to lowercase, and transforming each word to an integer index, then we need to count the frequency for each word and construct a frequent word list. There are several python tools we can use directly and we decided to use sklearn.

**2.2.2** Building word-word co-occurrence matrix

This is the most time consuming step. Since CUDA only supports 1-dimensional vector, indexing is a big problem. The rows and columns of this matrix both indicate the most frequent words we get during preprocessing. However, we can't directly use the original integer index for these words (some of them are larger than the largest index of the matrix), instead we mapped these integers in the range of [0, number of rows (columns) of the matrix) again, and use the secondary mapping results as the index for each frequent word. We feed the preprocessed documents as list of integers to GPU, use multiple threads to populate the matrix at the same time, meanwhile we also implement a python version and compare their results and running time.

**2.2.3** Training the GloVe model

After obtaining the word-word co-occurrence matrix, we train the GloVe model according to the following loss function:

$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

where X is the word-word co-occurrence matrix, $w_i$ and $w_j$ are our target word vectors, $b_i$ and $b_j$ are the biases [1] ; $f(x)$ here indicates a weight function, it has the form of

$$f(x) = \begin{cases} (x/x_{\max})^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases}.$$

and the optimizer we use is Adam optimizer.

**2.2.4** Testing

We conduct two kinds of tests: one is word similarity test, which is to test whether relative words are embedded close to each other in the vector space; the other is word analogy test, which is to answer questions like "a is to b as c is to __? ", the answer word "d" can be found according to following method [3]:

a) compute vector x by doing simple calculation

$$x = w_b - w_a + w_c$$

b) search the entire vector space, and the vector which is closest to x corresponds to the answer word "d".

In both tests, we use cosine distance as our measurement to indicate whether two vectors are close or not.

More detailed descriptions of above four steps can be found in 2.3.

## 2.3 Software Design

Bitbucket link:[https://bitbucket.org/zz2500/glove/overview](https://bitbucket.org/zz2500/glove/overview)

**2.3.1** Overall block diagram:

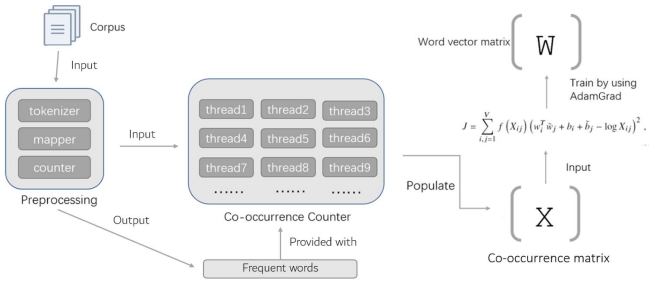


Figure 1 block diagram for this project

**2.3.2** Detailed description of each step

a) Preprocessing

We load all documents, tokenize them using CountVectorizer from sklearn, and write to csv file, where each line is a list of integers and represents one complete article. Then we take the most frequent 10,000 words and filter the tokenized corpora. For each word, if it is frequent, keep it unchanged; if not, change the corresponding index to 0. In GPU, when there is one '0', thread should skip it since it is infrequent.

b) Building word-word co-occurrence matrix.

One naive thought is that we let each thread deal with one article. This is not efficient since articles have varying lengths: some contain 15,000 words while others may contain only 50. Our way of balancing workload is that, let each block deal with similar number of words. One thing to understand is that the context of one target word must be limited within one article -- words in different articles are not always relevant. Therefore we must always treat one article in whole, making each block just have slightly different workload. Our actual approach is:

- Concatenate all texts to 1D array, store the length of each article, and do exclusive scan on the array of article lengths to find the index of first word of each article.
- Host sets a workload that is greater than the longest article and it is also multiple times of block size.
- Each block starts from the first article that starts after *block_index*workload*, and ends before the first article that starts after *(block_index+1)*workload*. Adopt same method of finding the start and end for every block , so that all of them do not overlap or miss any word.
- Assign number of blocks according to total length of words and workload.

This makes sure that all blocks have approximately, though not exactly, the same workload, and all articles and words are processed for exactly once.

Then in GPU, for the sake of coalesced memory access, we let all threads in one block start to look at the very beginning of each segment: thread 0 look at word 0, thread 1 look at word 1, etc. . Note that initially they do not necessarily look at the same article. After one loop, the word index they look at should increased by block size. Each thread should always keep track of the index of article it is currently looking at, and update it individually.

In one loop for each thread, they read the target word and the context: words before and after the target word within certain distance. In our approach, the distance is no more than 10. As mentioned before, we map the word integers to indices in the output matrix by using a device function, with 1 for most frequent and 9,999 for not-that-frequent (0 for all infrequent words), totally 10,000 frequent words. Whenever target word and context word are both frequent, we add 1/distance to the corresponding position in output matrix using atomic addition. If one of them is not frequent (not in the frequent words list), this case is skipped.

Indexing is indeed a trouble. We need to always take care of the index of words and index of articles, constantly checking whether indexes are within range. One major bug when coding was that we only updated index of article after each loop -- which actually should be updated both before and after each loop, since initially there may be more threads than the number of words in the first article, and these out-of-range threads will read the words in the second or even the third article.

**Pseudocode of kernel**

**Note:**

"The index of word": index in corpora[];

"The index of text": index in start_ptr[];

start_ptr[] contains the indexes of the first word of each text.

e.g. corpora[start_ptr[3]+10] is text 3, word 10, if corpora has at least 4 texts and text 3 has at least 11 words;

"Mapped index of word": Mapped from 0~260,000 to 0~9,999. Also index in output matrix

```
kernel function(corpora,    //All texts in 1-dimension
freq_word,          // List of frequent words
start_ptr,          //Index of first word of each text
workload,           //Workload of each block
num_of_texts,       //Number of texts
M_dim,              //Dimension of output, also number of frequent words
M )                 //Output

start_point = workload * blockIdx
find smallest i such that start_ptr[i] >= start_point          //i is the index of the first text this block will process
end_point = workload * (blockIdx+1)
find smallest j such that start_ptr[j] >= end_point            //j is the index of the first text NEXT block will process

if blockIdx+1 ==  gridDim:
        end_point = start_ptr[num_of_text]                     //Special case: last block
```

```
tx_ptr = threadIdx + start_point                    // Index of target word

Find the index of text that contains the word tx_ptr is pointing at

while tx_ptr < end_point:
        if corpora[tx_ptr] is not frequent word:
                tx_ptr += blockDim
                Find the index of text that contains the word tx_ptr is pointing at
                continue
        for each context word before and after the target word within the same text:
                if this word is not frequent:
                        tx_ptr += blockDim
                        Find the index of text that contains the word tx_ptr is pointing at
                        continue        //Or break if word after target
                atomicAdd 1/distance to corresponding position in M
        tx_ptr += blockDim
        Find the index of text that contains the word tx_ptr is pointing at
```

c) Model training

By training this model, it is only the nonzero values in matrix X that will be used, we randomly select elements from these nonzero values to fit into our loss function, and train two models with vector of 100 dimension and 300 dimension both for 100 iterations. The total amount of time cost when training is 1 hour 43 minutes. As for the optimizer, we choose learning rate to be 0.001, $\beta_1$ is equal to 0.9, $\beta_2$ is 0.999.

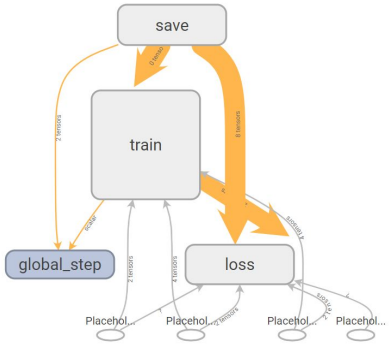


Figure 2 model graph in tensorboard

d) Testing

The first kind of test can be directly performed on tensorboard, which is a visualization tool for tensorflow (see part 3 for results). For the second kind of test, we randomly test several word pairs including semantic comparison as well as syntactic comparison, the results are also shown in part 3.

## 3. Results

We use various tools when finishing this project. In terms of hardware, the GPU we use to run our CUDA code is Tesla K40c, and CPU to run our python code is Core i5-7300U. As for software, we use python 2.7.12 and CUDA compilation tool 8.0.61 to run our kernel code, and use python 3.6.1 run CPU code. Besides, the version of tensorflow for building our model is 1.4.0.

### 3.1 Word-word co-occurrence matrix

The word-word co-occurrence matrix we finally obtained has size of 10,000 by 10,000. However, there are only 17,667,019 elements in it are nonzero, which is much less than the number of elements within the entire matrix.

We implemented both GPU and CPU version when populating this matrix, and verified their results. Verification is significant: we use GPU not because it can compute, but because it can compute faster. Python results and CUDA results are often different when doing computation on floating number [2]. How do I know the difference solely comes from CUDA rather the bug in the code?

Since floating number conforms to IEEE 754 and has 23 bits for significant figures and 8 bits for power of 10, we assumed that after 10^6 additions, 2% error rate was acceptable since 2^-23 * 10^6 =0.119. But we found another way to precisely determine whether our code is bug-free and CUDA is responsible for the difference: as is mentioned in 2.3, we add 1/distance to output matrix when one frequent word occurs in the context of another frequent word. If we add 1 regardless of the distance, Python and CUDA should provide exactly the same result, if there is no bug.

After adopting such integer algorithm and comparing, we got exactly same result from CUDA and Python version of word-word co-occurrence matrix,and the mean absolute error rate and max absolute error rate of float algorithm drop to 5.01e-8 and 1.59e-4 respectively.
In terms of time comparison, our GPU method only took 51.0199s in populating this matrix, however, the most straightforward CPU version took 22938s (around 6.4 hours), it is really a significant reduction of time (GPU/CPU = 1/449)!

## 3.2 training result

Figure 3 shows the training loss of both 100 dimensional model and 300 dimensional model

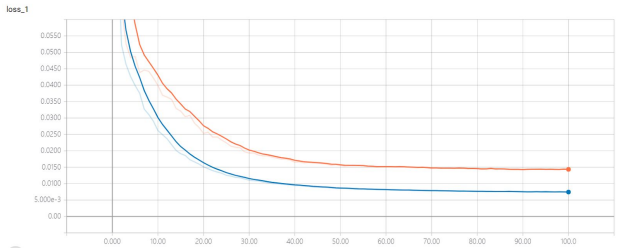


Figure 3 training loss for different model
orange line: 100 dimensional model  blue line: 300 dimensional model

From this graph, we can see when trained for the same time, 300 dimensional model always has lower loss than 100 dimensional one. This result is intuitively easy to understand: when there are more dimensions, we can obtain more information from the original data, thus the loss of higher dimensional model will be lower.

## 3.3 testing result

From figure 4 we can directly observe the result of the first kind of test. As we can see in the right side, those are the nearest 100 words, in the vector space, for our target word “paper”. These words such as “printed”, “newspaper”, “ink” and “plastic” are highly relevant to the target word, so they are closer to it compared with other words.

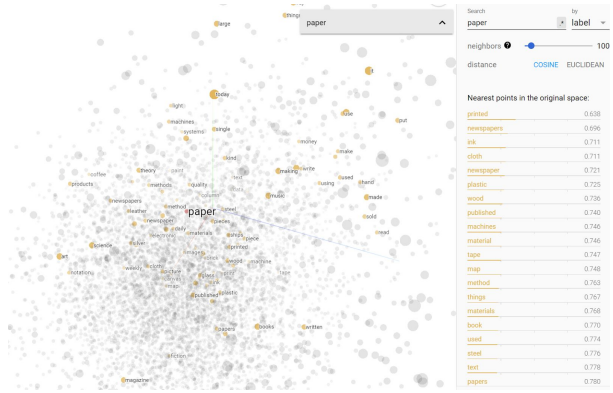


Figure 4 visualization of word vectors

As for the second kind of test, we randomly tested several words according to the methods mentioned above, and following are some results:

list 1 results for word analogy test

| | word “a” | word “b” | word “c” | word “d” |
|---|---|---|---|---|
| SEMANTIC TEST | china | beijing | france | berlin |
| | queen | king | girl | boy |
| | brother | sister | son | daughter |
| | good | reward | bad | punishment |
| SYNTACTIC TEST | thinking | think | do | does |
| | artist | art | scientist | science |
| | dollar | dollars | friend | friends |
| | greater | great | better | good |
| | said | saying | done | doing |

From these results we can conclude the word vectors we get do encode the information in the original text.

## 4. Demonstration

Raw document:

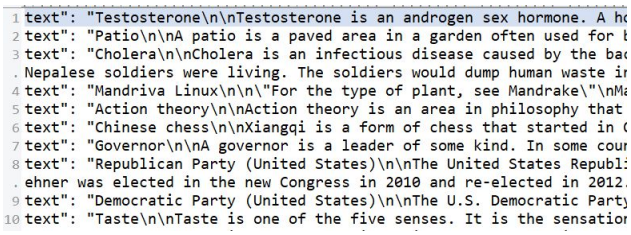


Figure 5 raw document

Tokenized document:
Each line is one text. 0 means infrequent words.

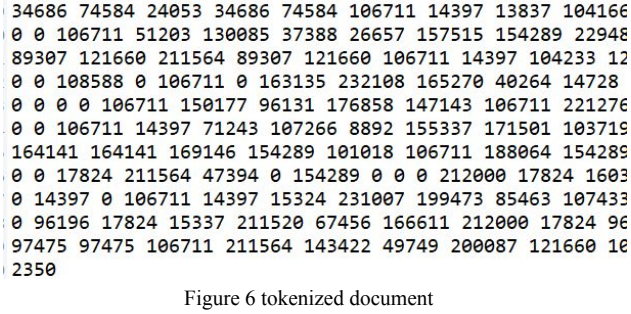


Figure 6 tokenized document

Frequent words:
UNK: unknown infrequent words

```
UNK 0
the 211564
of 154289
in 103719
and 14728
is 106711
to 214004
was 229078
it 107266
he 94900
for 79479
on 155337
as 19128
are 17824
that 211520
by 37388
```

```
option 155955
cork 52075
tactics 207386
cycles 55890
puzzle 173052
cherry 44691
urine 222243
drake 65085
indicated 104149
reunited 180017
phd 164300
pulmonary 172494
outbreak 157528
sequels 191233
```

Figure 7 word integer pairs

Output Matrix (10,000 by 10,000):

```
0.0000000000000000000e+00,3.5551333007812500000e+03,3.9375786132812500000e+03,2.404792
+01,3.5000396728515625000e+01,5.9230160713195800780e+00,2.2237695693969726560e+01,8.95
719e+01,1.1666666269302368160e+00,3.5489681243896484380e+01,1.7641664505004882810e+01,
2780762e+00,4.9298416137695312500e+01,8.0059518814086914060e+00,2.0638887882232666020e
94482421875e+01,1.7440082550048828120e+01,1.2007934570312500000e+01,1.8395236968994140
000000000000000e+00,1.6042854309082031250e+01,1.1846825599670410160e+01,1.1974204063439
.9851190567016601560e+01,0.0000000000000000000e+00,1.4010317802429199220e+01,1.6833333
00,3.9134924411773681640e+00,0.0000000000000000000e+00,3.9150793552398681640e+00,1.886
25e+01,1.0084127426147460940e+01,1.6861110925674438480e+00,3.6111110448837280270e-01,1
476562e+00,6.9301581382751464840e+00,1.8053571701049804690e+01,1.6703174591064453120e+
9802322388e-01,2.1027777194976806640e+00,1.0946428298950195310e+01,5.6194443702697753
7390899658203120e+00,1.8916667699813842770e+00,2.0000000298023223880e-01,1.6750000715255
9884920120239257810e+00,1.5123016834259033200e+00,2.3499999046325683590e+00,5.5289678573
```

Figure 8 part of word-word co-occurrence matrix

## 5. Discussion and Further Work

Our kernel can be further optimized by putting arrays frequently used into shared memory. We can simply copy the frequent word list to the shared memory since it is traversed every time finding the index of a given word. In our approach, we choose 10,000 frequent words which is far less than practical use. It contains 10,000 32-bit integers and will take up 40,000 bytes in total, whereas the GPU provides about 49,000 bytes shared memory.

In our 10,000 version, the kernel takes about 51 seconds:

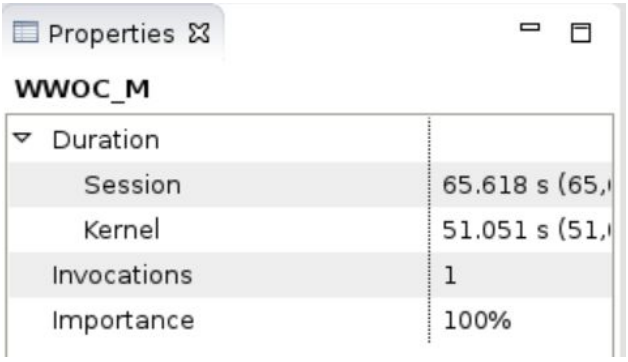


Figure 9 nvvp profiler result for 10,000 by 10,000 matrix

We also tested for 20,000 frequent words. With double frequent words, the output matrix became 4 times larger, but the running time only doubled:

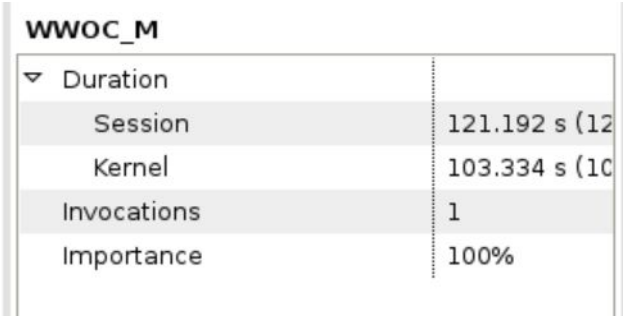


Figure 10 nvvp profiler result for 20,000 by 20,000 matrix

We can infer that finding the mapped index of given word is the most time-consuming part in kernel. Therefore, we suppose using shared memory to store frequent words can significantly save the time but will require GPU to provide enough shared memory.

Besides, based on the trained word vectors, we can actually apply them into some practical tasks such as documentation classification, named entity recognition and so on. These are also fine ways to test the accuracy of our results.

## 6. Conclusion

In this project we implemented Global Vector model for word representation in Python and tensorflow, and greatly increased the step of building word-word co-occurrence matrix using CUDA. We also implemented this step in Python and compared their results to prove that CUDA's output is correct. With this model we can find correlated words, and answer semantic and syntactic questions.

## 7. References

[1] Pennington, Jeffrey, Richard Socher, and Christopher Manning. "Glove: Global vectors for word representation." Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014.

[2] Floating Point and IEEE 754 Compliance for NVIDIA GPUs
http://docs.nvidia.com/cuda/floating-point/index.html

[3] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. Efficient Estimation of Word Representations in Vector Space. In ICLR Workshop Papers.

[4] Remi Lebret and Ronan Collobert. 2014. Word embeddings through Hellinger PCA. In EACL.