Paragraph Vector Implementation using CUDA

Darshan Hegde
Center for Data Science
New York University
e-mail: darshan.hegde@cims.nyu.edu

November 26, 2014

Supervisor	Prof. Mohamed Zahran
Course	CSCI-GA.3033-004: Graphics Processing Units (GPUs): Architecture and Programming

1 Introduction

Representing sentence / paragraph / document (refer to as paragraph) in a continuous vector space so that, semantically similar paragraphs are close to each other is one of the fundamentals problems in Natural Language Processing(NLP). For example, "the movie isn't worth much of my time" is close to "the rest of the movie was total crap", than "rating it 8 out of 10 ." These representations are called Distributed Representations (DR) in Natural Language Processing (NLP) / Deep Learning community. Typically, distributed representations are learnt using huge unsupervised corpus. They can be tuned later for supervised learning task at hand. Paragraph Vector [1] is a state of the art technique to find distributed representations of text of arbitrary length, which was introduced recently [June 2014]. However, this algorithm is computationally expensive for a large corpus. In this project, we propose to efficiently implement Paragraph Vector [1] using GPU.

We will be implementing Paragraph Vector Distributed Bag of Words (PV-DBOW) model presented in Paragraph Vector [1] which is illustrated in Fig 1. As the name suggests, it finds the representation that's invariant to word order. It also expects the Distributed Representations for words are already found. Typically, word distributed representations are got using Word2Vec [3]. Also, it uses the hierarchical softmax parameters and throws away word representations. It initializes the paragraph vector as random vector of size equal to that of word representations. Then using the present representation, it tries to predict a sampled word from the paragraph, propagates the error of prediction back and updates the present representation. The algorithm proceeds iteratively, until prediction error (or difference between iterates) is smaller than pre-specified threshold. This iterative update is independent for different paragraphs and can be run in parallel. So, it's suitable for massively parallel GPUs.

2 Design

2.1 Mathematical Formulation

Let's first formalize the PV-DBOW model, before we proceed to nitty-gritty's of GPU implementation. The Mathematical notations follows Word2Vec [4] for the most part. As shown in Fig 1, the task is to find PV such that, it predicts the vector representation for words in that paragraph well. To find such a vector, we

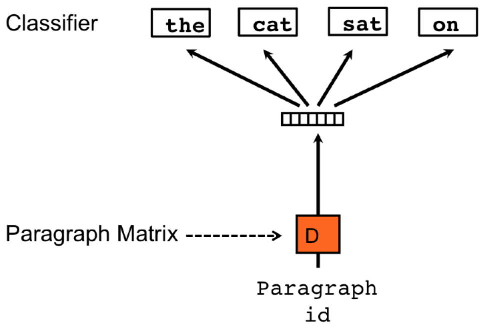

Figure 1: Paragraph Vector Distributed Bag of Words (PV-DBOW) Model. Image Courtesy: Paragraph Vector [1]

define a cost function (Negative Log Likelihood), which is low if our PV predicts the words well. p^*_{vk} is the minimal value for the optimization problem and will be the PV for k^{th} paragraph.

$$NLL(p_{vk}, P_k) = -\sum_{w_{ik} \in P_k} \log p(w_{ik}/p_{vk}) \tag{1}$$

$$p^*_{vk} = \arg\min_{p_{vk}} NLL(p_{vk}, P_k) \tag{2}$$

Where $P_k = \{w_{1k}, w_{2k}, \ldots w_{nk}\}$, set of words in the k^{th} paragraph. p_{vk} is PV for Paragraph k. We define the probabilities $p(w_{ik}/p_{vk})$ using the soft-max function.

$$\hat{p}(w_{ik}/p_{vk}) = \frac{\exp{(v^T_{w_i} p_{vk})}}{\sum_{w \in W} \exp{(v^T_w p_{vk})}} \tag{3}$$

However, computing softmax for corpus with large $|W|$ (size of vocabulary) is computationally inefficient. So we use hierarchical softmax [5] approximation instead of computing full sofmax. It basically means, we define a binary tree structure (usually huffman-tree for computational efficiency) where leaves represent words w and internal nodes also have representation $\hat{w}$. So, each word is represented as a path in this tree from root to the leaf. We compute the probability of word given the paragraph vector as below.

$$p(w_{ik}/p_{vk}) = \prod_{j=1}^{L(w_i)-1} \sigma(I(is\_left(j+1))\hat{w_{ij}}^T p_{vk}) \tag{4}$$

Where, $L(w_i)$ is length of the path from root to the word w_i. $\hat{w}^T_{ij}$ indicates j^{th} internal node in the path from root to w_i. $I(is\_left(j+1))$ is +1 if $(j+1)^{th}$ node is left child, -1 otherwise. So, we also need to store, T representation of the tree and $\hat{W}$, matrix containing internal node representation. We'll use Stochastic (in case of large paragraphs) Gradient Descent to solve the optimization problem mentioned in equation (2). Note that each optimization can be solved independently (However, they share the same parameters.). The gradient of the objective function w.r.t p_k is

$$\nabla_{p_k} NLL(p_{vk}, P_k) = -\sum_{w_{ik} \in P_k} \sum_{j=1}^{L(w)-1} [1 - \sigma(I(is\_left(j+1))\hat{w_{ij}}^T p_{vk})] * [I(is\_left(j+1))] * [\hat{w_{ij}}] \tag{5}$$

We'll refer to $\sum_{w_{ik} \in P_k}$ as outer summation and $\sum_{j=1}^{L(w)-1}$ as inner summation. The minimization involves repeated computation of $\nabla_{p_k} NLL(p_{vk}, P_k)$ for different values of p_k. As suggested in the paper [1], we'll approximate the outer summation through sampling. We'll not explicitly compute the summation, instead we sample one word from paragraph at a time and minimize the cost function. Also, instead of running the optimization until convergence, we'll run it for a fixed number of steps (proportional to number of words in the paragraph), which gives fairly good representations.

2.2 Data Structure

From the problem formulation, it's clear that here are the set of parameters we will require:

- **Word Indexes (W_D):** A hash-map from word strings in V to index, called as "word index" from hereon. We need this to avoid doing hashing in the kernel code. This will be maintained on CPU side and this word index is used in tree representation and indexing into softmax parameters.(Read only)

- **Tree Structure (T):** Hash-map from word index to bit-vector $\{0, 1\}^{L(w_i)}$. Where 0 represents left child and 1 represents right child. Length of the bit-vector is equal to length of the path from root to the word in hierarchical softmax tree. We'll refer to them as word code vector $T.code(w_i)$. Another hashmap from a word index to multiple indexes (stored as 1-D array) in $\hat{W}$, that correspond to representation of internal nodes visited in the path from root to the word. We'll refer to them as point vector $T.point(w_i)$. (Read only)

- **Soft-max Parameters ($\hat{W}$):** $\hat{W}$ corresponds to vector representation for internal nodes of hierarchical softmax tree. There are shared parameters across words. (Read only)

- **Paragraph Vectors (P):** Matrix that contains vector representation of paragraphs. 1^{st} row will refer to 1^{st} paragraph and so on. (Read / Write)

2.3 CPU Work

1. Load the data-structures mentioned above W, T and $\hat{W}$ and transfer them to device global memory. The data-structures are outputs of python version of Word2Vec [6].

2. Randomly initialize P for a batch of paragraphs and transfers those randomly initialized P to device memory.

3. For each paragraph in the present batch, sample one word w_k and transfer them to device global memory. Launch kernel such that, each block corresponds to a paragraph and each thread corresponds to a dimension of representation. Each launch computes $p_{vk(new)} = p_{vk(old)} - \alpha * \nabla_{p_k} NLL(p_{vk}, P_k, w_k)$, NLL is indexed with w_k to indicate that, it's gradient step w.r.t to a single sampled word w_k for k^{th} paragraph.

4. Read back the results of P^* from device memory.

2.4 GPU Work and Kernel Function

We'll assign one sentence and one sampled word to each block. This allows us to run optimization for each sentence independently in each block. Since, we optimize (just take one gradient step) w.r.t one sampled word at a time, we'll launch the kernel multiple time for each paragraph. Number of launches is proportional to number of words in each paragraph, to find good representations.

```
**Data**: $w_k$, $T$, $\hat{W}$, $p_{vk}$ and $\alpha$
**Result**: PV for paragraph k: $p^*_{vk}$
Load corresponding parts $T_{w_k}$, $\hat{W}_{w_k}$ and $p_{vk}$;
Load $p_{vk}$ shared memory ;
__syncthreads();
Compute inner loop of equation (5) with present value of $p_{vk}$ ;
__syncthreads();
Take the gradient step with size $\alpha$: $p_{vk(new)} = p_{vk(old)} - \alpha * \nabla_{p_k} NLL(p_{vk}, P_k, w_k)$ ;
__syncthreads();
Write $p^*_{vk}$ back to global memory
```

Algorithm 1: Kernel for minimizing $NLL(p_{vk}, P_k, w_k)$ w.r.t p_{vk}

3 GPU Optimizations

1. **Size of Distributed Representation:** Since each thread computes for a dimension of distributed representation, we are better of setting it with multiple of 32. Typically, it is set as multiple of 50. Since, more dimension will only provide us better representations, we can just set it to next multiple for 32, for maximum performance on GPU. However, note that Word2Vec model should also be trained with same dimensionality.

2. **Loading p_{vk} to shared memory:** Kernel function involves repeated dot-product p_{vk} and $\hat{w}_j$ (internal node) and updating of p_{vk} with gradient vector. So, loading it into shared memory would save, several trips to global memory.

3. **Global memory coalescing:** Our data structures T, $\hat{W}$ and P are all stored in row-major format on the device to allow for global memory coalescing in CUDA.

4. **Overlapping CPU and GPU computation:** While GPU is optimizing p_{vk} w.r.t w_k for each paragraph, CPU samples next batch of words w_k's for the next kernel launch.

4 Experimental Setup

Table 1 summarizes the device and compiler configuration used for the experiments.

NYU Cluster	cuda1 - device1
Device	GeForce GTX TITAN Black with compute capability 3.5
Python Version	2.6.6
PyCUDA Version	(2013, 1, 1)
CUDA SDK Version	cuda-5.5

Table 1: Device and Compiler Configuration

We used Stanford's Large Movie Review Dataset [2] for training the word vectors. We used all of */pos*, */neg* and */unsup* reviews for training our Word2Vec model. We used one sentence from each review as a paragraph for which distributed representation has to be found. For showing the speedup, we used all sentences from */unsup* subset of the corpus. In total we had $343,296$ sentences after omitting sentences of length less than 4 and greater than 20. The reason why we drop such sentences is that, we want to run on paragraphs of comparable lengths so that we can run them for same number of iterations. However, the code is general enough so that one can run paragraphs of differing sizes in different batches with appropriate number of iterations. Also, we use 96 dimensional word vectors and paragraph vectors to exploit full warp capacity of CUDA devices (Multiple of 32).

It is very challenging to access the correctness by just looking at representations found (vector of numbers). We assess the correctness of the program qualitatively by looking at nearest neighbors of a fixed set of sentences. For each sentence in our test set, we look at what sentences are close to the given sentence and asking if these sentences semantically make sense. We use cosine similarity metric for finding nearest neighbors (Note: Cosine score is always in $[0, 1]$ and higher score, more similar are sentences). Instead of picking just one sentence, we pick top five sentences. We also compare the nearest neighbors to nearest neighbors found using our CPU code. The nearest neighbor lists won't be identical because of the random initialization of paragraph vectors. But the lists will be comparable and there may be some overlap between the CPU and GPU lists.

5 Experimental Results

First, let's look at the correctness results. Table 2 to 6 shows examples of nearest neighbors found after training paragraph vectors. First sentence in bold is our test sentence and next five sentences are nearest neighbors found with their corresponding cosine scores. Notice that, all nearest neighbors semantically make sense with fairly high scores of similarity. Which confirms the correctness of the implementation. Nearest neighbors talk about same aspect of the movie. However, they ignore the polarity (positive sentence / negative sentence) of the sentence because they aren't trained to do that. In some cases, there are no common words between sentences. In Table 3 look at the last sentence. In Table 4 forth nearest neighbor talks about actors performance in a movie but gives negative opinion. This non-trivial generalization of paragraph vectors make it very powerful in many NLP applications.

sentence	**cosine score**
she looked lovely and her voice was done really well.	-
she looked amazing and i was completely mesmerized by her beauty.	0.826635479927
she plays her part very well.	0.816613674164
she just clicked in the role and it worked wonderfully well.	0.799343049526
preity looks great in her new look and has acted well.	0.789747416973
oh and did i mention she looked hot in those pants!	0.784641623497

Table 2: Nearest Neighbor Example

sentence	**cosine score**
i'll be honest with you.	-
ok, i'll be honest with you.	0.938422322273
i'll tell you; it will be turkey.	0.843528151512
you won't be disappointed, i guarantee you.	0.842787504196
now this movie may not be for everyone.	0.828628003597
i'll try my best to be objective but it won't be easy.	0.819331347942

Table 3: Nearest Neighbor Example

Our CPU version is implemented with python and pushes the compute incentive parts to highly optimized numpy (Numerical Python) routines. We have also vectorized our python implementation to exploit computational benefits of numpy. But the entire python code runs on a single thread. To measure the speedup, we run our experiments on corpus of different sizes. A collection of 1024 sentences which we'll refer to as a batch will be our unit of size. We run both CPU and GPU codes on corpus size of 1, 2, 4, ... 64 ... 256,

sentence	**cosine score**
she was great and put those other actresses to shame.	-
everyone played a great part from unknown people to vivek, nana, saif and others.	0.796581208706
i believe chloe will be starring in other movie's, as she is a good actress like her mother.	0.752288043499
i liked it that luck played a part in jaguar paw's endeavours.	0.741691172123
half the cast members don't seem to be into this story, either, especially duvall who usually gives great performances.	0.739531993866
david will come to his senses and will realize he has loved rachel all along.	0.739332199097

Table 4: Nearest Neighbor Example

sentence	**cosine score**
he deserved an academy nomination at the very least.	-
duvall certainly deserved his academy award nomination.	0.884090423584
gloria grahme however was honored with an academy award for her delightful and refreshingly sanguine portrayal of rosemary.	0.862568259239
actress gloria grahame won an academy award.	0.85575735569
i am one of many who also feel she deserved an academy award for her portrayal of ruth etting.	0.849912762642
daniel day-lewis should have received the best actor award at the academy awards.	0.849499940872

Table 5: Nearest Neighbor Example

sentence	**cosine score**
i saw the unrated version from netflix.	-
i saw it as part of the popeye the sailor volume 2 (1938-1940) dvd.	0.721547722816
i watched the subtitled version of the film.	0.717457652092
the dvd i rented had all the zits and dust of the original film.	0.68967628479
i just saw this series collection box at a store, was fooled by it's cover and bought it.	0.686215996742
but for me, this is the definitive version of the film.	0.683518886566

Table 6: Nearest Neighbor Example

334 batches. Figure 2 shows the punch line of this project. For all GPU and CPU runs we present average running time across 5 runs with an exception for CPU runs with size greater than 64 batches. We obtain about **25x** speed improvements over the CPU version. For our full test corpus, which has 343, 296 sentences, CPU takes about **63 minutes** to run where as our optimized GPU version takes about **2.5 minutes**.

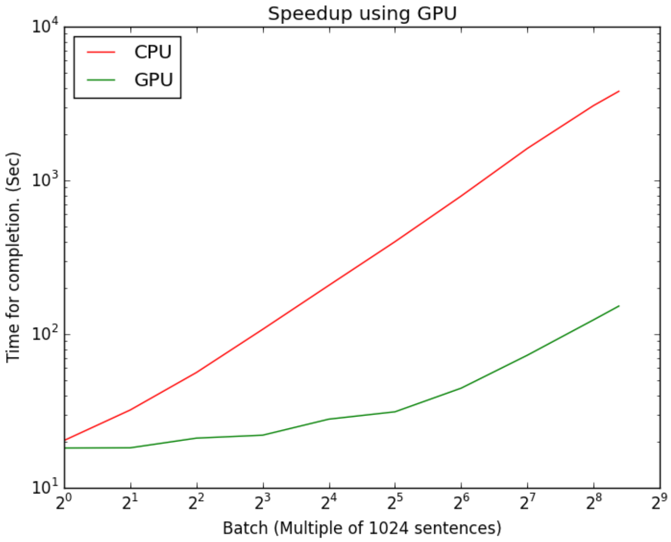

Figure 2: Speed up due to GPU

6 Conclusion

Because optimization for each paragraph is interdependent, GPU was very suitable for the problem. Through our implementation, we obtain ∼**25x** speed improvements over our own CPU implementation. To get the above mentioned speedup, we require that dimension of paragraph vectors be multiple of 32. However, one can always set the dimension to next multiple of 32 without hurting the statistical efficiency of representation. One of the major overheads of our present implementation is that data associated with our soft-max tree T, is transferred to GPU memory one word at a time. Which gives initial overhead of $\sim 18secs$. We can overcome this by using a different data structure which stores representations in a contiguous fashion. Another area of improvement is load balancing. Since number of kernel launches is proportional to number of words in a given paragraph, we can put paragraphs of comparable sizes into same batch. We can gain some speed improvements by running it for shorter iterations for shorter paragraph.

References

[1] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[2] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[3] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[4] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[5] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *AISTATS05*, pages 246–252, 2005.

[6] Radim Rehurek. Gensim. `http://radimrehurek.com/gensim/models/word2vec.html`, 2013.