A Fixed-Point Decoding Approach for Statistical Machine Translation on Mobile Terminals

Xiang Li, Jin'an Xu
School of Computer and Information Technology
Beijing Jiaotong University
Beijing, China
{06281013, jaxu}@bjtu.edu.cn

Wenbin Jiang, Qun Liu, Yajuan Lü
Key Laboratory of Intelligent Information Processing
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
{jiangwenbin, liuqun, lvyajuan}@ict.ac.cn

***Abstract*—The demand for statistical machine translation on mobile terminals is increasing rapidly, but translation speed is restricted by the embedded processors without a floating-point unit. This paper proposes an approach to convert floating-point numbers into fixed-point numbers for SMT decoding on mobile terminals in order to reduce the impact of the processors without a floating-point unit on translation speed. The experiments based on PC and mobile terminal show that this approach ensures the quality of translation and the speed of fixed-point arithmetic operations is 135.6% faster than that of floating-point arithmetic operations. Therefore, this approach can efficiently improve translation speed of SMT systems on mobile terminals with weak ability in floating-point arithmetic operations.**

key words: SMT, fixed-point, mobile terminals

1. INTRODUCTION

Machine Translation has multiple personalities. It is a technological challenge and has come to be understood as an economic necessity. It is a venerable scientific enterprise and a component of the larger area of studies concerned with the studies of human language understanding capacity [1].

Statistical machine translation, commonly known as SMT, provides a solution to overcome language barrier, and at the same time, rapid development of embedded technology enables people to install and run complex SMT systems on mobile terminals. We can fully expect that a mobile terminal with a SMT system would be an indispensable tool in international trade and communication.

Current commercial machine translation systems on mobile terminals are mainly based on the rule-based method requiring the manual development of linguistic rules, which can be costly, and which often does not generalize to other languages. On the contrary, the statistical method is not dependent on linguistic knowledge, and then translation model supports multi-language translation better, but it often consumes much memory requirements [2] [3].

For the past few years, with the development of machine translation systems, recent progress in corpus-based speech and language processing technology has made it possible to realize speech translation in real situations. Some multilingual speech translation systems have been constructed successfully [4-12]. It means that machine translation will be widely used on mobile terminals in the near future.

However, many low-cost embedded processors do not have a floating-point unit (FPU) support for floating-point arithmetic operations, which results in low translation performance of SMT systems on mobile terminals requiring numerous floating-point arithmetic operations. Similarly, speech translation on mobile terminals has also the same problem as that. However, fixed-point hardware implementations of SMT decoding algorithms can often achieve higher performance with lower computational requirements than floating-point hardware implementations. This paper proposes a fixed-point decoding approach to improve the problem. The experiments show that this approach enables users to achieve almost the same result as in floating-point implementation with minimum hardware resources and improves the translation speed of SMT systems on mobile terminals effectively.

The organization of this paper is as follows. In section 2, the isolated numerical foundation used in this paper is provided in detail. In section 3, we introduce three current traditional methods to handle floating-point arithmetic operations when processors have no FPUs, and then we present the detail implementation of the fixed-point approach. In section 4, we present the experimental results. Section 5 is the summary and future work.

2. NUMERICAL FOUNDATION

Before presenting the fixed-point decoding approach, we provide a brief introduction to the numerical foundation closely related to the conversion from floating-point numbers to fixed-point numbers.

2.1 Floating-point

The IEEE Standard for Binary Floating-Point Arithmetic [13], namely IEEE 754-1985, is the most common representation today for real numbers on computers. IEEE 754-1985 floating-point numbers have three basic components: the sign, the exponent and the mantissa. The mantissa is composed of the fraction and an implicit leading digit. The exponent base (2) is implicit and need not be stored.

This work was supported by the Fundamental Research Funds for the Central Universities (2009JBM027)

The IEEE 754-1985 specifies how single-precision (32 bit) and double precision (64 bit) floating point numbers are to be represented, as well as how arithmetic should be carried out on them.

(1) Single-precision

The IEEE 754-1985 single-precision floating point standard representation requires a 32-bit word, which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, S, the next eight bits are the exponent bits, E, and the final 23 bits are the mantissa M.

The Fig. 1 shows the representation of single-precision floating-point number.

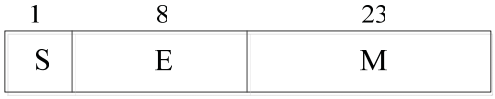

Fig. 1. Single-precision floating-point representation

(2) Double precision

The IEEE 754-1985 double precision floating point standard representation requires a 64-bit word, which may be represented as numbered from 0 to 63, left to right. The first bit is the sign bit, S, the next eleven bits are the exponent bits, E, and the final 52 bits are the mantissa M.

The Fig. 2 shows the representation of double precision floating-point number.

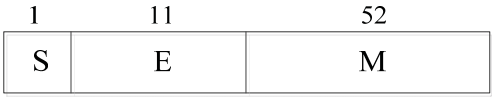

Fig. 2. Double-precision floating-point representation

The sign bit

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the floating-point number.

The exponent

The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE 754-1985 single-precision floating-point number, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), namely 73. Note that exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers. For IEEE 754-1985 double precision floating-point number, the exponent field is 11 bits, and has a bias of 1023.

The mantissa

The mantissa, also known as the significand, represents the precision bits of the floating-point number. It is composed of an implicit leading bit and the fraction bits.

To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number one can be represented as any of these:

1.00×10^{0}

0.01×10^{2}

100×10^{-2}

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, one is represented as 1.0×10^{0}.

2.2 Fixed-point

In computer science, the fixed-point number representation is a real data type. Fixed-point numerical representation uses a series of bits in binary format to represent a value. It is specified in the form of Fig. 3, where S indicates signed representation, IWL is the integer wordlength, and FWL is the fractional wordlength.

After deciding the location of implied binary point, the fixed-point format of all numbers is uniform, so we never consider the decimal problem.

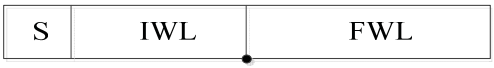

Fig. 3. Fixed-point specification

3. THE FIXED-POINT APPROACH

There are three traditional methods to handle floating-point arithmetic operations when processors have no FPUs:

(1) Define variables as floating-point type. Advanced programming languages call run-time function to deal with floating-point arithmetic operations automatically. It eliminates the difference between fixed-point processors and floating-point processors so that programmers can make less work. But the method will produce bloated code and slow processing speed;

(2) Define variables as integer type. We handle floating-point arithmetic operations using a magnification method. The method is simple, but lack of flexibility;

(3) If processors have no FPUs, then the floating-point instructions are trapped and executed by the floating-point emulator module. Programmers don't have to know whether or not the processor has a FPU. The only real difference is execution speed.

This paper proposes a fixed-point approach to address the deficiencies of the traditional methods and improve the floating-point arithmetic operations speed of processors without FPUs.

3.1 Conversion

The first step is to decide the fixed-point format by analyzing the quantization and resolution of involved floating-point numbers. Q-format representation is a fixed-point format where the number of fractional bits (and optionally the number of

integer bits) is specified. For example, a Q15 number has 15 fractional bits; a Q1.14 number has 1 integer bit and 14 fractional bits. Because Q-format numbers are fixed-point numbers, they can be stored and operated as integers. It is often used in processors that have no FPUs to represent fixed-point numbers.

Therefore, for SMT decoding on mobile terminals, we adopt the Q-format to deal with floating-point arithmetic operations. We can also change the Q-format to fit in with different requirements.

The implementation of converting floating-point numbers into Q-format fixed-point numbers is as follows.

(1) Get the consecutive bytes of a floating-point number in float or double variable type, and then store them in a specific integer variable.

(2) Get the sign, the exponent and the mantissa of the floating-point number according to the IEEE 754-1985.

(3) Scale the mantissa according to the exponent, and then convert the scaled mantissa into a corresponding fixed-point number in the Q-format. In addition, we handle the redundant bits (more than the word length of fixed-point number) in a truncated pattern.

Finally, we take a single-precision floating-point number 1.1234 as an example to explain the conversion intuitively. The floating-point format of 1.1234 is as shown in Fig. 4.

S	E	M
0	0111111	00011111100101110010010

Fig. 4. Single-precision floating-point format of 1.1234

Symbol: S=0, it shows that the number is positive;

Exponent: E=0x7F, the actual exponent E'=E-Bias=127-127=0;

Mantissa: M=0x8FCB92, the actual mantissa M'=M|0x800000= 0x8FCB92.

We convert the floating-point number into a Q8 format fixed-point number which is interpreted as a short integer type in C++ language. The Q8 fixed-point number format of 1.1234 is as shown in Fig. 5.

S	IWL	FWL
0	0000001	00011111

Fig. 5. Q8 format of 1.1234

3.2 Fixed-point class

In order to utilize variable precision fixed-point arithmetic in the SMT system on mobile terminals, a fixed-point class, called Fixed_SMT, is developed in the C++ language.

The implementation of the fixed-point class will be described later. In this way, the main operators in a floating-point implementation can be represented by fixed-point objects. A Fixed_SMT object contains two main attributes: IWL and FWL.

By defining a different integer word length and fraction word length for each fixed-point operator, all operators can be of arbitrary wordlength.

Floating-point numbers can be represented in fixed-point format, and the format used in this work is shown in Figure 4. All fixed-point numbers are represented in 2's complement format. The integer wordlength is the number of bits used to represent the integer part, and the fraction wordlength is the number of bits used to represent the fraction part.

C++ is used because it offers faster execution than other object oriented languages. Several operators such as +, -, * and = are overloaded by the fixed-point class.

- +/-: These operators can perform addition or subtraction of two fixed-point objects or a floating-point variable and a fixed-point object, and the result is a fixed-point object.
- *: This operator can perform multiplication of two Fixed_SMT objects or a floating-point variable and a Fixed_SMT object. The product's wordlength will be equal to the summation of the two input operands' wordlength minus one. Since the specific wordlength of fixed-point numbers is used for the whole decoding, we set the product's wordlength as same as the specific wordlength. It is important to note that we must keep track of the implicit implied binary point after multiplying Q-format numbers.
- =: If the input operand is a floating-point number, this operator will convert the floating point number into a fixed-point object at a precision specified by the target object. If the input operand is a fixed-point object, this operator will round it up to the target object's precision.

3.3 Operator overloading

Converting a floating-point program into a fixed-point program can be done by replacing the variable definition. The rest of the program is unchanged. For example, the following floating-point program:

```
float a;
float b;
float c;

a = 1.11;
b = 2.22;
c = a + b;
```

It can be transformed into the fixed-point implementation:

```
Fixed_SMT a;
Fixed_SMT b;
Fixed_SMT c;

a = 1.11;
b = 2.22;
c = a + b;
```

In the above fixed-point program, in statements a=1.11 and b=2.22, since the "=" operator is overloaded, floating-point values 1.11 and 2.22 will be converted into fixed-point format and stored in Fixed_SMT objects a and b. Statement "a+b" will be handled by the overloaded operator "+" and the result will be a Fixed_SMT object.

3.4 Design methodology

Fig. 6 shows the system design flow employed in this paper. The isolated model training was done using floating-point arithmetic. The decoder was rewritten in C++ using the fixed-point class. And then we need know the quantization of the floating-point variables involved in SMT decoding in advance, and then use the fixed-point objects to represent and replace the floating-point variables and simulate the original floating-point arithmetic operations according to the principles of fixed-point arithmetic operations. At last, it is necessary to test and adjust whether the quantization and precision of the fix-point class can make the system word correctly.

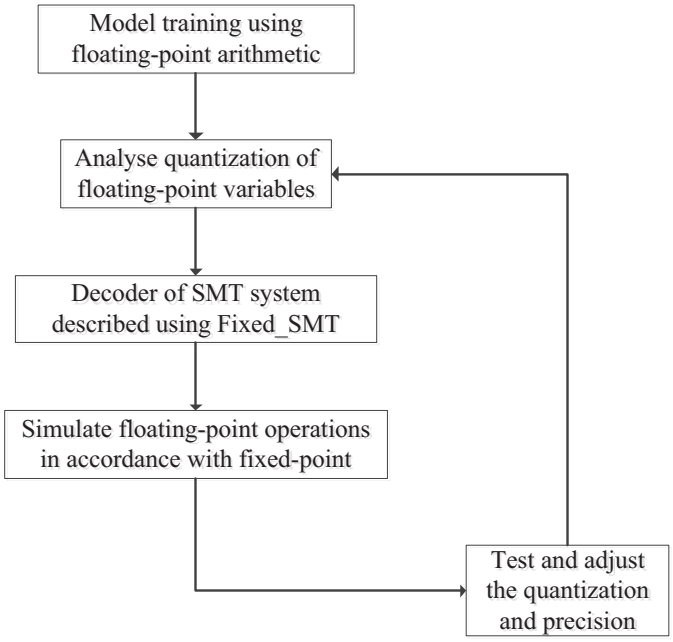

Fig. 6. System design flow

4. EXPERIMENTS

In this paper, we implement a fixed-point decoder based on Xiong et al. (2006)'s system Bruin [14] as our experimental decoder.

In this section, we present the experiments based on PC and mobile terminal to compare the performance of the fixed-point and the floating-point decoder.

4.1 experiment based on PC

We make the experiment with Chinese-to-English translation. The training corpus consists of 239K sentence pairs. We used the 2002 NIST MT Evaluation test set as our development corpus, and used the 2005 NIST MT Evaluation test set as out test corpus.

For the language model, we used SRI Language Modeling Toolkit [15] to train a 4-gram language model with modified Kneser-Ney smoothing [16]. Our evaluation metric is BLEU [17] as calculated by the script mteval-v11b.pl with its default setting.

Table 2 shows that the fixed-point decoder can ensure the quality of translation. However, the fixed-point decoding is only a little slower than the floating-point decoding, so we can predict that the translation speed can be much improved when we deploying the fixed-point decoder on mobile terminals without FPUs.

Table 1. Configuration of PC

Processor	Memory	Operating system
Quad-Core AMD Opteron Processor 8347 HE, 1.9 GHz	60 GB	Red Hat Enterprise Linux AS，X64

Table 2. Comparison of fixed-point and floating-point decoding

Decoder	Translation time	BLEU
Floating-point decoding	8978.74 s	0.3035
Fixed-point decoding	9066.57 s	0.3035

SMT systems have a significant requirement for mobile terminals in memory, but we are limited to the hardware condition, we decide to make the experiment to compare the arithmetic performance between fixed-point and floating-point for a mobile terminal so that we can indirectly verify the faster translation speed of the fixed-point decoder of SMT systems on mobile terminals.

4.2 experiment based on mobile terminal

We make 10,000,000 loop operations, and then select frequency multiplication and accumulation of SMT decoding as arithmetic calculation type in the experiment based on mobile terminal.

Table 4 shows that arithmetic performance of fixed-point is faster 135.6% than that of floating-point. Therefore, we come to a conclusion that this approach can improve translation speed of SMT decoding when deploying the fixed-point decoder on the mobile terminal without FPUs.

Table 3. Configuration on mobile terminal

Processor	Memory	Operating system
Intel ARM920T PXA27X, 312MHz	64 MB	Windows Mobile 6.0 Standard

Table 4. Comparison of fixed-point and floating-point arithmetic operations

	Operation time
Floating-point	14.75 s
Fixed-point	6.25s

5. SUMMARY AND FUTURE WORK

Based on the analysis of current SMT systems on mobile terminals, we draw a conclusion that embedded processors without FPUs restrict decoding speed. By analyzing the difference between floating-point and fixed-point, and associated arithmetic operations algorithm, we propose a fixed-point approach of SMT decoding to solve the bottleneck. This approach improves decoding speed of SMT systems on mobile terminals with weak ability in floating-point arithmetic operations significantly so that SMT decoding on mobile terminals don't have to rely on the FPU. SMT decoding can be finished by using a universal CPU on mobile terminals. It improves translation speed and ensures good quality of translation.

The fixed-point approach is compatible with most on mobile terminals with different processors and SMT decoders so that it is convenient for users to adjust the parameter configuration of the fixed-point class and satisfy different application requirements.

In addition, this approach also can be applied to speech recognition, audio & video decoding and other related fields. In our future work, some experiments would be made to verify the validity in these fields.

ACKNOWLEDGMENT

I am indebted to Jin'an Xu and Qun Liu for their guide. Thanks to Yajuan Lü, Wenbin Jiang, Hao Xiong, Zhaopeng Tu, Wei Luo, Linfeng Song and Meng Sun's help. Finally, I am indebted to the reviewers for helpful suggestions that improved the quality of this paper.

REFERENCES

[1] Nirenburg, Sergei and Yorick Wilks. 2000. Machine translation. Advances in Computers, 52:160-189.

[2] Jinan Xu, Prospects in Machine Translation, Cross-Strait Conference on Information Science and Technology, CSCIST 2010, pp368-372, Qinhuangdao.

[3] Qun Liu, Survey on Statistical Machine Translation, Journal of Chinese Information Processing，2003，17(4):1-12

[4] R. Zhang, H. Yamamoto, M. Paul, H. Okuma, K.Yasuda, Y. Lepage, E. Denoual, D. Mochihashi, A.Finch, and E. Sumita, "The NiCT-ATR Statistical Machine Translation System for the IWSLT 2006 Evaluation," Proc. of the International Workshop on Spoken Language Translation, pp. 83-90, Kyoto, Japan, 2006.

[5] T. Shimizu, Y. Ashikari, E. Sumita, H. Kashioka, and S. Nakamura, "Development of client-server speech translation system on a multi-lingual speech communication platform," Proc. of the International Workshop on Spoken Language Translation, pp. 213-216, Kyoto, Japan, 2006.

[6] S. Nakamura, K. Markov, H. Nakaiwa, G. Kikui, H. Kawai, T. Jitsuhiro, J. Zhang, H. Yamamoto, E.Sumita, and S. Yamamoto. The ATR multilingual speech-to-speech translation system. IEEE Trans. on Audio, Speech, and Language Processing, 14, No.2:365–376, 2006.

[7] FügenC, KolssM, PaulikM, WaibelA (2006b) Open domain speech translation: from seminars and speechesto lectures. In: TC-STAR workshop on speech to speech translation, Barcelona, Spain, pp.81-86.

[8] Hamon O, Mostefa D, Choukri K (2007) End-to-end evaluation of a speech-to-speech translation system in TC-STAR. In: Machine translation summit XI, Copenhagen, Denmark, pp 223–230.

[9] Stüker S, Paulik M, Kolss M, Fügen C, Waibel A (2007) Speech translation enhanced ASR for European Parliament speeches – on the influence of ASR performance on speech translation. In: ICASSP 2007, international conference on acoustics, speech, and signal processing, Honolulu, Hawaii, pp. 1293-1296.

[10] Zhou, Bowen, Déchelotte Daniel, and Gao Yuqing , Two-way Speech-to-Speech Translation on Handheld Devices, Intl. Conf. on Spoken Language Processing, 10/2004, Jeju Island, South Korea, p. 1637-1640, (2004)

[11] Shimizu T, Ashikari Y, Sumita E, et al. NICT-ATR Chinese-Japanese-English speech-to-speech translation system. In: Proceedings of the 9th National Conference on Man-Machine Speech Communication (NCMMSC). Huangshan,China, 2007.

[12] N. Tsourakis, M. Georghescul, P. Bouillon, and M. Rayner. 2008. Building mobile spoken dialogue applications using Regulus. In Proceedings of LREC 2008, Marrakesh, Morocco.

[13] IEEE Standards Board and ANSI. IEEE Standard for Binary Floating-Point Arithmetic, 1985, IEEE Std 754-1985-1985.

[14] Deyi Xiong, Qun Liu, Shouxun Lin. Maximum entropy based phrase reordering model for statistical machine translation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, p. 521-528, 2006.

[15] Andreas Stolcke. SRILM—An Extensible Language Modeling Toolkit. *Proc. Intl. Conf. on Spoken Language Processing*. v. 2, p. 901-904, 2002.

[16] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. Technical Report TR-10-98, Harvard University Center for Research in Computing Technology, 1998.

[17] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of ACL 2002*, p. 311-318, 2002.

[18] David Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic". ACM Computer Surveys, v. 23, n. 1, p. 5-48, 1991.

[19] Andrea G. M. Cilio, Henk Corporaal. "Floating-Point to Fixed-Point Conversion of C Code". 1997.

[20] Robert Gordon. "A Calculated Look at Fixed-Point Arithmetic". Embedded Systems Programming, p. 72-78, 1998.

[21] Bjarne Stroustrup. The C++ Programming Language, 2nd ed. Reading, MA: Addison Wesley, 1993.

[22] Computer Organization and Architecture, William Stallings, pp. 222-234 Macmillan Publishing Company, ISBN 0-02-415480-6.

[23] K.K. Shin, J.C.H. Poon, and K.C. Li, "A fixed-point DSP based Cantonese recognition system," in IEEE International Symposium on Industrial Electronics, pp. 390–393 vol.1,1995.

[24] Nishida, Y., Nakadai, Y., Suzuki, Y., Sakurai, T., Kurokawa, T., and Sato, H., "Voice recognition focusing on vowel strings on a fixed-point 20-MIPS DSP board," in IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 137–140 vol.1, 1999.

[25] Guanghui Hui, Kwok-Chiang Ho, and Zenton Goh, "A robust speaker-independent speech recognizer on ADSP2181 fixed-point DSP," in 1998 Fourth International Conference on Signal Processing, pp. 694–697 vol.1, 1998.

[26] Yuet-Ming Lam, Man-Wai Mak, and Philip Heng-Wai Leong, "Fixed-Point Implementations of Speech Recognition Systems," GSPx Conference, Apr.3, 2003.