Parallelizing a Statistical Machine Translator

CS267 Spring 2010

Chao-Yue Lai

I. Introduction

Machine translation (MT) is one of the classic problems in computer science and a vast area of research in the field of natural language processing (NLP). High-quality and fast MT enables a variety of exciting applications, such as real-time translation in foreign environments on handheld devices as well as defense and surveillance applications. The massive amount of documents on the Internet, which increases second by second, also demands the existence of fast machine translators so that people speaking different languages can share these resources.

However, state-of-the-art machine translators take considerable time to perform the task. It takes on average a minute to translate a 50-word sentence to ensure the quality of the outcomes. This is due to the huge dictionary-like statistical models and the complicated algorithms inherent in the translation process, which will be explained in the following sections.

Our goal is to make MT faster and scalable. State-of-the-art machine translators, such as Google Translate, employ a cluster of machines and their MapReduce framework[1] to speed up the process. We, on the other hand, will parallelize MT on GPUs by exploiting data parallelism in the statistical models. This is both scalable and available to every person.

II. Overview of a Statistical Machine Translator

The problem of MT is as follows: given a sentence in a source language, the goal is to produce its most likely translation in the target language. MT systems can be divided into two categories: statistical. Statistical systems use information gathered from a large set of written documents using statistical learning techniques, while rule-based systems apply grammar and lexical rules in the process. We will focus on statistical systems, the current mainstream in NLP.

A high level structure of an MT system is shown in Figure 1. In order to translate the sentence, the machine translator consults a language model and a translation model. The language model represents statistical knowledge about the syntactic and semantic structure of the target language, while the translation model represents the mappings of words or phrases from source to target language.
Figure 1: A machine translation system

The translation model records the mappings of phrases from the foreign language to the target language with associated probabilities. For example, several translations are possible for the French phrase C’est la vie. In the translation model, the entries concerning C’est la vie can be $P(\text{This is life}|\text{C’est la vie}) = 50\%$ and $P(\text{This is awful}|\text{C’est la vie}) = 10\%$, which means it can most likely be translated as This is life.

The N-gram language model calculates the probability that a certain word follows previous (N - 1) words. For example, in the trigram model, probabilities of 3-word phrases are collected. Entries can be $P(\text{way}|\text{by the}) = 30\%$, which means after by the, 30% of chance is the word way. N can vary from 2 to 10, and the quality of translation improves as N increases. Nonetheless, the size of the language model grows exponentially with N increases. Therefore, we always have to find a balance between solution quality and memory/runtime requirements in an MT system.

III. Algorithms

The specific algorithm we use is called the CKY algorithm[2][3], which is essentially a dynamic programming algorithm also used in language parse. The CKY algorithm is a 3-step procedure applying translation and language models onto sentences. Here I will use the German sentence Ich mag Äpfel. to illustrate the translation process.

Given a pre-segmented sentence in the source language, the first step is to find translations of phrases in the translation model. In Figure 2, a triangular chart represents the translation process of the four-word sentence Ich mag Äpfel. (the period is a “word” in the system). Grid 1 stores translations for Ich mag Äpfel while grid 2 is for mag Äpfel. The lower parts of the chart are filled with translations and associated probabilities according to the translation model. For example, the grid corresponding to the sentence segment Ich mag can have entries (I like : 22%) and (I love : 18%) and so on, depending on the size of the translation model.
The second step is to combine the translations and fill up the chart using the language model in a bottom-up fashion. Once we have translations of short sentence segments in place, we can concatenate them and fill up upper grids using results in the lower grids. For example in Figure 3, to generate translations of *Ich mag Äpfel*, we combine translations of two split positions, which are *Ich mag* and *Äpfel* corresponding to the red arrows, and also results for *Ich* and *mag Äpfel*, which are the green arrows. The combinations require the N-gram language model to calculate the probabilities, as the blue bow arrow shows. In general, upper grids need results from lower grids, so a bottom-up process is necessary. There can be thousands of entries per grid, which presents opportunity for parallelization, especially for upper grids.
Finally, the most likely translation of the whole sentence is extracted with a top-down traversal of the chart, as shown in Figure 4. After the whole triangular chart is filled up, the entry with the high probability in the top grid is targeted, and we search through all combinations of translations of different split positions to find the one pair which generates the same probability. This is a recursive process until it reaches the bottom level.

The bottleneck of this algorithm is the second step. Although this process sounds time-consuming, it is actually lightweight comparing to the second step. We will try to parallelize the second step via utilizing the concurrency inherent in the process.

IV. Parallelization Strategy

Opportunities for parallelism abound in the second step. In computing a grid, combinations corresponding to different split positions are independent. Furthermore, inside a combination, there can be thousands of right translations and left translations. Each combination is independent, which creates a million-way parallelism. Thus, this seems a perfect fit for SIMD architectures like GPUs.

However, two factors hinder the speed of GPU. The first is the unrestricted number of translations per grid. A grid can have up to 6000 translations, which require too much memory for manycore systems to handle. We restrict the number of translations in two ways. First, we set a threshold and only reserve the translations above the threshold. This is called the beam search and the threshold is referred as a beam. Second, we limit the number of translations per grid to a fixed number 1024. Experiments show that this restriction gives little or no loss of the BLEU score[4], a
de facto standard for evaluating quality of translations.

Another hindrance is the accesses to the language model. These accesses are frequent and inevitable, but they slow down the application, not only because can the language model only be stored in the global memory in a GPU due to its enormous size, but also in that they are uncoalesced and irregular. We reduced these accesses by storing some needed entries in the local memory and shared memory beforehand. We are trying to restructure the language model to have a more balanced memory access pattern.

V. Results

This is a research project conducted by me and Ekaterina Gonina from Par Lab. The NLP group in Berkeley, which is led by Professor Dan Klein, gave us a simple CKY-based machine translator for parallelization. We transferred the code from Java to C/C++ and then CUDA. The transfer from Java to C/C++ was done by both of us, where I was responsible for the first and third steps and she was for the second one. After that, I parallelized the second step by myself.

We experimented the machine translator on 1000 sentences from Spanish to English with an average length of 28 words. The CUDA code was experimented on a 1.30GHz NVIDIA GTX280 GPU and also on a 1.40GHz NVIDIA GTX480 Fermi GPU, while the serial code ran on 2.83GHz Intel Core 2 Quad CPU. I also did a comparison of an OpenMP parallelization with 4 threads on a 2.67GHz Intel Nehalem CPU. The runtimes are shown in Table 1. As we can see, the naïve parallelization on Tesla GPU is even slower than the serial version, while Fermi does better because the number of cores is doubled and it handles irregular memory accesses better. Nonetheless, OpenMP parallelization performs even better, which means we still have to further optimize our CUDA code.

<table>
<thead>
<tr>
<th></th>
<th>Serial</th>
<th>CUDA(Tesla)</th>
<th>CUDA(Fermi)</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime(secs)</td>
<td>235.7</td>
<td>270.5</td>
<td>132.0</td>
<td>101.2</td>
</tr>
<tr>
<td>Speedup</td>
<td>1x</td>
<td>0.9x</td>
<td>1.8x</td>
<td>2.3x</td>
</tr>
</tbody>
</table>

*Table 1: Runtimes for 1000 sentences with different parallelization schemes*

Longer sentences have more split positions, which provide more concurrency and should facilitate SIMD parallelization. The same experiment was also conducted on 350 sentences with over 40 words to show GPU performs better given more concurrency. As shown in Table 2, the speedup for CUDA is better than speedup for shorter sentences shown in Table 1. OpenMP parallelization still outperforms CUDA code on Fermi, which means we still have a long way to go.
Table 2: Runtimes for 350 sentences with more than 40 words

<table>
<thead>
<tr>
<th></th>
<th>Serial</th>
<th>CUDA(Tesla)</th>
<th>CUDA(Fermi)</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime(secs)</td>
<td>281.3</td>
<td>238.0</td>
<td>124.3</td>
<td>108.3</td>
</tr>
<tr>
<td>Speedup</td>
<td>1x</td>
<td>1.2x</td>
<td>2.3x</td>
<td>2.6x</td>
</tr>
</tbody>
</table>

VI. Conclusion and Future Work

Machine translation is a promising application worthy of parallelization. It has plenty of concurrency to be utilized, yet the irregular memory accesses hampered the performance. The initial parallelization shows limited amount of speedup, but we will restructure the N-gram language model to comply with GPU architecture. We believe improvements can be achieved based on the previous success of speech recognition in our group[5], where a similar N-gram language model is used.

The NLP group in Berkeley also has great achievements in machine translation[6]. We plan to collaborate with them and parallelize their Berkeley Translator after the parallelization of a simple machine translator reaches as reasonable speedup. In the long run, we plan to develop an application framework to help NLP experts parallelize their application with ease.

References


