Word Embedding Models Evaluation and Word Inference Accelerator

By

Zizhen Liu

Senior Thesis in Computer Engineering

University of Illinois at Urbana-Champaign

Advisor: Wen-Mei Hwu

December 2018

Abstract

Many word2vec algorithms have been developed to represent vocabularies with more accurate number vectors. The most popular ones to date are word2vec [2], FastText [1] and GloVe [3]. Despite their popularity, there has been a lack of published work that helps new users to understand how models trained by different algorithms are performing on words from different categories. Additionally, despite the great work being done in implementing word2vec training algorithms on GPUs [4], there has been a lack of published work in implementing word2vec inferencing algorithm on GPUs. Therefore, this thesis will first present my research on inference accuracy of models trained by word2vec, FastText and GloVe across multiple categories of datasets and explain why we see such differences in prediction accuracy. Secondly, the thesis will present an algorithm for performing batched word analogy queries running on GPUs, that can adapt to different models trained by any algorithm and thus can significantly reduce the efforts required in comparing models trained by different word vector generation algorithms.

Subject Keywords: NLP, GPU, Word Embedding

Contents

1. Introduction 1
2. Word Embedding Models 2
2.1 GloVe 2
2.2 Word2Vec Skip-gram 2
2.3 FastText Skip-gram 3
3. Experiments 4
3.1 Setup 4
3.2 Results 5
4. Word Inferencing Accelerator 8
4.1 Word Similarity 8
4.2 Model Loading API 8
4.3 Word Analogy Queries 9
4.4 Experiment Setup 10
4.5 Discussion 11
5. Conclusion 12
References 13
Appendix A: Accuracy Measurements 14
Appendix B: Accelerator Timing Measurements 16

1. Introduction

Representing words with high dimension vectors, also known as word2vec or word embedding, is one of the keys to grasp the meaning of words in natural language processing and a founding stone to future breakthrough in artificial intelligence. In the industry, there exist multiple word2vec models in the community: word2vec FastText and GloVe. Most of these models produce a high dimensional, length customizable vector representation of each word in a vocabulary bag, pre-generated from the text corpus. These models can be used to perform meaningful operations such as topic modeling, word similarities and word analogy. Even though a well-trained model has been demonstrated to accurately extract the gist of a given sentence, the models become much less effective when it comes to practical deployment.

One reason for such deficiency is the lack of systematic comparisons between different models based on well-designed experiments. Performing such comparison by hand is also highly inefficient and unrepeatable because models often use different training and inference APIs. This thesis has investigated three of the state-of-art word2vec models in the industry: GloVe, word2vec and FastText, each representing a different approach at the word to vector problem. It will first go over the design differences. Then it will elaborate on how the experiment is set up and provide comparison results on prediction accuracy using multiple data sets. This thesis will also introduce the architecture of an accelerated word inference tool developed in CUDA using trending libraries that improve the matrix vector multiplication and model loading performance.

2. Word Embedding Models

There are currently two standards that are developed based on different approaches that attempt to solve the word embedding problem. They are frequency-based modeling and prediction-based modeling.

In frequency-based modeling, the word vectors are generated according to the occurrence of each word in a text corpus. The approach is preferred for text with less correlated words (such as list of names or jobs) and can be adapted to co-occurrence matrix with each row representing a different word and each column representing the count of occurrence of that word in every context. The frequency-based modeling approach had been believed to be cheaper to compute than prediction-based modeling but with a drawback on deriving logical relationships between words. It is less effective for tasks such as word analogy and word similarities, but more effective for topic modeling. However, an empirical result [4] shows that GloVe shares similar performance with prediction-based modeling.

Prediction-based modeling is another approach which gained its popularity after the introduction of word2vec model [2]. The approach came in two architectures: Continuous-bag-of-words (CBOW) and skip-gram (SG). CBOW predicts a word from a window of context words and SG predicts context based on input word. The two architectures are both able to perform word analogy task, but this thesis will focus on skip-gram architecture because it does a better job for infrequent words according to the author's note [2]. This thesis will also take FastText skip-gram modeling, a slight modification of SG, into account and its key difference will be explained in the following sections.

2.1 GloVe

Abbreviated from "global vector", GloVe is the state-of-art frequency-based modeling method. GloVe is trained by co-occurrence matrices extracted from unit of context in text corpus. The most common units might be sentences or paragraphs. The matrix records the word to word co-occurrence probabilities in a context and hope that the probability will resemble the relationship between words. GloVe is also not predicting context words based on an input word; rather, it is trying to reproduce another co-occurrence matrix given an input word and perform prediction based on the similarities between the reconstructed co-occurrence matrix and the matrix of other words in vocabulary list.

2.2 Word2Vec Skip-gram

Skip-gram is a model introduced by Mikolov in his thesis in 2012. The model is constructed and performed mostly like common neural network models. The objective of skip-gram is to predict the

context given a word, and the training is done with <word, context word> pairs extracted from the text corpus. The model is faster than previously published neural word embedding models due to its single hidden layer architecture. The cost function is computed based on dot product between two word-vectors. This is because the dot product of two vectors is the length of projection from one vector to another and is an indicator of how similar the two vectors are.

2.3 FastText Skip-gram

FastText is an extension of word2vec model. Instead of treating each word as an entity, FastText breaks each word into a set of all n-grams made up by its characters. For example, “red” will be a set of n-grams {r, re, red, e, ed, d}. A simple twist in structure like this introduces several pros and cons:

Word embedding in FastText for rare words is more accurate because rare words have fewer context words in GloVe or word2vec but in FastText, its n-grams will be shared with many other words. Thus, even if the word does not exist in the text corpus as a single word, we can still query the word’s nearest neighbor reasonably accurately. [1] As the size of text corpus grows, the memory and time usage also accelerate and therefore controlling on minimal word occurrence is necessary.

3. Experiments

3.1 Setup

The following experiment is performed with controlled variables to give a benchmark on three models' behavior on different tasks given the same training text corpus. The text corpus, text8.zip, is the state of art training data for word2vec related researches.

The prediction data sets used in the following experiment are from Google's initial word2vec repository at code.google.com/archive/p/word2vec/ because the group provided high quality test vocabularies from multiple common topics at scale. The prediction data is arranged in a matrix where each line contains words in the order of A B C D where A-B+D=C. The topics of prediction data is as follows and will only be represented using index in the following tables:

1. capital common countries
2. capital world
3. currency
4. city-in-state
5. family
6. gram1-adjective-to-adverb
7. gram2-opposite
8. gram3-comparative
9. gram4-superlative
10. gram5-present-participle
11. gram6-nationality-adjective
12. gram7-past-tense
13. gram8-plural
14. gram9-plural-verbs

The logic behind prediction is that the vector subtraction between A and B will resemble the vector subtraction between C and D, an example would be king - man = queen - woman. Thus A-B+D will resemble C if the model is reasonably trained.

Then a nearest neighbor algorithm is run by first computing the dot product between A-B+D and C, i.e., calculating the projection of arbitrary C's vector onto A-B+D's vector. The C with the highest dot product

will be the most likely prediction, but the models usually allow the freedom to output arbitrary number of top most likely predictions.

The following results' prediction absolute accuracy rate will be based on the most likely prediction result, and the almost-accurate rate will be based on the number of predictions where the correct result falls in the top ten most likely predicted outputs (including the most likely predicted output). Figure 1 will present the Absolute Accuracy and the Almost Correct Rate of three models when evaluated on all datasets as an atomic entity. Figure 2 will present the accuracy and almost accurate rate of different models and different vector size across individual datasets.

The data that supports the following graphs will be available in Appendix A

3.2 Results

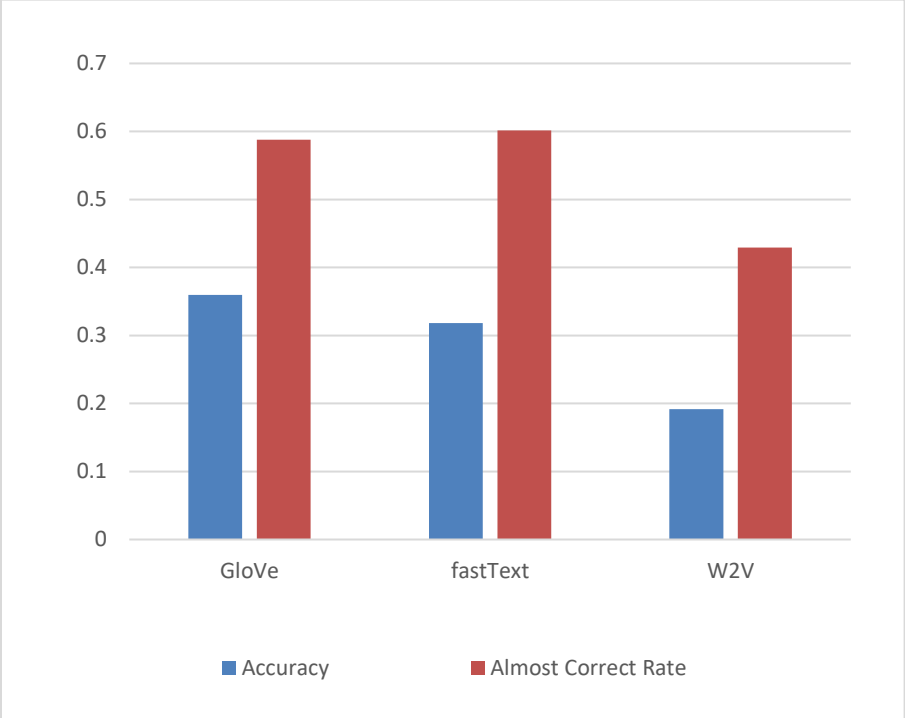


Figure 1: Average accuracy chart when vector size = 200

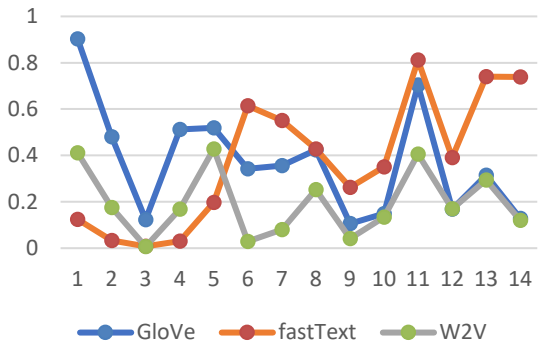
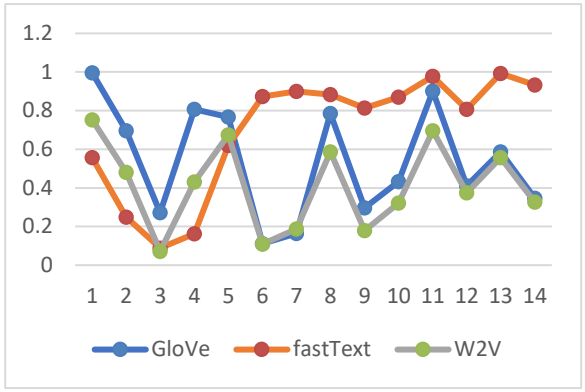
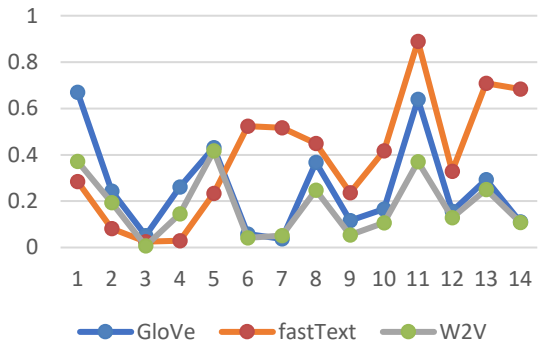
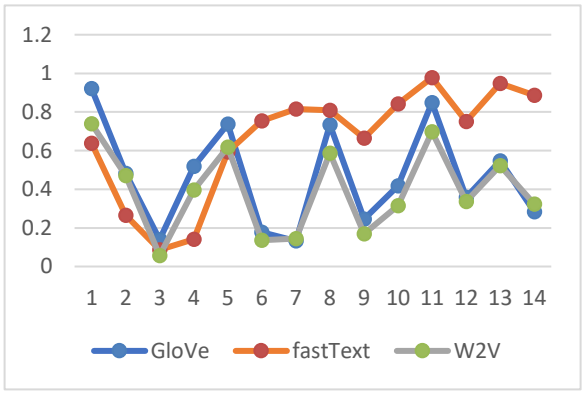
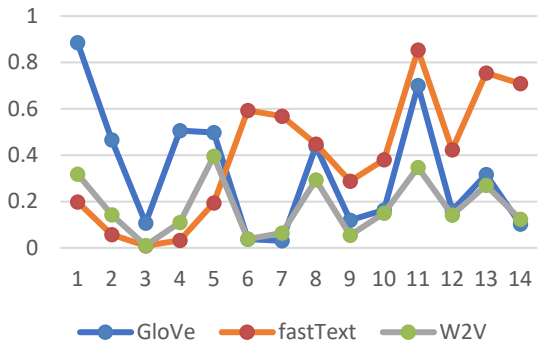
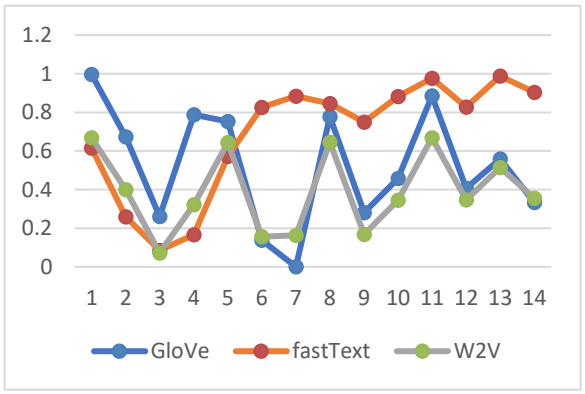
Vector size	Accuracy Rate	Almost Accurate Rate
200		
150		
100		

Figure 2: Accuracy plots among different models on datasets at three vector sizes

It is observed that GloVe and word2vec are performing relatively well on predicting semantic words (the first five datasets) such as nouns and fastText is doing relatively well on predicting syntactic vocabularies.

This result is coherent with conclusions made in prior work papers. It is also consistent with how the three algorithms are implemented. Because FastText is an n-gram SG training algorithm, words that share a common sub-string will be strongly correlated. This gives FastText natural strengths at predicting syntactic vocabularies like adjectives to adverbs or adjectives to comparatives, here are some examples:

1. **amazing amazing**ly **apparent apparent**ly
2. **obvious obvious**ly **quick quick**ly
3. **big big**ger **bright bright**er

Glove's training is based on the number of occurrence of a given word in fixed length context, and general word2vec training is based on the relative positioning of a word to its neighbors. They are behaving relatively better on the semantic vocabularies because words that are related to similar topic tend to occur more in the same context and in a predictable pattern.

The distinct difference in performance of the three models can potentially be used to better separate the members of a sentence and better understand the syntactical structure of a sentence.

4. Word Inferencing Accelerator

The state-of-art word embedding library to-date is Python's Gensim, which covers almost any aspect of word embedding application or research. Gensim is also extremely fast despite that its APIs are written in Python. The underlying matrix multiplication efficiently utilizes multicore CPUs. This section of the thesis explores the possibility of implementing an inferencing algorithm running on GPU.

This section will first introduce how word inference/word similarity is done. Then it will explain the architecture of the inference accelerator for both single query and batched queries written in CUDA and present with data of the time consumption on predicting size-varying datasets.

4.1 Word Similarity

The degree of similarity is represented by the scalar products of reference vector and vectors in the dictionary. The more similar the two words are, the closer the scalar product is to 1. Following is an example to illustrate how word similarity helps with predicting word analogy results:

Query triplet (A - B + C) = ? **berlin germany france**

paris 0.896462

bourges 0.768954 (another city in France)

louveciennes 0.765569 (a commune in France)

… (more words)

4.2 Model Loading API

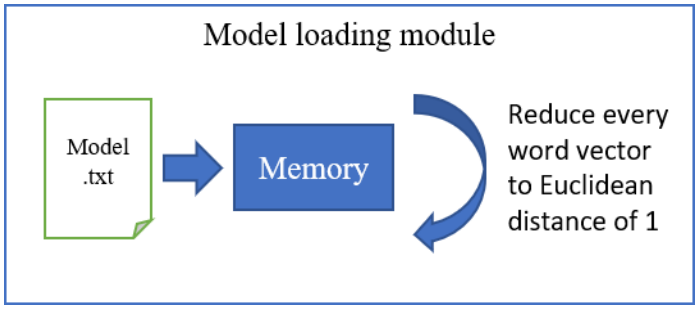


Figure 3: High level architecture on model loading and normalization

Figure 3 shows the high-level view of the model loading API, which includes code that loads words and their corresponding vectors into memory in a dictionary data structure and a function that normalizes the

vectors so that every word vector will have an Euclidean distance of 1 in a high dimensioned space. For example, a word vector of [a, b, c, d] will be adjusted to

$$\left[\frac{a}{x}, \frac{b}{x}, \frac{c}{x}, \frac{d}{x}\right] \quad | \quad (x = \sqrt{a^2 + b^2 + c^2 + d^2})$$

4.3 Word Analogy Queries

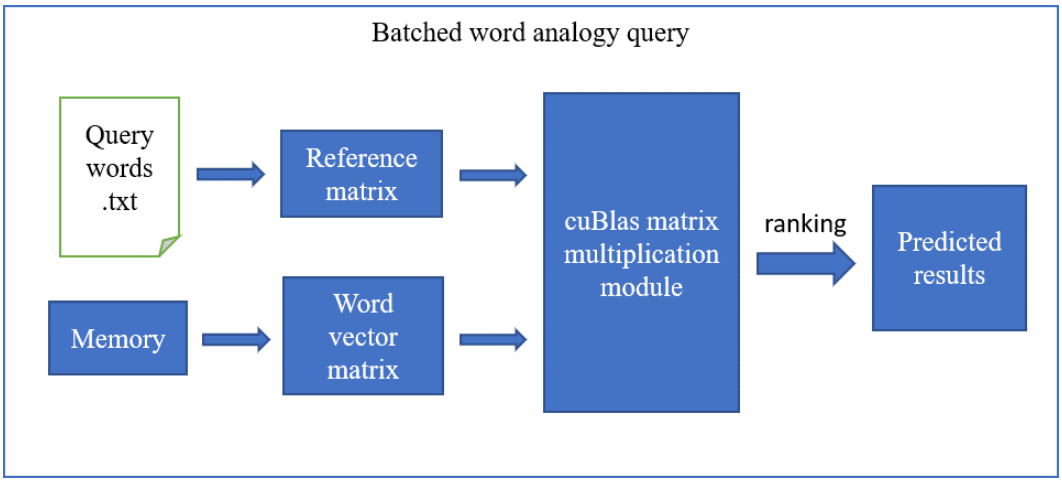


Figure 4: High level architecture on word analogy queries

Figure 4 shows the high-level architecture of the design and the data flow when dealing with batched word analogy queries. The design includes a function that computes the addition and subtraction of input word vectors to get the reference vectors:

$$reference\ vector = dict[king] + dict[woman] - dict[man]$$

The reference vector will then be passed to matrix vector multiplication module with the entire vector space for word similarity checking and finding target word.

The purpose of this module is to compute the scalar product between reference vector and all vectors in the dictionary. The result can be used to predict the most similar word to the reference vector in the dictionary.

The matrix multiplication module is done with kernel call in cuBlas library. The design does a single 2D matrix-matrix multiplication kernel call for batched queries of any lengths which saves time compared with doing vector-matrix multiplication for every query. The end-to-end performance break down will be presented in the next section.

4.4 Experiment Setup

The two most time-consuming kernel activities that will be significantly influenced by change of query count are:

1. CUDA memcpy DtoH
2. matrix multiplication

The pretrained model used for this experiment is trained by GloVe. The dimension per vector is 150 and total vocabulary count is 71291.

Queries of different size from 2000 to 20000 will be applied, and the time consumption growth pattern can be observed in Figures 5-7. The exact data supporting the diagrams is available in Appendix B.

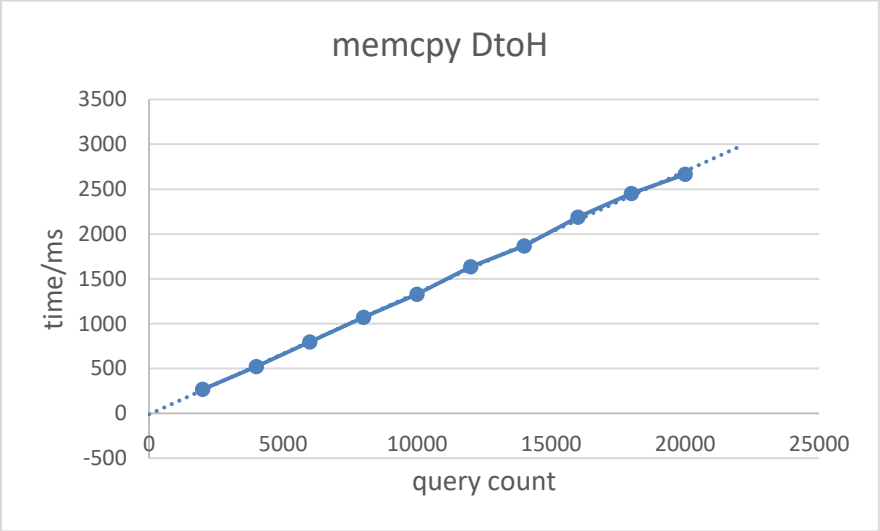


Figure 5: Memcpy DtoH time consumption

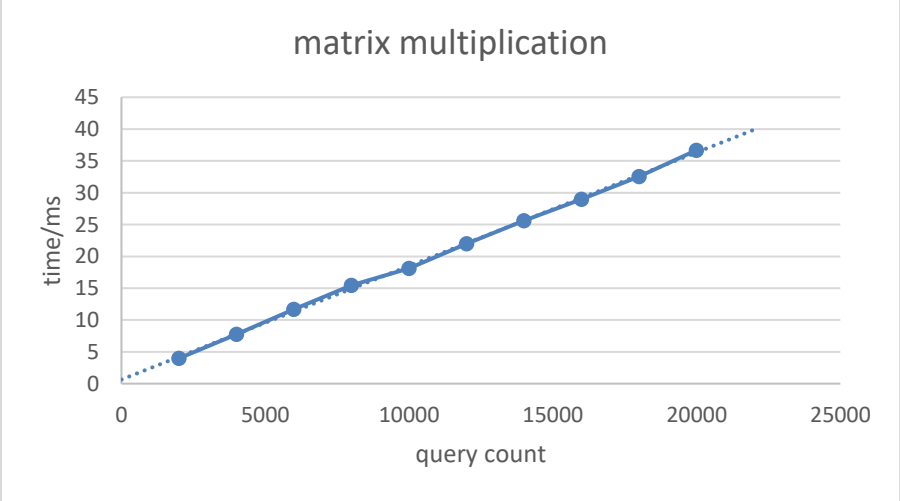


Figure 6: Matrix multiplication time consumption

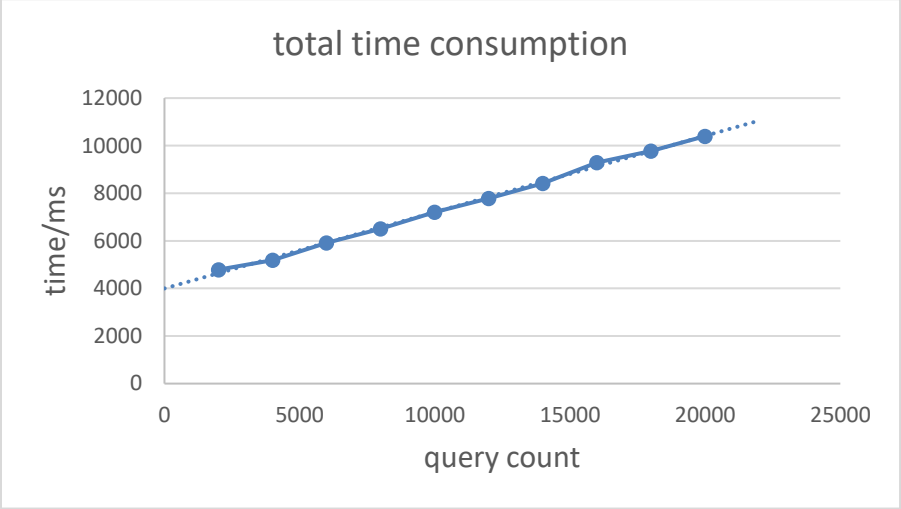


Figure 7: Total time consumption

4.5 Discussion

From Figure 5-7, it can be observed that the inference accelerator can handle query counts in linearly increasing time as query count grows. Total time consumption of entire API call has an offset of 4000ms when no query is being done. The majority components of the offset are CUDA kernel set up time and model loading time, which includes loading model from text file to CPU memory, and mem-copy from host to device.

The stride length between each consecutive test cases is 2000 lines of query. The GPU's (Nvidia TITAN V) memory is not enough to compute any query count starting from 22000 in one round. One future project is to enable active analysis of GPU memory usage and actively splitting batched queries into groups and finish in multiple rounds.

5. Conclusion

This thesis first conducted an experiment on three trending word2vec models to compare their performance on test sets grouped by topics. The result was that fastText model is working better on syntactic vocabularies and word2vec/GloVe is working better on semantic vocabularies. We then explained our observation based on the difference between designs of three algorithms.

Then this thesis proposed an inferencing tool written in CUDA and presented an experiment to investigate the time consumption of individual kernel calls and the entire API call. The design can handle query count of up-to 20000 with a pretrained model of dimension 150 and vocabulary size of 71291. Larger query size will result in cublasSgemm kernel's failure.

As for future work, we can rewrite part of the inferencing tool to support the functionality of keeping the model in GPU memory instead of loading it for every query. Secondly, we can make the program to actively split batched queries into groups so that the program will not fail on extra-large batched query sets.

The inferencing algorithm is only used on word analogy for this thesis. We can explore its potential usage on more complex and interesting applications like doc2vec or sentence2vec. The algorithm can also be integrated into the C3SR team's MLmodelscope website's backend to provide fast query service to researchers working on word2vec algorithms or students learning word2vec models.

References

[1] Joulin, A.; Douze, M.; J´egou, H.; and Mikolov, T, *Fasttext.zip: Compressing text classification models*, 2016.

[2] Mikolov, T. and Geoffrey Z., *Context Dependent Recurrent Neural Network Language Model*, 2012 IEEE Spoken Language Technology Workshop (SLT), 2012.

[3] Pennington, Jeffrey, et al. *Glove: Global Vectors for Word Representation.* Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014.

[4] T. M. Simonton and G. Alaghband, *Efficient and accurate Word2Vec implementations in GPU and shared-memory multicore architectures*, 2017 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, 2017.

Appendix A: Accuracy Measurements

Prediction Data set #	Vector size = 100		Vector size = 150		Vector size = 200	
	Accuracy	Almost Correct Rate	Accuracy	Almost Correct Rate	Accuracy	Almost Correct Rate
1	0.6700	0.9209	0.8854	0.9960	0.9032	0.9960
2	0.2430	0.4823	0.4655	0.6740	0.4801	0.6961
3	0.0520	0.1393	0.1074	0.2601	0.1225	0.2718
4	0.2609	0.5197	0.5064	0.7884	0.5120	0.8064
5	0.4310	0.7381	0.4976	0.7524	0.5190	0.7667
6	0.0585	0.1784	0.0373	0.1381	0.343	0.1119
7	0.0370	0.1322	0.0304	0.1402	0.357	0.1640
8	0.3671	0.7335	0.4384	0.7778	0.4242	0.7860
9	0.1159	0.2460	0.1190	0.2813	0.1058	0.2963
10	0.1667	0.4176	0.1638	0.4574	0.1496	0.4328
11	0.6397	0.8481	0.7009	0.8849	0.7048	0.9007
12	0.1558	0.3590	0.1609	0.4058	0.1692	0.4109
13	0.2935	0.5473	0.3168	0.5586	0.3153	0.5863
14	0.1103	0.2839	0.1023	0.3345	0.1287	0.3471
Average	0.2577	0.4798	0.3547	0.5760	0.3595	0.5876

Table 1: Accuracy measurements for model trained by GloVe

Prediction Data #	Vector size = 100		Vector size = 150		Vector size = 200	
	Accuracy	Almost Correct Rate	Accuracy	Almost Correct Rate	Accuracy	Almost Correct Rate
1	0.2845	0.6382	0.1976	0.6145	0.1245	0.5573
2	0.0820	0.2654	0.0572	0.2592	0.0333	0.2479
3	0.0254	0.0866	0.0092	0.0854	0.0080	0.0888
4	0.0299	0.1417	0.0324	0.1669	0.0312	0.1617
5	0.2332	0.5928	0.1936	0.573	0.1976	0.6205
6	0.5241	0.7549	0.5927	0.8265	0.6139	0.8739
7	0.5172	0.8152	0.5677	0.8829	0.5504	0.8989
8	0.4489	0.8085	0.4481	0.8452	0.4286	0.8835
9	0.2370	0.6648	0.2878	0.7503	0.2620	0.8136
10	0.4176	0.8427	0.3797	0.8815	0.3503	0.8692
11	0.8899	0.978	0.8542	0.9761	0.8117	0.9768
12	0.3294	0.7511	0.4230	0.8274	0.3903	0.8076
13	0.7087	0.9481	0.7545	0.9879	0.7394	0.9924
14	0.6839	0.8873	0.7091	0.9033	0.7390	0.9321
Average	0.3300	0.5701	0.3358	0.596	0.3184	0.6013

Table 2: Accuracy measurements for model trained by FastText Skip-gram

Prediction Data #	Vector size = 100		Vector size = 150		Vector size = 200	
	Accuracy	Almost Correct Rate	Accuracy	Almost Correct Rate	Accuracy	Almost Correct Rate
1	0.3181	0.6680	0.3715	0.7391	0.4111	0.7530
2	0.1427	0.4001	0.1933	0.4707	0.1755	0.4808
3	0.0101	0.0721	0.0067	0.0570	0.0067	0.0721
4	0.1095	0.3214	0.1457	0.3961	0.1691	0.4314
5	0.3952	0.6429	0.4167	0.6190	0.4285	0.6738
6	0.0383	0.1563	0.0413	0.1371	0.0292	0.1099
7	0.0641	0.1638	0.0513	0.1453	0.0797	0.1880
8	0.2935	0.6449	0.2477	0.5871	0.2523	0.5863
9	0.0548	0.1677	0.0538	0.1699	0.0409	0.1785
10	0.1496	0.3447	0.1070	0.3144	0.1335	0.3210
11	0.3465	0.6680	0.3702	0.6976	0.4057	0.6963
12	0.1417	0.3468	0.1282	0.3360	0.1694	0.3745
13	0.2698	0.5151	0.2507	0.5230	0.2937	0.5563
14	0.1241	0.3552	0.1080	0.3241	0.1206	0.3264
Average	0.1698	0.3973	0.1783	0.4140	0.1915	0.4294

Table 3: Accuracy measurements for model trained by Word2Vec Skip-gram

Appendix B: Accelerator Timing Measurements

Query count	CUDA memcpy DtoH	Matrix Multiplication	Total
2000	267.25ms	3.9783ms	4791 ms
4000	520.94ms	7.7440ms	5182 ms
6000	797.90ms	11.662ms	5917 ms
8000	1.07288s	15.464ms	6510 ms
10000	1.32859s	18.093ms	7204 ms
12000	1.63519s	21.982ms	7778 ms
14000	1.86701s	25.599ms	8411 ms
16000	2.18862s	28.977ms	9297 ms
18000	2.45041s	32.532ms	9779 ms
20000	2.66481s	36.690ms	10400 ms

Table 4: Accelerator timing measurements