Language Model on GPU

Junyi Li

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2017

Abstract

Language model is an important component in natural language processing and it contributes most of the performance to its applications such as machine translation, optical character recognition, and automatic speech recognition. Recently, neural language model and N-gram model are two popular model and applied widely. Neubig and Dyer (2016) contributes a framework, MODLMs, which combines the features of these two language model and achieve a better performance than either single one.

Though there are some existing N-gram language model toolkits in the world, no one was designed for MODLMs because it requires a complete distribution for each n-gram history. Therefore, This project was designed to build a queryable language model for MODLMs. We analyze the requirements of this framework and design the data structure and query algorithm for it. To improve the query speed, this language model is built on GPU and based on the first GPU language model (Bogoychev and Lopez, 2016).

Acknowledgements

Many thanks to my family for growing and supporting me. Thanks to my supervisor Adam Lopez and Nikolay Bogoychev, who instruct me so much during the period of doing my project and dissertation and help me learn many interesting things. Thanks to my colleagues in the study group instructed by Adam, I was so lucky that I have this chance to study with them. Many thanks to Andreas Grivas especially give me many inspirations. Besides, thanks to all my friends, Chinese or not Chinese, all of them give me many support.

Thanks to Edinburgh University for offering me this study environment. Thanks to the people who share their study experience on the Internet. Thanks to my country, the powerful backup of me.

Finally, many thanks to myself. Thanks to you for insisting and did not give up. Good luck.

Declaration

(*Junyi Li*)

Contents

Chapter 1

Introduction

The two most important days in your life are the day you are born and the day you find out why.

—-*Mark Twain*

1.1 Motivation

1.1.1 No Language model for MODLMs

Language model is a pivotal unit in natural language processing such as machine translation, optical character recognition, automatic speech recognition, and natural language generation. Its performance affects these applications. Therefore, people are trying to improve language model so as to improve these kinds of useful applications. Mixture Of Distribution Language Models (MODLMs) (Neubig and Dyer, 2016), is a novel framework. It combines the features of the N-gram language and neural language model to get better results than either single kind of language model.

There are already many existing N-gram Language Model toolkits which are designed to store and query language model such as KenLM (Heafield, 2011), IRSTLM (Federico et al., 2008), BerkeleyLM (Pauls and Klein, 2011) and Sheffield (Guthrie and Hepple, 2010), gLM (Bogoychev and Lopez, 2016). However, none of the LMs mentioned above was designed for MODLMs (Neubig and Dyer, 2016). MODLMs offers a Neural/ngram Hybrid Model performing interpolation on softmax layer, which require the LM return a complete distribution for reach n-gram history, rather than a single n-gram parameter to calculate the probability of a sentence. Therefore, existing LMs are inefficient for MODLMs. **The primary purpose of this thesis is to build an**

efficiently queryable N-gram language model, called giLM, for MODLMs framework.

1.1.2 The successful application of GPU

Because N-gram language is usually used to process large vocabularies, the query speed is often a problem (Heafield, 2013; Spence Green and Manning, 2014). To address this problem, massively parallel hardware, general purpose graphics processing units (GPUs) was used by gLM (Bogoychev and Lopez, 2016), which proved the language model on GPU could outperform on CPU. GPU is better then CPU over memory bandwidth and computational throughput (Figure 1.1).

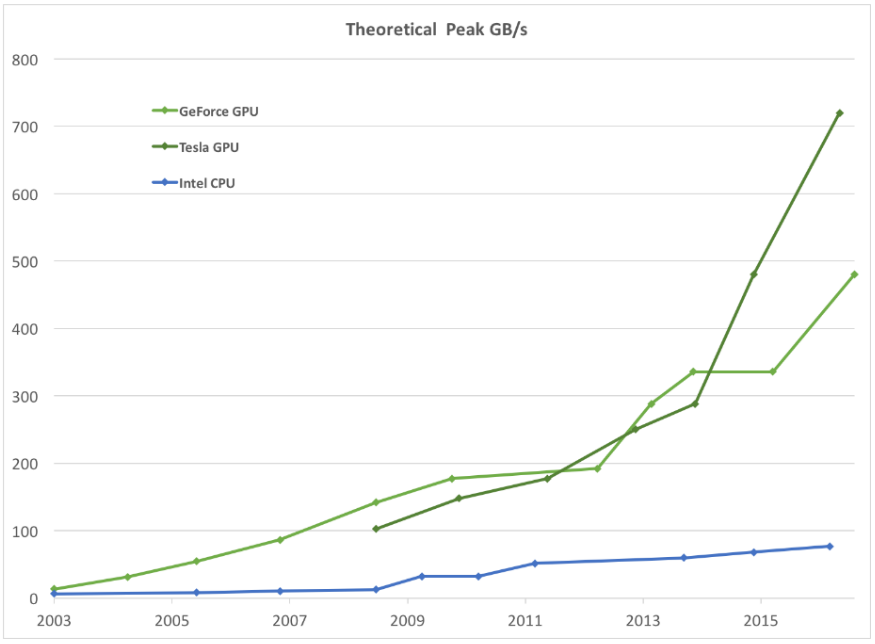


Figure 1.1: Comparison between CPU and GPU on bandwidth (Nvidia, 2017)

Therefore, this project was based on gLM and also was designed on GPU. We choose forward trie and Btree as the basic data structure, design a new data analysis algorithm on CPU and a parallel traversal algorithm on GPU for querying. Besides, we designed serials of experiments to test its performance and the results will be analyzed in chapter 5.

1.1.3 Motivation evolution

Figure 1.2 shows the this thesis's Pilgrimage. Initially, we are trying to improve Neural Machine Translation (NMT) system (1), because the language is really a prob-

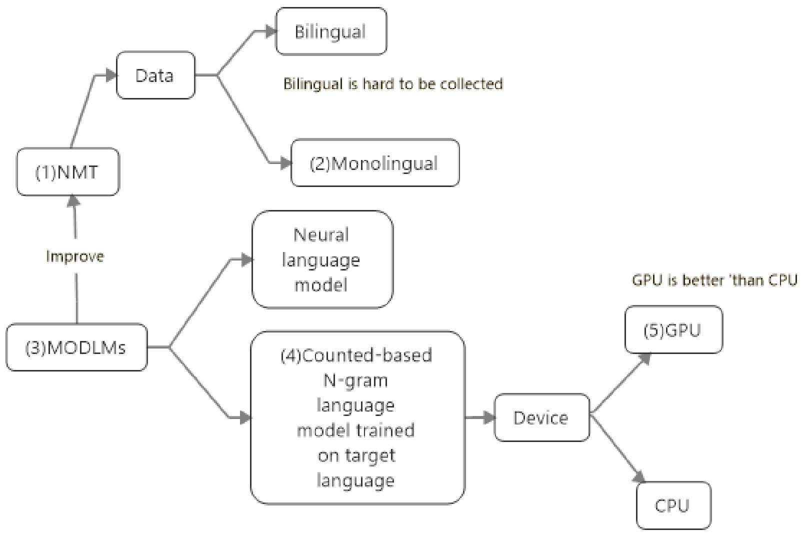


Figure 1.2: This diagram shows the evolution of motivation.

lem for knowledge sharing all over the world and NMT system has showed its power to help solving this problem. However, bilingual training data is hard to get, so the performance of NMT system was hard to be improved especially for minority language (Koehn and Haddow, 2012). Therefore, we turn to utilizing monolingual language (2). The N-gram trained on target language can be integration into MODLMs (3) to improve the neural language model, which affects NMT system. Therefore, we want to build an efficient language model for MODLMs (4). GPU (5) is an accelerator to our implementation and inspired by gLM (Bogoychev and Lopez, 2016).

1.2 Aim and objective

The aim of this project is to build an N-gram efficiently queryable language model for MODLMs framework. Five sub-objectives are formulated to instructed the advancement of this project. This project is expected to improve MODLMs and related applications which need neural language model, such as Neural machine translation.

- **Preparation** Firstly, understands the requirements of MODLMs. Then, make a survey about the implementation of language model, and find a suitable data structure and algorithm for this project.

- **Implement a forward trie** Words are stored in forward trie in a natural left-to-right order, so it is efficient to ask forward trie for a list of rightward extensions given context.

- **Design and implement a data analysis on CPU** Because MODLMs need every possible conditional probabilities for different n-gram order, it needs a special algorithm to process input sentence to generate queryable sentence for GPU.

- **Design and implement a query algorithm on GPU** To return the all the possible conditional probabilities given a specific context, we need a new query algorithm different from gLM's.

- **Compare giLM with gLM and make an evaluation** In order to evaluate the implementation and performance of this project, we design a serials of experiments to compare giLM with existing GPU language model gLM, and make an evaluation for giLM.

1.3 Achievements

This thesis presents a forward trie, a data analysis algorithm and query algorithm to build an efficiently queryable language model for MODLMs framework. We also design serials of experiment to compare the performance of giLM and discover the points to be improved.

1.4 Outline

General purpose graphics processing units (GPUs) consists of thousands of smaller than CPU's cores designed for parallel tasks. In contrast, CPU consists of fewer cores designed for sequential tasks. The comparison showed in Figure 2.2. Parallel computing is suited to GPUs and the application of it will improve the query speed of language model, which is the main reason why it is coming to compete with CPU. The rest of **chapter 2** will describe the details of GPUs and parallel computing.

Because of the limited memory size in GPU, trie was chosen to be the data structure storing the language model. To do parallel searching and query, Bogoychev and

Lopez (2016) skillfully combined Btree and reverse trie to do parallel search and backoff computation. This thesis replaces reverse trie with forward trie because former one is inefficient for our model (we do not do backoff computation). **Chapter 3** contributes the details about the methodology to implement language model design and algorithms.

Chapter 4 depicts the experiments to test the performance of this project in order to achieve the fifth sub-objective mentioned in section 1.2. We also raised four hypotheses, which will be discussed in chapter 5.

Chapter 5 demonstrates the results of the experiments presented in chapter 4. For each experiment, we analyze the results and presents a conclusion for these experiments. Besides, it also mentioned the problems I encountered during this process and suggested solutions for them.

In the end, **chapter 6** summarizes and draws final conclusion of this project and suggests ideas for the possible future work.

Chapter 2

Background

The world is your oyster. It is up to you to find the pearls.

—-The pursuit of happiness

2.1 Language Model

Models that assign a probability to a sentence is Language Model. It can be used to measure how likely a sentence to be a natural sentence. Therefore, it is widely applied in machine translation, speech recognition and natural language generation. Below is its mathematic definition:

Given a sequence of words:

$$w_1, ..., w_l$$

where l is the number of words in this sentence, and w_i is the i^{th} word in this sentence. The probability of it is:

$$P(w_1, ..., w_l)$$

The conditional probability is defined as:

$$P(w_i | w_1, ..., w_{i-1})$$

Therefore, the probability of a sequence of words can be defined as below using chain rule:

$$P(w_1, ..., w_l) = \prod_{i=1}^{l} P(w_i | w_1, ..., w_{i-1}) \tag{2.1}$$

2.1.1 N-gram language model

N-gram language model is the simplest language model under the application of Markov assumption. As Markov assumption, the words are only related with a finite number of preceding words. Let $n-1$ be this number, equation 2.1 can be transformed to:

$$P(w_1,...,w_l) = \prod_{i=1}^{l} P(w_i|w_{i-n+1},...,w_{i-1}) \tag{2.2}$$

where $n-1$ represents how many preceding words will be applied to generalize the conditional probability.

For one of the conditional probability computations in equation 2.2, the equation shows below:

$$P(w_i|w_1,...,w_{i-1}) = P(w_i|w_{i-n+1},...w_{i-1}) = P(w_i|w_{i-n+1}^{i-1}) \tag{2.3}$$

where w_{i-n+1}^{i-1} represents a word sequence from $i-n+1$ to $i-1$. Therefore, only $n-1$ words before w_i affect the probability of w_i.

2.1.2 Neural network language model

Neural network was presented (Bengio et al., 2003) to train language model. Usually, neural network contains a wordembedding layer, a few hidden layers and a softmax layer.

Take a one hidden layer neural network as an example. Given w_{i-n+1}^{i-1}, it firstly converts each word into vectors v_{i-n+1}^{i-1} on wordembedding layer. Then v_{i-n+1}^{i-1} will be concatenated into an vector W, which represents all the information of w_{i-n+1}^{i-1}. Then this vector W will be fed into hidden layer to get a hidden state:

$$h = hiddenlayer(W, \theta) \tag{2.4}$$

where θ is the parameter in neural network. Finally, the probability vector will be calculated by softmax layer.

$$p = softmax(h, \theta_s) \tag{2.5}$$

where h is transfered from hidden layer, θ_s is parameter in softmax layer.

2.2 MODLMs

MODLMs (Neubig and Dyer, 2016) is a framework which can be applied to implement a Neural/n-gram Hybrid Model.

$$\overbrace{\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_J \end{bmatrix}}^{\text{Probabilities } \boldsymbol{p}^\intercal} = \underbrace{\begin{bmatrix} d_{1,1} & \cdots & d_{1,N} & 1 & \cdots & 0 \\ d_{2,1} & \cdots & d_{2,N} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ d_{J,1} & \cdots & d_{J,N} & 0 & \cdots & 1 \end{bmatrix}}_{\text{Count-based probabilities } \underline{\text{and}}\ J\text{-by-}J \text{ identity matrix}} \overbrace{\begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_{J+N} \end{bmatrix}}^{\text{Result of softmax}(\text{NN}(\boldsymbol{c}))}$$

Figure 2.1: Neural/n-gram hybrid LMs (Neubig and Dyer, 2016)

Figure 2.1 shows the function used to mixture two language model by performing interpolation on softmax layer. In this function, λ_{J+N} contains not only probabilities of words, but also the mixture coefficients for the count-based n-gram model. λ is the result of softmax in the output of neural language model. λ_{J+1}... λ_{J+N} is the distribution over different Markov order $P(p_n(w_i|w_{i-1},...,w_1))$. $\lambda_{1...J}$ is the probability distribution over vocabulary set containing J elements. $d_{1,1}$...$d_{J,N}$ is the data from count-based N-gram language model for word J. Matrix 2.6 is what this framework needs, so the elements in this matrix are also the return results of this project.

$$\begin{bmatrix} d_{1,1} & \cdots & d_{1,N} \\ d_{2,1} & \cdots & d_{2,N} \\ \vdots & \ddots & \vdots \\ d_{J,1} & \cdots & d_{J,N} \end{bmatrix} \tag{2.6}$$

MODLMs access the information both from n-gram LM and neural LM and it allows neural network focus on the information not well captured by n-gram LMs. Besides, it theoretical may need less training time and training data than normal neural network because the training is based on a trained n-gram LM. Neubig performed experiments on the Penn Treebank (PTB) dataset (Mikolov et al., 2010), and the first 100k sentences on the English side of the ASPEC corpus (Richardson et al., 2015), taking a method called block dropout (Ammar et al., 2016) to avoid poor local minimal value. The experiment results and comparison shows this hybrid LM achieve a big improvement in perplexity over single n-gram models or LSTM neural language model with block dropout. In this case, this framework can produce a more precise final probability than the result of either language model inside it.

2.3 GPU and Parallel computing

GPUs are constructed in a different way from CPUs to address parallel computing, which decides the existing language model data structure designed for widely-used x86 CPUs is not suited well for GPU.

2.3.1 GPU Construction

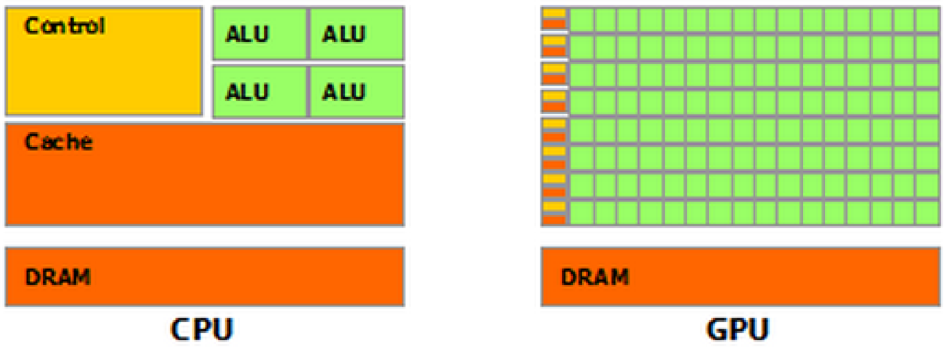


Figure 2.2: Constructions of CPU and GPU (Nvidia, 2017)

As Figure 2.2 shows, GPU has much more cores than CPU, and each core can execute single thread on one data element. An NVIDIA GPU consists of several ”Streaming Multiprocessors” (SMs), each of which hosts several cores. Each SM executes the same instruction at a time step. As Figure 2.4 shows, each thread has its own register, and all the threads of in a block share one memory. Block is a software concept. (Figure 2.3 shows more information about thread, block and grid)

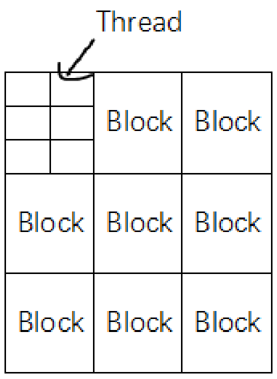


Figure 2.3: This is a diagram showing the concept of grid, block and thread.

This entire picture (Figure 2.3) is a grid, where nine blocks in and each block has several threads. Computation on GPU is performed by kernel, which is similar to the function in traditional C/C++ program. When the kernel is called, the size of grid and block will be defined at the same time. The cores allocated to one kernel is an integer multiple of warp, which is a hardware concept. If the demand of cores is less than a warp, warp is still been assign but the rest of cores in a warp idle.

Table 2.1: The comparison of different memory in a general situation (Nvidia, 2017)

Memory type	Latency	Size
Register	0	4B
Shared	4-8	16KB-96KB
Global GPU memory	200-800	2GB-12GB
CPU memory	10K+	16GB-1TB

Figure 2.4 presents the relationship between the memory and thread in GPU. And Table 2.1 shows the size and latency of different memory. These information implicates that to make sure the query speed needs as many queries happened on the memory that has low latency as possible. However, the faster query memory has less size, so the size of data structure should be small enough. Furthermore, we can not avoid access data from Global GPU memory even though we can store the whole language model on GPU (we did), so the coalesced read is important (Details in section 2.3.2.1).

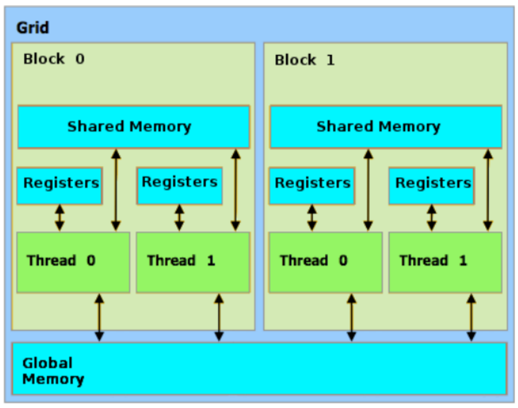


Figure 2.4: GPU-memory (Nvidia, 2017)

Thread	store x	if x>0	z=sqrt(x)	z=x^2	
1	x=3	yes	z=sqrt(3)	idle	
2	x=2	yes	z=sqrt(2)	idle	
3	x=1	yes	z=sqrt(1)	idle	
4	x=-1	no	idle	z=1	
5	x=-2	no	idle	z=4	
6	x=-3	no	idle	z=9	
	Start	time=1	time=2	time=3	End

Figure 2.5: Warp-divergence (branch instruction)

2.3.2 Notes for GPU programming

Warp divergence As described in the previous section, every cores in a warp must execute the same instruction. However, if there is branch instruction, only the part of cores satisfies the condition will execute and rest of them idle. In Figure 2.5, at time = 1, all of the threads in a warp execute the instruction to make a comparison between x and 0. At time = 2, only do thread 1,2,3 execute the instruction z = sqrt(x). At time = 3, only do thread 4,5,6 execute the instruction z = x^2.

Therefore, Warp divergence brings extra loss of parallel efficiency (one more time step than normal parallel computing), which is the thing we need to avoid in GPU programming.

Coalesced reads This is a concept from CPU memory access. Because data reading takes block as a basic unit on CPU, and all the data in one block will be read at once. Therefore, we should try to store the relevant data in the same or adjacent block to reduce access times. Similar to CPU, we also want to reduce the access times on GPU global memory, so we try to access GPU global memory with coalesced reads. If threads, which have a common shared memory, could request and fetch consecutive data from GPU, then single access can satisfy all these threads. gLM (Bogoychev and Lopez, 2016) utilized Btree to do this, so does this project.

2.4 gLM

gLM (Bogoychev and Lopez, 2016) is the first language model designed on GPU, the results of its experiments showed it could outperform KenLM (Heafield, 2011) when the expense of hardware is limited.

2.4.1 Backoff computation

The computation of Backoff language model used in gLM is:

$$P(w_i|w_{i-1},w_{i-2}...w_{i-n+1}) = \begin{cases} \hat{P}(w_i,...w_{i-n+1}) & \text{when } \hat{P}! = 0 \\ P(w_i|w_{i-1},...w_{i-n+2})\beta(w_{i-1}...w_{i-n+1}) & \text{otherwise} \end{cases}$$

where $\hat{P}$ and β is two parameters stored in language model to compute the conditional probability.

2.4.2 Reverse trie

Because of backoff computation, gLM (Bogoychev and Lopez, 2016) takes reverse trie, a popular implementation for backoff computation, as the basic data structure to store data read from language model in a reverse order. In gLM, there are two kinds of arrays to construct trie:

- **array K**: store word
- **array V**: store value of associated word which has the same index with its value. The value contains $\hat{P}$, β and the address to the new array K of next word in trie.

Below is an example to explain reverse trie and the query process. Assuming the query sentence is [A,B,C], for 5-ngram, then:

$$P(A,B,C) = P(C|A,B)P(B|A)P(A)^1 \tag{2.7}$$

Let us take the longest conditional probability of equation 2.7 as an example:

$$P(C|A,B) = \begin{cases} \hat{P}(A,B,C) & \text{when } \hat{P}(A,B,C)! = 0 \\ \hat{P}(B,C)\beta(A,B) & \text{when } \hat{P}(A,B,C) = 0 \text{ and } \hat{P}(B,C)! = 0 \\ \hat{P}(C)\beta(B)\beta(A,B) & \text{when } \hat{P}(A,B,C) = \hat{P}(A,B) = 0 \end{cases} \tag{2.8}$$

Because the words are stored in a reverse order, if trigram $A \rightarrow B \rightarrow C$ stores in the data set, then it will be stored in the form: $C \rightarrow B \rightarrow A$. Therefore, the query process starts from C, then look for B and A. If A is not find, then fetch the $\hat{P}(B,C)$ from B,

[1] We ignore the begin/end symbol to simplify the explanation for the entire thesis except where noted. Let $< s >$ be begin symbol, and $< /s >$ be the end symbol. Then in actual situation, it should be: $P(< s >,A,B,C,< /s >) = P(< /s > |C,B,A,< s >)P(C|A,B,< s >)P(B|A,< s >)P(A| < s >)$

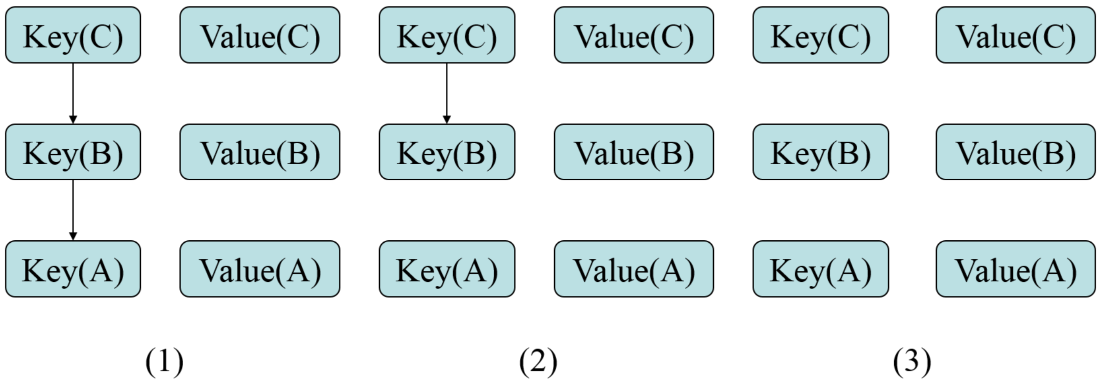


Figure 2.6: A reverse trie diagram for the example. This diagram only shows a single trie from C to B to A in three situations. (1) is the first equation in equation 2.8. (2) is the second equation in equation 2.8 . (3) is the third equation in equation 2.8.

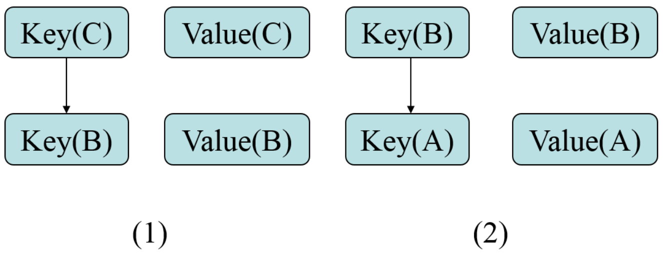


Figure 2.7: This is an explanation for (2) in Figure 2.6. If there is no connection between $C \to B$ and A, which means there is no $\hat{P}(A,B,C)$ in this language model. So we take $\hat{P}(B,C)$ from B. And then start from B in root (this is a new staring point, different from B in $C \to B$) to looking for A and then get $\beta(A,B)$

and go back to root then start from B to A to get $\beta(A,B)$ (Equation 2.8-second answer). If B is not find, then fetch the $\hat{P}(C)$ from C, and go back to root then start from B to A to get $\beta(B)$ and $\beta(A,B)$ (Equation 2.8-third answer). In this model, the first traversal is to start from the C and look for B then A to get $\hat{P}(A,B,C)$ or $\hat{P}(B,C)$ (No A found showed by Figure 2.6-(2)) or $\hat{P}(C)$ (No A or B found showed by Figure 2.6-(3)). The second traversal is from B to look for A to get $\beta(A,B)$ (Figure 2.7) or $\beta(A,B)$ and $\beta(B)$[2]. Therefore, reverse trie only needs twice traversals to get all the parameter to calculate a conditional probability. (But reverse trie is not suited to our problem, which will be discussed in chapter 3)

2.4.3 Btree and K-ary search

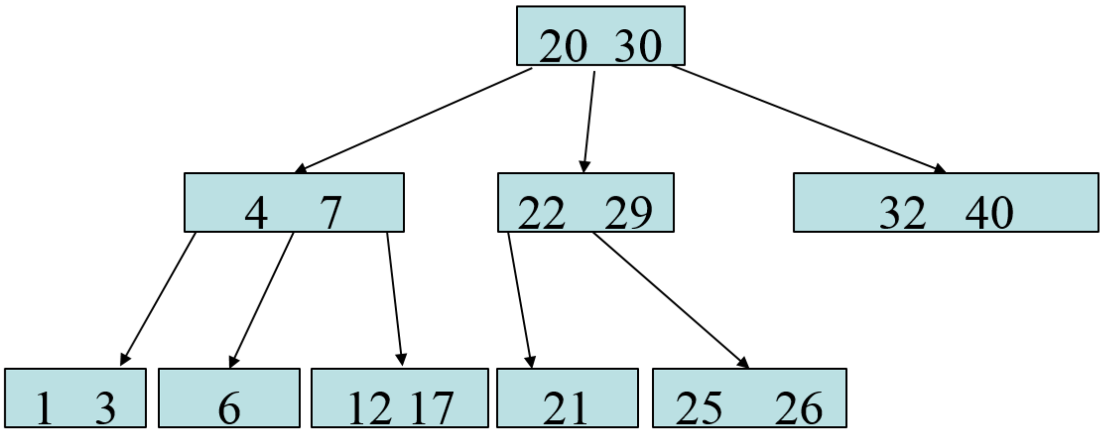


Figure 2.8: An example of Btree

Besides, gLM takes Btree (Bayer and McCreight, 2002) to do K-ary parallel search (Schlegel et al., 2009; Kaufmann and Boston, 2011) for word searching in trie. Each node in a full Btree has (k-1) elements and k child node. As Figure 2.8 shows, keys in the k^{th} child of a node are always larger than $(k)^{th}$ key and less than $(k+1)^{th}$ key in that node [3].

To do K-ary search, similar to binary search, the comparison in Btree will be done at the same time. For example, if we want to find a number 22, two threads will be run to compare 22 with 20 and 30 at the same time. Then, we find the node where 22

2 If any elements in second traversal is not found, just return 0 as β

3 If the key is not number, then as long as the key can be sorted in an order, k^{th} child of a node are always at the right of $(k)^{th}$ key and at the left of $(k+1)^{th}$ key in that node (vice verse)

is stored. Because data in one node of Btree are stored in consecutive memory, so we can do coalesced read with it and also get a faster search speed with K-ary search.

In this project, we keep the basic idea about data structure construction of gLM.

2.4.4 Parallel computing platform

CUDA, introduced by NVIDIA in 2006, is a general-purpose parallel computing platform and programming model designed for NVIDIA GPUs. The programming on CUDA is similar to C/C++ programming but needs to regard the codes as a parallel program. Besides, it provides some toolkits like Memcheck and Racecheck, both of which help coder to find the problem in the CUDA code.

OpenCL, is a open standard framework maintained by the non-profit technology consortium Khronos Group (Trevett, 2013). It offers a platforms for programmer to program code run on GPU.

In this project, we choose CUDA as our parallel computing platform. Because CUDA is more popular than OpenCL presently. Furthermore, CUDA was designed for Nvidia GPU and both of them have been applied to do most of the machine learning problem.

Chapter 3

Methodology

Man is not made for defeat. A man can be destroyed but not defeated

—-The Old Man and the Sea

To achieve the sub-objective 2, 3, 4 stated in chapter 1, this chapter describes the design for this project. A overview of this project design is showed in Figure 3.1. We at first analysis the problem that giLM aims to solve in details. Secondly, we explain the data structure of giLM. Then, we present the query algorithm of giLM. Finally, we describe the implementation of this project.

3.1 Problem Overview

According to the function in Figure 2.1 we explained in section 2.2, this function needs the probabilities of all possible following words given context (matrix 2.6), in which giLM will return all the elements for one query. Therefore, the query process for giLM can be modeled below:

- Input sentence: $w_1, w_2, ...w_l$ (Can be expressed as: w_1^l)
- Output: $P(w_i = X|w_{i-n+1}^{i-1})$ for all $n \in \{1...N\}$, all $i \in \{1...l\}$. In this case, X represents every possible word in the language model and N is the order of N-gram. The outputs will contain all the elements in matrix 2.6, which can be written as matrix 3.1:

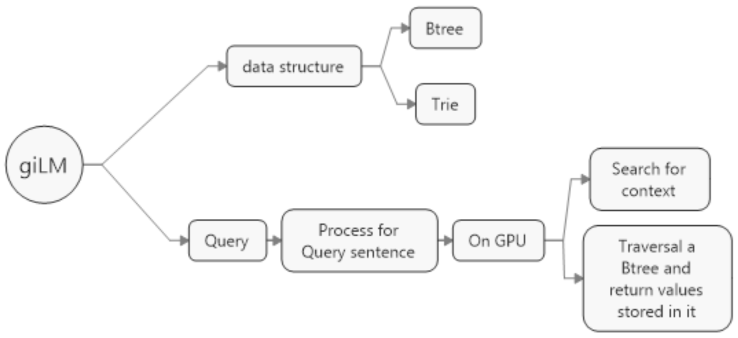

giLM
data structure
Btree
Trie
Query
Process for Query sentence
On GPU
Search for context
Traversal a Btree and return values stored in it

Figure 3.1: The overview of the project design

$$\begin{bmatrix} P(w_1) & \cdots & P(w_1|w_{i-N+1}^{i-1}) \\ P(w_2) & \cdots & P(w_2|w_{i-N+1}^{i-1}) \\ \vdots & \ddots & \vdots \\ P(w_l) & \cdots & P(w_l|w_{i-N+1}^{i-1}) \end{bmatrix} \tag{3.1}$$

For an existing language, gLM:

- Input sentence: $w_1, w_1, w_2, ...w_l$ (Can be expressed as: w_1^l)
- Output: $P(w_1^l)$, the probability of query sentence.

According to the analysis above, gLM only returns one probability of a query sentence. However, this is not what we want. This project is designed to output all the elements in matrix 3.1 for the function showed in Figure 2.1. With the requirements of MODLMs in mind, we design the data structure and query algorithm for giLM.

3.2 Data structure

After reading and making survey about existing data structure used to implement language model, this project is based on gLM (Bogoychev and Lopez, 2016), and takes forward trie and Btree to build the language model. With our talk in section 3.1, this problem needs a list of rightward extensions given context and forward trie is suited to solve this problem. The rest parts of this section will discusses forward trie and the data structure used by this project.

3.2.1 Forward trie

Different the reverse trie used by gLM (Figure 2.6), forward trie takes a reverse order compared with reverse trie (Figure3.2).

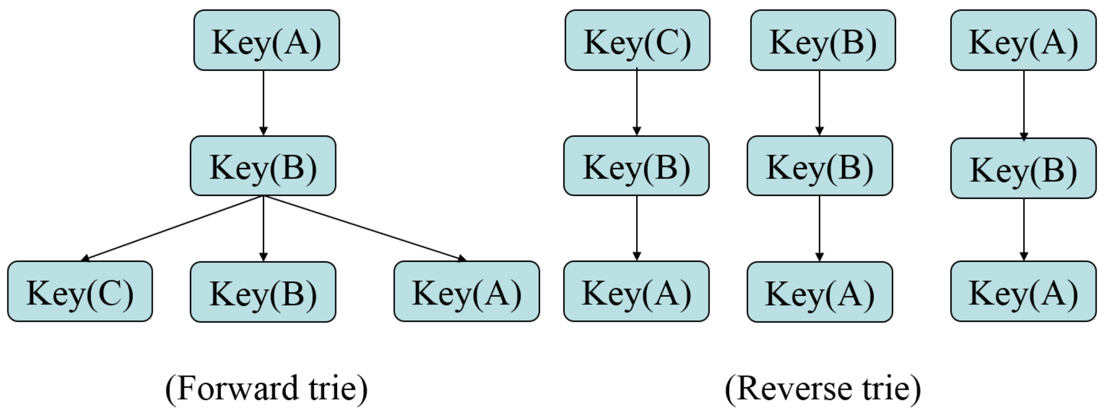


Figure 3.2: The comparison between forward trie and reverse trie taking an example data set = A,B,C. We ignore the value of them to save space.

As Figure 3.2 shows, if we want to return $P(X|A,B)$ (here X represents A,B,C), the things we need to do is find A firstly, and then find B. After that, return all the child nodes of B. However, for reverse trie, we need to go through three trie to get the $P(X|A,B)$. If the data set is very large (the data set used in chapter 4 contains more than 100 thousand different words), we have to go through more than 100 tries. But for forward trie, we only need to go through once and return the child nodes of B. If we do not find the word we are querying, just return 0, which will be talked in section 3.3.

As we talked above, forward trie can return a list of rightward word given context and it has been used by IRSTLM (Federico et al., 2008), so we choose forward trie as our basic data structure.

3.2.2 Terms used in this chapter

Here are some terms used in this chapter.

- Trie: the whole data structure is a trie.
- Btree_trie: a node of trie except root of trie. And if we extract one Btree_trie, it will show in form of Figure 3.5
- Btree_trie_level: the level in one Btree_trie

- Trie_level: the level in trie regardless of Btree.
- VocabID: A number as the representation of word in language model [1].

3.2.3 Data structure of this language model

The design of this language model showed in Figure 3.3: The unigram is stored in root node, and 2-gram is stored in the child node of root and so on. Figure 3.4 showed the data layout in trie. Similar to gLM, there are two arrays to store the language model.

$$K[index] = word \tag{3.2}$$

$$V[index] = value \tag{3.3}$$

The reason why we store backoff here is to make comparison relatively fair, the details of which will be described in experiment for comparison with gLM.

For unigram, we store word into an array K. Then the index of array K is the associated vocabID of that word. Then the value of this word will be stored into an array indexed by vocabID. So equations 3.2 and 3.3 can be written for root node as follows.

$$K[vocabID] = word \tag{3.4}$$

$$V[vocabID] = value \tag{3.5}$$

Because we will transfer the words in query sentences into associated vocabIDs in terms of equation 3.4, the probability of one word ($P(w_i)$in equation 3.1) can be retrieved directly by equation 3.5. Therefore, the query algorithm which will be talked in next two sections just for conditional probabilities.

Apart from root trie node, every child node of trie is a Btree, called Btree_trie here. As Figure 3.5 shows, each number in the figure is the index of word and also the index of value. As long as we find the word, then the value of this word can be retrieved.

The data used to construct language model are known in advance, so they can be sorted in advance. Therefore, the construction algorithm can build the entire trie in a sorted order and the Btree to be in the form showed in the diagram.

[1] Word's vocabID is equal to the associated index of array in root trie node as equation 3.2.

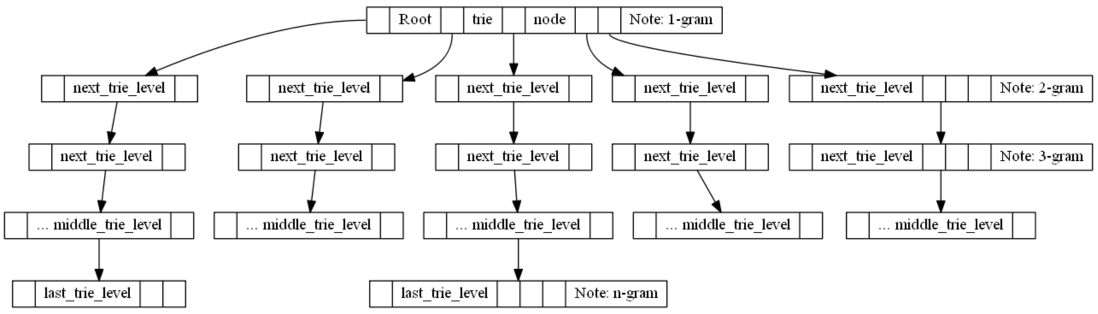


Figure 3.3: The overview of the data structure, each node in the diagram is a Btree showed in Figure 3.5 except root node. This figure is inspired by (Bogoychev and Lopez, 2016).

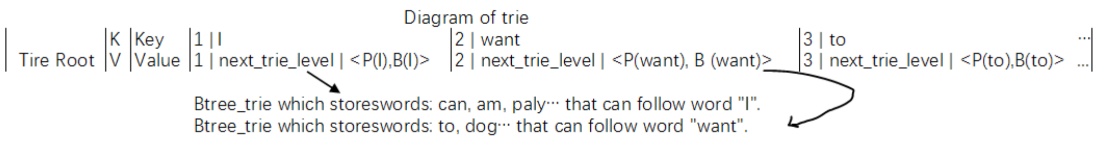


Diagram of trie
Tire Root
K
V
Key
Value
1 | I
1 | next_trie_level | <P(I),B(I)>
2 | want
2 | next_trie_level | <P(want), B (want)>
3 | to
3 | next_trie_level | <P(to),B(to)>
...
Btree_trie which storeswords: can, am, paly··· that can follow word "I".
Btree_trie which storeswords: to, dog··· that can follow word "want".

Figure 3.4: The design of trie with a example

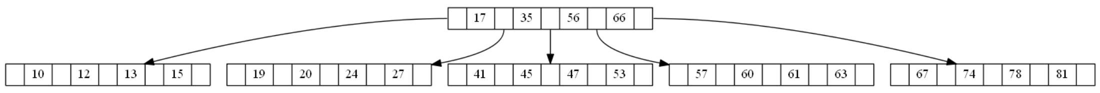


Figure 3.5: This is a diagram for Btree_trie when K = 5. The entire Btree is a node of trie showed in Figure 3.3. This figure is inspired by (Bogoychev and Lopez, 2016)

3.3 Query

As we talked in section 3.1, the form of the output for one query takes this form:

$$P(w_i|w_{i-n+1}^{i-1}) = \begin{cases} P(w_i|w_{i-n+1}^{i-1}) & w_{i-n+1}^{i} \text{ is stored in language model} \\ 0 & otherwise \end{cases}$$

According to the experiments performed by Neubig and Dyer (2016), the n-gram used in this framework is smoothed by Kneser-Ney (Kneser and Ney, 1995). In fact, neural language model could handle the information not captured by N-gram language model. Therefore, even though we do not process the zero probability in N-gram language model, this framework is still able to work.

For unigram probability, we do not need to query on GPU because it is stored into an array and can be fetched directly on CPU. The details will be discussed in section 3.4.

3.3.1 Input data analysis

Before we perform query, we need to read input sentence and transfer it into an appropriate form. From input sentence w_1^l, We fetch pieces w_{i-n+1}^{i-1}, called queryable sentences here[2]. After that, we copy the language model and queryable sentences from CPU to GPU.

3.3.2 Query on GPU

Before this section, all of the steps mentioned above happened on CPU. After accepting the data from CPU, then we do query on GPU. The process of single query is divided into two steps:

- Search the queryable sentence
- Return all the probabilities stored in a trie node for one query

These two steps are described as follows. Let N be the largest order of language model.

[2] The reason why we do it is to take advantages of GPU. Each queryable sentence will be processed by one block. The details of how a block works on a queryable sentence will be discussed in section 3.3.2.1 and section 3.4.4.1

3.3.2.1 Search the queryable sentence

Let us take one queryable sentence, w_{i-N+1}^{i-1}, as an example.

When N = 2, then the query sentence is w_{i-1}. All the data that users want are stored in the root trie, which store all possible words. Array K stores the words, and an associated array V stores the payloads of that word. For example, if we want to retrieve word: w_{i-1} = like, we seek like in array K. Assume K[j]=like, then V[j] is the information we want. This process does not need parallel searching because the array A can be indexed in fixed time.

When N $>$ 2, the first step is the same as previous one described in the last paragraph. After getting the first word w_{i-N+1}, the payloads of w_{i-N+1} offers the address to next_trie_level. This address leads us to a child node of trie, called Btree_trie, which is exact an address to a root Btree node. After that, we do K-ary search in Btree_trie to find w_{i-N+2}. If it can be found, then fetch the address to next trie_level. Otherwise, 0 will be returned and no Btree_trie return. This is a recursive process until find all the words in the queryable sentence.

3.3.2.2 Return all the probabilities stored in a trie node (Btree_trie)

After finding w_{i-N+1}^{i-1} in last section, we get the address to next trie_level, where all w_i and their associated values are stored in a Btree_trie. We perform a preorder traversal algorithm here to go through all the data stored in Btree_trie.

Different from last section, we do not need to search anything, the only thing we need to do here is to fetch the data, which we need, stored in this Btree_trie and write them into an result array. At first, we fetch all the information we need from the root node of Btree_trie by multi-threads at the same time. Then, each thread is responsible for one sub-Btree_trie to do preorder traversal. The details will be showed in next section. After traversal done, we copy that result array from GPU to CPU.

3.4 Implementation

With the conceptual description in previous sections, this section will go deep and describe a complete procedure of querying on this language model in details.

3.4.1 Read and store language model on CPU

This is the first step to read a language model into our designed data structure. Because we need to store these data into our data structure, the following content will be related to memory operation and address calculation.

- For unigram, we store the values of words at address = $f(vocabID)$, where f [3] is a function to calculate the address where store the value of word. Therefore, we do not need to store the actual words, which save the space for words. Since we have a map between word and vocabID, we can just store word using associated vocabID to save space and the address to word's value can be calculated by f [4].

- For bigram, the vocabID of second word will be stored in the child nodes of root trie. The layout of them is showed in Figure 3.6. The data layout in child node is slightly different from root node. The Figure 3.6 presents two different layouts of a Btree_trie. For root node of this Btree_trie or inner node of Btree_trie, the first part, which shows in many numbers only, in the diagram represents the vocabIDs. Offset[5] to Btree_trie child of each node follows the first part. After these, there are payloads of the words stored in this level. Payloads consists of address of next_trie_level, probability and backoff. For the leaf node, there is no offset because they have no child in this Btree. For the last_level_trie, there are only probability and backoff but no address to next_trie_level or backoff. Therefore, there are two different trie node layouts except root trie, two different Btree_trie node layouts, and 4 kinds of layouts in total. We fetch the data from data structure in four different functions, f_1, f_2, f_3, f_4, which are similar to each other but changes a little in terms of layout.

  - Let f_1 be the function to calculate the last Btree_trie in the last trie node(the highest trie and it is the leaf Btree_trie node). The way to calculate the value address is **skip all the vocabIDs and skip previous vocabIDs' value**. The second grid from in Figure 3.6 shows this process. The values only contains probability. Because the size of vocabID or value need to be defined in program, so we just explain the concept.

[3] Function we used is (vocabID-1)*3 here. The value on these three address are: address to next trie_level, probability and backoff (total 3 values).

[4] vocabID is a number for 4 bytes but the length of word is various, which may be very long.

[5] Which used to calculate the address to its Btree_trie child based on current address

- Let f_2 be the function to calculate a inner node in Btree_trie in the last trie node. The way to calculate the value address is **skip all the vocabIDs, offsets and skip previous vocabIDs' value**. The values only contains probability.
- Let f_3 be the function to calculate the last node in Btree_trie but not in the last trie node. The way to calculate the value address is **skip all the vocabIDs and skip previous vocabIDs' value**. The value, different f_1, contains address to next_trie_level, probability and backoff.
- Let f_4 be the function to calculate a inner node in Btree_trie but not in the last trie node. The way to calculate the value address is **skip all the vocabIDs, offsets and skip previous vocabIDs' value**. The value contains address to next_trie_level, probability and backoff.

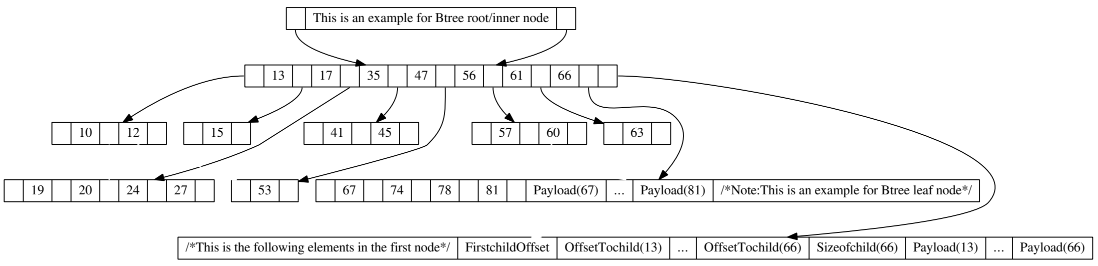


Figure 3.6: The layout of a Btree_trie, which is a general node in trie. The bottom grids are the following elements of the first node, drawn like this for layout. The data in a Btree_trie are arranged in a form showed in the bottom grid (the leaf node has no offset to child). The reason why we just store offset but not address is to save space. This figure is inspired by (Bogoychev and Lopez, 2016)

- For higher order gram, the store is the same as bigram. Only the highest gram does have no offset to next_Btree_level and no backoff parameter.

After that, the words stored in it will be mapped to a number, called vocabID. This language model will participant the next step to make queryable sentences.

3.4.2 Transfer input sentences on CPU

The original input sentences, $w_1, w_2, ...w_l$, can not be used directly to perform query on GPU. We fetch the pieces w_{i-n+1}^{i-1} for all $i \in (1...l)$ and all $n \in (2...N)$ from them as the condition for the conditional probability $P(w_i|w_{i-N+1}^{i-1})$ in function 3.1, where N is greater than 2 [6]. When GPU take w_{i-N+1}^{i-1}, it will assign a block for each one queryable sentence in w_{i-N+1}^{i-1}. Besides, we have to replace the actual word with a number called vocabID in terms of the root trie in language model we built in last section. vocabID will be used to find the associated value in data structure. This process happens on CPU.

3.4.3 Copy data from CPU to GPU

After we having data structure (section 3.4.1) and queryable sentence (section 3.4.2), we need to allocate the memory of GPU for all the data that we will copy from CPU to GPU. One important thing is we have to copy a pointer pointing to an address to store the query results from CPU to GPU and set the memory allocated to results to be zero because GPU will not initialize the value for second query [7]. Finally, we copy the language model and queryable sentences from CPU to GPU and perform query.

3.4.4 Query on GPU

After accepting the data from CPU, we do query on GPU. The process of query was described in section 3.3. Let N be the highest order of language model. When we run the program on GPU, it will call kernel to execute the commend. Meanwhile, we have to assign appropriate size of Grid_size and Block_size. The size of grid can be simply regarded as the number of blocks in one grid. Similarly, the size of block can be regarded as the number of threads in one block. These assignment of these two numbers is showed in 3.1.

Grid_size	Block_size
num_ngram_queries	max_num_children

Table 3.1: The value of Gridsize and blocksize. The max_ngram_queries is the total number of queryable sentences. The max_num_children, which has been set building the language model, is the max number of child nodes in Btree_trie

Let us take one queryable sentence, w_2^1, as an example to begin the journey of querying on GPU.

[6] Because if N < 2, there is no probability for bigram or other following word

[7] If we do not do this, the second query may return the results of last query because GPU will not erase the value even though we free them.

3.4.4.1 Parallel search the query sentence

At first, we will fetch the value of w_1 from root tire.As we described in section 3.3.2.1, this process does not need parallel searching because the array V can be indexed in fixed time by vocabID.

Then, the payloads of w_1 contains the address to a child trie node of w_1, where w_2 stores. Because this trie node is actually a Btree_trie in the form of Figure 3.5, vocabID of w_2 will be one of these number. Then, all of the threads i will work at the same time to get all the vocabIDs in root Btree_trie node. Similar to binary search, after all threads finding vocabID[i], K-ary search is making comparison with the vocabID[w_2]. If vocabID[w_2] = vocabID[i], then fetch the address to next trie_level. If vocabID[w_2] $>$ vocabID[i], fetch the address to next Btree_trie_level which is located in the left of vocabID[i]. If vocabID[w_2] $<$ vocabID[i], fetch the address to next Btree_trie_level which is located in the right of vocabID[i]. This is a recursive process until find all the words in the query sentence. Whenever the word is not find in trie, we will not forward to next step and just end this query. Zero is the default answer for this situation [8].

3.4.4.2 Return an entire node of trie (Btree_trie)

This is a process run by threads in one block. After finding the last word in the query sentence, then we have got the address to next trie_level from w_{i-N+1}^{i-1}. To get $P(w_i = X|w_{i-N+1}^{i-1})$, the next step is to return the whole node of trie, because all the words following w_{i-N+1}^{i-1} are stored in this node.

This process is divided into two steps in terms of requirements. The first step is to fetch all the vocabID, and associated probabilities stored in root (Red part in Figure 3.7). Then store all the probabilities into results indexed by $F(blockIdx.x, vocabIDs)$. [9]. The second step is to fetch the offset stored in root Btree_trie. If the offset is not zero, then we can find the child Btree_trie node. Each thread will be responsible for the return of a subBtree_trie (Different color represents different thread in Figure 3.7)[10]. And then for each subtree_trie, the thread will traversal the whole subBtree_trie

[8] Since we can not find w_1 or w_2, it is impossible to find a word following w_1, w_2, so the probability of it is zero.

[9] F(blockIdx.x, vocabIDs) = (blockIdx.x*number of unigram) + vocabIDs. Each block is responsible for one queryable sentence, so the data stored in results array will be start with bigram conditional probabilities, then trigram ...: $P(X|w_{i-1})$,$P(X|w_{i-1}, w_{i-2})$,...,$P(X|w_{i-N+1}^{i-1})$ for all $i \in (1,...l)$. Where X represents all the possible words

[10] The number of threads is equal to the max number of Btree_trie children nodes

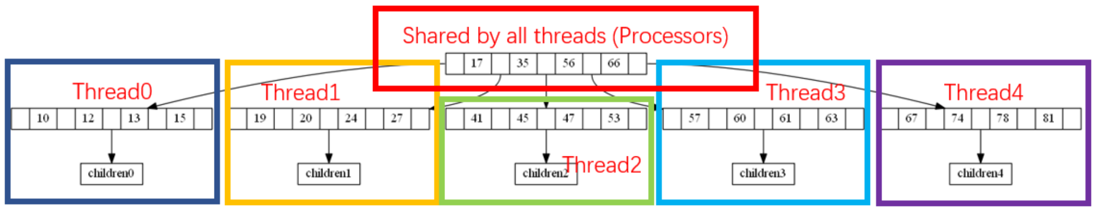


Figure 3.7: The root Btree_trie will be processed by all the threads (max_num_children-1 exactly), and the children of this root Btree_trie will be assigned to different threads. This picture is inspired by a tutorial designed for parallel course (Weeden and Royal, 2017)

and return the values.

The traversal Algorithms can be included using the commands as shown in algorithm 1.

Algorithm 1 Traversal algorithm

```
1: procedure SUBBTREE_TRIE_TRAVERSAL(address to Btree_trie node)
2:     if address to Btree_trie node !=0 then
3:
4:         for $i = 1$, $i$++, while $i$<size of Btree_trie node do
5:             result[F(blockIdx.x, vocabIDs[i])]=value(vocabID[i])
6:             address to Btree_trie node[i]=address_to_child(vocabID[i])
7:
8:
9:         for $i = 1$, $i$++, while $i <$ max number of children do
10:            subBtree_trie_Traversal(address to Btree_trie node[i])
11:
```

3.4.4.3 Batch queries

As we described in previous section, there will be different n-gram queries made on CPU in terms of the query sentence. One block is responsible for one queryable sentence. Batch queries will be run by many blocks on GPU. Simplify the process, no matter how many sentences are inputted, we just have many queryable sentences after the input sentences are transferred. When the number of queries reaching a point, then this process will utilize all the compute power of GPU, which is called saturating GPU. The experiments to saturate GPU run in section 4 will need batch queries.

Chapter 4

Experiments

Get busy living, Or get busy dying.

—-*The Shawshank Redemption*

To achieve the sub-objective 5 mentioned in chapter 1, this chapter describe the experiments designed to test the performance of giLM. The experiments are divided into two main parts. The first part was designed to test whether the results are correct and the same as what we expected form (section 3.1). The second part was designed to test the performance of giLM including query speed and the robust to the different language model data set. Because there is no existing language model designed for the same purpose of giLM, we compare it with an GPU language model, gLM. The results and analyses are in the next chapter.

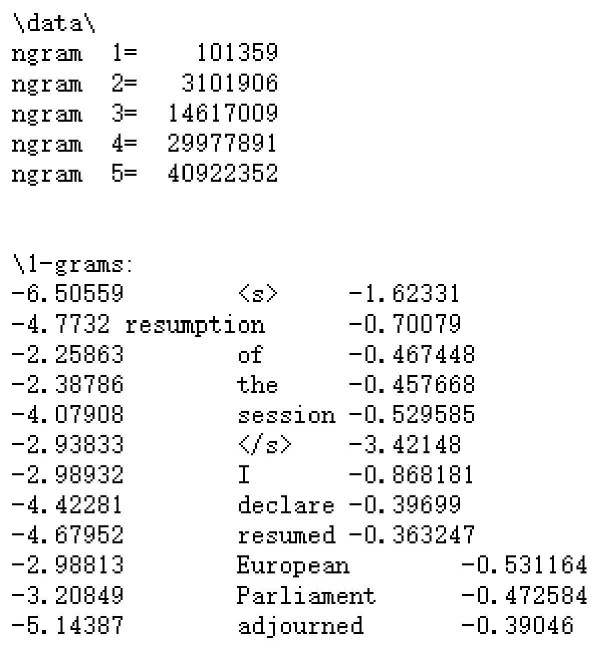
4.1 Resource and environment

4.1.1 Environment

Table 4.1: The experiment environment

GPU	Tesla K80
Programming language	C++
Platform	NVIDIA CUDA8.0
Operation system	Ubuntu 16.4

We did the experiments on a VM instance installed 16.04.2 LTS system on Google Cloud Platform. This VM instance is assigned a NVIDIA Tesla K80, the most popular



```
\data\
ngram  1=    101359
ngram  2=   3101906
ngram  3=  14617009
ngram  4=  29977891
ngram  5=  40922352


\1-grams:
-6.50559        <s>     -1.62331
-4.7732 resumption      -0.70079
-2.25863        of      -0.467448
-2.38786        the     -0.457668
-4.07908        session -0.529585
-2.93833        </s>    -3.42148
-2.98932        I       -0.868181
-4.42281        declare -0.39699
-4.67952        resumed -0.363247
-2.98813        European        -0.531164
-3.20849        Parliament      -0.472584
-5.14387        adjourned       -0.39046
```

Figure 4.1: A glance of europarl.lm.1. The form of data is: $\hat{P}$, words, β

GPU to build data center and also a major accelerator for all major Deep Learning frameworks [1].

4.1.2 Dataset

Except where noted, we use europarl.lm.1 (Figure 4.1 shows a piece of this file), a 5-gram language model (used by gLM to do test) [2] to saturating the GPU. It is 3.3G and contains more than 88.7 million n-grams where the number of unigram is 101359.

Furthermore, we made three small different ngram language models whose order are 3, 4, 5 respectfully taking a piece of Europarl v7 [3] with the help of KenLM toolkit. We also make the comparison between different model size by using europarl.lm.1 and a 5-gram model containing 23120 n-gram (1350 unigrams), which was one of the three small different ngram language models made by myself.

4.2 Correctness

The output of this model is a big result array where all the possible conditional probabilities in matrix 3.1 store in. Because the unigram probability can be accessed

[1] http://www.nvidia.com/object/tesla-k80.html

[2] http://www.statmt.org/moses/RELEASE-3.0/models/fr-en/lm/europarl.lm.1

[3] http://www.statmt.org/wmt14/translation-task.html#download

directly, the probability of unigram is not contained in this model. The layout in results array is showed below:

For example: given input sentence is w_1^l, then:

$$\text{results}[\underbrace{P(w_i = X|w_{i-1})}_{\text{For all } i \in (1...l)}, \underbrace{P(w_i = X|w_{i-1}, w_{i-2})}_{\text{For all } i \in (1...l)}, ..., \underbrace{P(w_i = X|w_{i-N+1}^{i-1})}_{\text{For all } i \in (1...l)}]$$

To verify whether the results are correct, we sample a few query sentences from website and Europarl v7, and query them on giLM. To verify the results, we checked them by hand one by one.

4.3 Performance test

To test the performance of giLM, we designed a series of experiments and made comparison with gLM, the first GPU language model. Because gLM has a computation cost of backoff computation, which is not required in our implementation. However, to make this experiment fairer, We modify our implementation giLM to be a giLM with a simulated backoff computation[4] in order to reduce the effect of unfair factor. All the giLM performed in experiments is with backoff computation except where noted.

The B-tree node size is 31 which is a default setting in gLM. The metrics used in this test shows below:

- **Throughput** Throughput (N-gram queries per second) was used in gLM's paper to measure the query speed, excluding the cost of initializing and copying data structure, and moving data between GPU and CPU.
- **Throughput_output** Throughput_output (number of output per second) is very similar to throughput but using output numbers per second as a measurement.
- **Cost-effective(CE)** We presented a metric called Cost-effective(CE). Because these two GPU language model return different results for different purposes, we need to compare them by measuring how much we have to spend to reach a common goal using these two GPU language models respectfully. The cost here includes the cost of initializing queries and moving data between GPU and CPU.

This part of experiments is to test the performance of giLM including query speed and the robustness to the different language model data set. Therefore, according to the implementation of giLM and experiments did by gLM, we presented six hypotheses:

[4] Whenever we query a sentence which is not find in model, instead of ending search directly, we perform another search from root node to do second search and also fetch β, which add an extra cost

1. For batch query (many input sentences under saturating the GPU), gLM should outperform giLM according to throughput (N-gram queries per second).

2. For batch query (many input sentences under saturating the GPU), giLM should outperform gLM according to throughput_output (number of output per second).

3. For and batch query (many input sentences under saturating the GPU), giLM should outperform gLM according to Cost-effective(CE).

4. The backoff computation in giLM should not affect the performance of giLM because most of the time cost should be on traversal process.

5. The effect of N-gram order and language model size on giLM should be similar to gLM because they have similar data structure (giLM replaces reverse trie in gLM with forward trie)

6. The process writing data to result data in GPU global memory will take up the main query time because the GPU global memory access is very *expensive* (Table 2.1).

Based on the hypotheses mentioned above, we design the following experiments to verify these hypotheses.

4.3.1 Saturating the GPU

To compare the performance between two GPU language models, we should make the comparison under the situation in which both of them are saturating GPU, because they have different GPU utilization rate even though the input query sentences are same. Therefore, we gradually increase the number of query sentences, and then make use of them to do investigation. Finally, the variation tendency of throughout is shown in Figure 5.1 and Figure 5.2 in next chapter.

When these two language models are saturating the GPU, they get the best performance they could achieve on the same hardware. Then we analysis the results of their best performance to verify the hypotheses 1,2,3.

To verify the hypothesis 4, we perform a *pure* giLM without backoff to see the effect of backoff computation.

4.3.2 Effect of N-gram order and language model size on performance

To explore the effect of N-gram order of language model and the effect of language model size, we perform the experiments *saturating the GPU* on different N-gram order language models. Besides, one of it is 5-gram order and it is smaller than the data set used in last experiment, so we can compare it with the experiment results get from last experiment. Because giLM is based on gLM, and the data structure of them are very similar, giLM is expected to have similar variation tendency when we change the language model.

4.3.3 Runtime experiment for each step in Query

This was designed for hypothesis 6. As we talked in chapter 3, there are two steps in Query process:

- Search the queryable sentence
- Return all the probabilities stored in trie node

For second step, we at first fetch data from data structure and then write the data we fetched from data structure into result array. However, result array is allocated to GPU global memory, so the *write* operation will needs much time according to Table 2.1. Therefore, these three steps are responsible for the main time cost. We want to explore the time cost of each step to know which step is the one we need to optimize at first.

To do this, we modify the code of giLM:

- Only search the queryable sentence
- Only fetch all the probabilities stored in trie node but not write them into result array
- Normal one.

The results of this experiment will be discussed in chapter 5.

Chapter 5

Results and Evaluation

It always seems impossible until it's done.

—*-Nelson Mandela*

With the two experiments in mind, we analysis their results in this chapter. We discussed the hypotheses mentioned in last section.

5.1 The analysis of Correctness

As we talked about in chapter 4. The probabilities returned by giLM are same with the values stored in language model for each query.

$$\text{results}[\underbrace{P(w_i = X|w_{i-1})}_{\text{For all } i \in (1...l)}, \underbrace{P(w_i = X|w_{i-1}, w_{i-2})}_{\text{For all } i \in (1...l)},...,\underbrace{P(w_i = X|w_{i-N+1}^{i-1})}_{\text{For all } i \in (1...l)}] \tag{5.1}$$

Compare the output showed in equation 5.1 with matrix 3.1, we find all the elements in matrix we need can be found in this results array apart form the unigram probabilities[1]. Therefore, This project can satisfy the requirements of MODLMs.

However, because of the limited time, we did not test the results automatically by program, which will be a future work to do.

Table 5.1: Throughput comparison (N-gram queries per second) between gLM, giLM and giLM without backoff computation.

	Europarl.lm.1	small(5-gram)
giLM	0.36M	27.90M
gLM	13.08M	21.45M
giLM-without backoff	0.35M	38.56M

5.2 The analysis of performance test

Because gLM has an backoff computation model, we measure query speed with or without the extra backoff computation in giLM. Therefore, we gradually increase the number of query sentences, and then make use of them to do investigation. Finally, the variation tendency of throughout is shown in Figure 5.1 and Figure 5.2.

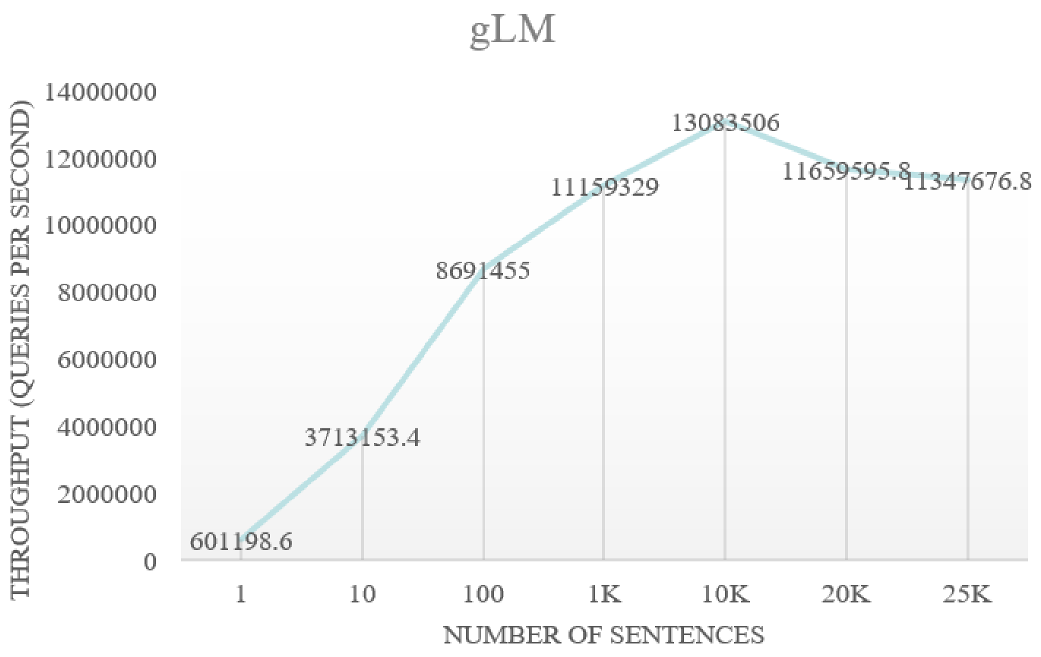


Figure 5.1: Throughput against various number of sentences for gLM.

With the hypotheses raised in last chapter, we analysis them one by one according to the results of experiments.

[1] The unigram probability can be fetched directly from language model without GPU. It was talked in section 3.3.2.1

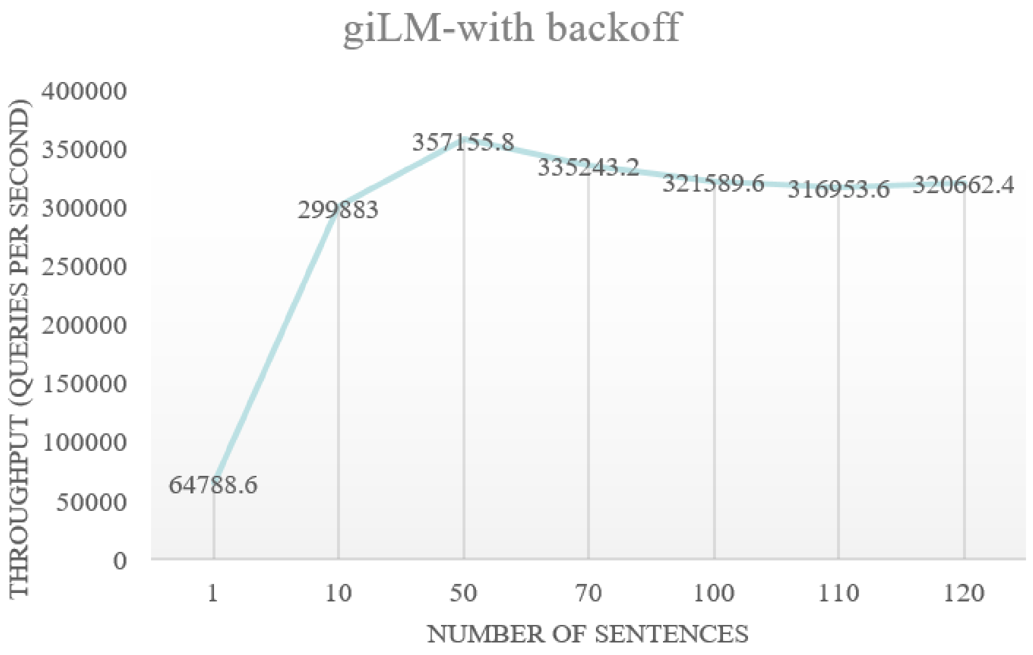


Figure 5.2: Throughput against various number of sentences for giLM.

Table 5.2: How to get throughput_output(number of output per second) of gLM and giLM using Throughput (N-gram queries per second). Let s be the number of unigram in the language model

giLM	Throughput_output(giLM)=Throughput(giLM)*s
gLM	Throughput_output(gLM)=Throughput(gLM)

For batch query (many input sentences under saturating the GPU), gLM should outperform giLM according to throughput (N-gram queries per second).

The Figure 5.2 depicts giLM saturating the GPU when the query sentences is more than 50. Throughput of giLM stay at 0.36M and keep relatively steady or even slightly lower, which may be caused by the complexity of query sentence which increase the cost of search on GPU, as the increasing size of query beyond this point. In contrast, the throughput of gLM (Figure 5.1) reach the peak when query sentences are more than 10K. From these two graphs and Table 5.1, throughput of gLM is bigger than giLM when both of them saturating the GPU. However, giLM can get a better throughput than gLM's on small language model. This is different from our hypothesis.

We already analyzed the query process of gLM and giLM. One possible reason giLM outperform than gLM in terms of throughput is there is little data to be fetched, so the time cost in traversal is so little that giLM outperforms gLM. Another possible reason is that the simulated backoff computation in giLM is not enough to make up the cost of backoff computation in gLM though it really works (Table 5.1 and this part will be discussed later again).

If we change to another angel, size of output, to analyze this phenomenon. Because of the limited GPU memory, where the results are stored in, throughput_output of language model will be limited too (the results has to be allocated by GPU memory). According to table 5.2, the throughput of giLM should be less than gLM when the throughput_output is given. However, this conclusion is based on the throughput only affected by throughput_output. Apart from the different effect of GPU memory on gLM and giLM, the data processed by them, the searching process and the computation (giLM only has a simulated computation), are not the same. The only conclusion can be draw here is when both of gLM and giLM saturate the GPU, if the throughputs of them can be get, then we can know the relationship between throughput_outputs of them.

Though we have many speculating, We still can not make a clear conclusion for the comparison between gLM and giLM over throughput. Therefore, the first hypothesis false.

For batch query (many input sentences under saturating the GPU), giLM should outperform gLM according to throughput_output (number of output per second).

According to table 5.2, we can get:
throughput_output(giLM) = 0.36M*101359 = 3648.92M for europarl.lm.l
throughput_output(gLM) = 13.08M for europarl.lm.l
throughput_output(giLM) = 27.90M*1350 = 37665M for small(5-gram)
throughput_output(gLM) = 24.45M for small(5-gram)
From the computation results showed above, the giLM takes absolute predominance both on two data sets on throughput_output. Therefore, the second hypothesis is verified to be correct.

For and batch query (many input sentences under saturating the GPU), giLM should outperform gLM according to Cost-effective(CE).

Let us take CE as our matrix. In terms of the problem analysis in section 3, we need an array of number where all the conditional probabilities store. For gLM, firstly, it returns only one number which is not what we want. Secondly, even though we modify the source code of gLM to return a number of separate conditional probabilities

$P(w_i|w_{i-1},...,w_{i-N+1})$, it just return (q-1) results, which is far smaller than the results of giLM.

For example in this experiment (s=101359):

$$0.36M * 101359 = 3648.92M$$

To get the same size of results as giLM, for gLM,

$$3648.92M/13.08M = 278.97.$$

The computation above shows the actual return of giLM is 3648.92M, but gLM is only 13.08M. To achieve the same results, gLM needs to spend 278.91 times the time than giLM even ignoring the other cost like we mentioned in next paragraph. Therefore, in this condition, giLM is far better than gLM.

For gLM, it does not have the ability to initialize queries for the problem in section 3 so we need to prepare special query sentence or modify the source code of gLM to do the same thing. And for giLM to return s results, it only need to copy one query from CPU to GPU, but gLM needs to do s times.

The results on small dataset shows similar results to the big dataset.

Therefore the third hypothesis can be verified to be correct.

The backoff computation in giLM should not affect the performance of giLM because most of the time cost should be on traversal process

We delete the backoff computation model in giLM to observe the *pure model* not for performance comparison. We hypothesized giLM-without backoff should be better than giLM. We observe it has no advantage on big language model (The result is slightly worse than giLM because of different initialization of data structure, this is an adaptively error) but outperform gLM on small language model. This may be due to the cost of backoff computation in giLM can be ignored on big language model, where most of the cost of giLM afforded by the traversal described in section 3.4.4.2. Another reason is due to the *B-stump* (Bogoychev and Lopez, 2016), more than 88% N-gram (where N $>$ 2) only have a root Btree_trie if the K is appropriate, which means mostly only one parallel search will fetch all the data we need. Therefore, when the dataset is very small, most of trie node will be *B-stump*, which is a possible reason why giLM perform so good on small dataset.

Therefore, the forth hypothesis not totally correct, it depends on the size of language model.

The effect of N-gram order and language model size on giLM should be similar to gLM because they have similar data structure (giLM replaces reverse trie in gLM with forward trie)

Table 5.3: Throughput comparison (N-gram queries per second) of giLM on different n-gram language models

	5	4	3
Europarl.lm.1	0.36M	0.27M	0.18M
small language model [2]	30.00M	28.38M	22.70M

- **Order of N-gram language model** Bogoychev and Lopez (2016) offered their conclusion and the paper says the throughput of gLM will increase along with the decrease of n-gram. However, giLM has a opposite results with gLM (Figure 5.3). At first, we thought it may be affected by the model size, so we did the same experiment on small dataset and the results are showed in Table 5.3. Therefore, the decrease of throughput is not because of the language model size. Theoretically, lower order leads less queryable sentence, the query time should be less. However, the results of experiment is different from theoretical the guess. This will be a future work to analyze.
- **Different language model size** Table 5.1 and Table 5.3 shows that the effect of model size has a more effect on giLM than gLM. Because as the increasing of language model size, giLM and gLM both bothered by the increasing complexity of vocabulary search and associated computation. However, giLM has an extra traversal for the vocabularies whose number is equal to the number of words in unigram and has to copy all of the data (size is the number of unigram) to CPU. Therefore, model size influences giLM more than gLM on throughput. And along with the increase of model size, the throughput will decrease.

The process writing data to result data in GPU global memory will take up the main query time because the GPU global memory access is very *expensive* (Table 2.1).

As Table 5.4 shows, for big language model, the traversal process slow the process from *only search* by 96% for big data, 87% for small data. And the write data process also slow the process from *no write* by 95% for big data,33% for small data.

Table 5.4: Throughput comparison for different Query step

	only search	no write	normal
Europarl.lm.1	200.24M	7.95M	0.36M
small language model	353.65M	45.25M	30.00M

From the computation above, the traversal process accounts for much time of the whole query process. The process to write data into GPU global memory is relatively less serious than traversal. This may be due to the there is not too much data stored in language model need to be fetched and return. This conclusion can be ensure by the comparison between big data set and small one. Because small data structure store less data, and the requirement to write data into result array is less than big one, write process accounts for time cost on big dataset more than on small one. In contrast, the traversal process accounts for a similar percent both on two data set.

Therefore, the hypothesis 6 is not right. Traversal accounts for most time cost. However, if we do not execute data writing, the limitation of GPU global memory will disappear because no write operation on GPU global memory. Then the query throughput can be double times than *no write*, but it is not useful in our project.

5.3 Summary

5.3.1 The verification of hypotheses

The hypotheses raised in chapter 4 have been talked in this chapter. Hypothesis 2 and 3 are verified to be correct. Hypothesis 1 can not be ensure by this experiment. The validity of hypothesis 4 depends on the size of language model. And hypothesis 5 have to be discussed into two parts: one of them false but another one is partly correct (the size has more effect on giLM than on gLM). Hypothesis 6 is wrong but we can get the information about time cost of each query step from this experiment, which inspires us which part should to be optimized.

5.3.2 Evaluation to experiments

- The design of experiment is not comprehensive to evaluate giLM. Because there is no existing implementation for MODLMs, we need to *make* a implementation to be reference. This problem can be solve by modifying gLM to return the same results and compare performance of two different implementations.

- We only can know giLM can satisfy the requirements of MODLMs, but the information get from experiment for performance is not enough.

5.3.3 Problems and suggested solutions

Problem 1 The first experiment for correctness has been done and checked by hand. It is not convenient and not enough to test all the possible situation.

Solution 1 Potential solution to this problem is write program to run this test automatically including the query sentence sampling and the results verifying.

Problem 2 Because the limited memory of GPU, the query file is also limited by it. If the query file is too big, there will be memory error.

Solution 2 There could be a program on CPU which can split the query file automatically according to the condition of GPU.

Chapter 6

Conclusion and Future work

Hope is being able to see there is light despite all of the darkness

—-Desmond Tutu

6.1 Conclusion

This thesis firstly introduced a language model framework, MODLMs (Neubig and Dyer, 2016), and the first GPU language model gLM (Bogoychev and Lopez, 2016). Based on the implementation of gLM, this thesis contributes:

- A forward trie
- A data analysis algorithm on CPU
- A parallel traversal algorithm on GPU

Finally, it builds a queryable language model on GPU for MODLMs. From the results of experiments, giLM can satisfy the requirements of MODLMs. Compared with existing GPU language model, it outperforms over the requirements of MODLMs.

This project is expected to contribute a efficient queryable language model on GPU for MODLMs and as a foundation to improve the applications where MODLMs can be applied, such as neural machine translation.

6.2 Potential future works

Due to the time constraint, there are still too many things can be explored in this project:

- As we talked in chapter 5, there are some problems in the experiments. The potential solutions could be executed in the future.
- Because the result is allocated on GPU global memory. The writing operation to it (section 3.4.2) is inefficient especially when Btree_trie is very deep. This could be a point to improve the performance of giLM
- Since giLM is a foundation of MODLMs, integrating giLM into MODLMs will be a future work. After that, we can try to compare the original MODLMs and giLM-MODLMs and the other applications of it.
- Because we do not need to do the backoff calculation, we do not need to store them into our data structure. If we omit the store of backoff parameter, the size of data structure can be smaller.
- As the experiment shows, traversal accounts for most time cost. Because the existing implementation in giLM is recursive function, which is inefficient. This part could be improved in the foreseeable future.
- Since there is no existing language model designed for MODLMs, we could modify the code of gLM to let it produce the same results as giLM. Then we can compare the performance between two GPU language model, both of which can do the same thing.

Bibliography

Ammar, W., Mulcaire, G., Ballesteros, M., Dyer, C., and Smith, N. A. (2016). Many languages, one parser. *arXiv preprint arXiv:1602.01595*.

Bayer, R. and McCreight, E. (2002). Organization and maintenance of large ordered indexes. In *Software pioneers*, pages 245–262. Springer.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Bogoychev, N. and Lopez, A. (2016). N-gram language models for massively parallel devices. In *ACL (1)*.

Federico, M., Bertoldi, N., and Cettolo, M. (2008). Irstlm: an open source toolkit for handling large scale language models. In *Interspeech*, pages 1618–1621.

Guthrie, D. and Hepple, M. (2010). Storing the web in memory: Space efficient language models with constant time retrieval. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 262–272. Association for Computational Linguistics.

Heafield, K. (2011). Kenlm: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 187–197. Association for Computational Linguistics.

Heafield, K. (2013). *Efficient Language Modeling Algorithms with Applications to Statistical Machine Translation*. PhD thesis, University of Edinburgh.

Kaufmann, M. M. and Boston, G. (2011). Computing gems emerald edition (applications of gpu computing series).

Kneser, R. and Ney, H. (1995). Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181–184. IEEE.

Koehn, P. and Haddow, B. (2012). Towards effective use of training data in statistical machine translation. In *Proceedings of the Seventh Workshop on Statistical Machine Translation*, pages 317–321. Association for Computational Linguistics.

Mikolov, T., Karafiát, M., Burget, L., Cernockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Interspeech*, volume 2, page 3.

Neubig, G. and Dyer, C. (2016). Generalizing and hybridizing count-based and neural language models. *arXiv preprint arXiv:1606.00499*.

Nvidia (2017). Cuda c programming guide.

Pauls, A. and Klein, D. (2011). Faster and smaller n-gram language models. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 258–267. Association for Computational Linguistics.

Richardson, J., Dabre, R., Chu, C., Cromières, F., Nakazawa, T., and Kurohashi, S. (2015). Kyotoebmt system description for the 2nd workshop on asian translation. In *Proceedings of the 2nd Workshop on Asian Translation (WAT2015)*, pages 54–60.

Schlegel, B., Gemulla, R., and Lehner, W. (2009). K-ary search on modern processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 52–60. ACM.

Spence Green, D. C. and Manning, C. D. (2014). Phrasal: A toolkit for new directions in statistical machine translation. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 114–121.

Trevett, N. (2013). Opencl introduction. *Khronos Group*.

Weeden, A. and Royal, P. (2017). Parallelization: Binary tree traversal.

.