Implementing the Log-Add Algorithm in Hardware

Melnikoff, Stephen; Quigley, Steven

DOI:
10.1049/el:20030594

Document Version
Peer reviewed version

Citation for published version (Harvard):
Melnikoff, S & Quigley, S 2003, 'Implementing the Log-Add Algorithm in Hardware', *Electronics Letters*, vol. 39, no. 12, pp. 939-941. https://doi.org/10.1049/el:20030594

Implementing the log-add algorithm in hardware

S J Melnikoff & S F Quigley

Electronic, Electrical and Computer Engineering, University of Birmingham,

Edgbaston, Birmingham, B15 2TT, United Kingdom

Corresponding author: Steve Melnikoff

E-mail: s.j.melnikoff@iee.org

Implementing the log-add algorithm in hardware

S J Melnikoff & S F Quigley

We present a hardware implementation of the log-add algorithm, being a simple method of computing ln(A + B) given ln(A) and ln(B), as used in speech recognition. We show that it can be efficiently implemented in hardware using a small look-up table plus some additional arithmetic logic, with no significant loss of accuracy over direct calculation.

Introduction

As part of the speech recognition process [1], we are required to compute probabilities based on Gaussian mixtures. Each mixture has components which are computed in the log domain, but which must be added in the normal domain.

While we could use large look-up tables [2] or CORDIC (co-ordinate rotation digital computer) [3] to convert between domains, a convenient algorithm exists for this specific problem [4]. It removes the need to perform a conversion at all, instead relying on a look-up table significantly smaller than that of the logarithm or exponential operations, along with some simple arithmetic computations — and hence well suited for implementation in hardware.

Accordingly, in this Letter we describe the theory behind this algorithm, and give details of our novel implementation on a field-programmable gate array (FPGA),

which forms part of a hardware speech recognition system. This implementation requires fewer resources and has a lower latency than the alternatives.

Theory

Given two values $\ln(A)$ and $\ln(B)$ for which we would like to compute $\ln(A+B)$:

$$A+B = A(1+B/A) \quad (1)$$

$$\Rightarrow \ln(A+B) = \ln(A(1+B/A))$$

$$\Rightarrow \ln(A+B) = \ln(A) + \ln(1+B/A).$$

To compute the result, we work out $\ln(B/A)$, which is simply equal to $\ln(B) - \ln(A)$, and then use a look-up table to map that value to $\ln(1+B/A)$. Since the values in this table are dependent on the *relative* values of A and B, it can be smaller than one which relies on their actual values.

In order to minimise the size of the look-up table without compromising accuracy, we require that $A \geq B$, hence limiting $1+B/A$ to the range 1 to 2, and switch the values if this condition is not met.

Data representation

The calculations for speech recognition are best performed in the negative log domain, as it reduces the many multiplications associated with the process to additions, for which hardware is better suited.

Recognition is a statistical process, and so the values used are probabilities. If a probability A is converted to the negative log domain by computing $-\ln(A)$, a 16-bit

log-domain integer value would represent the probabilities from 3×10^{-28462} to 1, a range which is far too broad for our purposes. A more reasonable range is 10^{-12} to 1, which can be achieved by computing $-K\ln(A)$, where K equals –2371.8. This approach is used by the HTK speech development toolkit [5], which was used to generate the speech models that we used, and to verify the results of our system.

We found that for the more complex speech models, 16 bits was not sufficient to maintain accuracy, and so 24-bit values were used instead, resulting in the range of probabilities being 10^{-3072} to 1. The value of K was kept constant in order to maintain compatibility with HTK.

Implementation

As a hardware implementation, this algorithm seems ideal, since it relies on functions easily realisable on a chip. But in order to give it a significant advantage over the alternative methods described below, the look-up table needs to be kept as small as possible without adversely affecting accuracy.

Hence the first step of the implementation was to analyse the data to be used in the table. Software was written to perform the calculations directly, and produce a full set of values. These were then inspected in order to identify patterns which could be used to produce a more efficient design. This involved keeping to a minimum both the number of entries in the look-up table, and the amount of additional logic required.

Inspection of this data revealed that when $K\ln(B/A)$ is 0, $K\ln(1+B/A)$ is –1644. Since all of the outputs are negative, we ignore the sign at this stage. So taking the outputs as positive numbers, as the input value increases, the output decreases, initially at the rate of 1 for every 2 increments of the input, and then more and more slowly. The first consequence of this was that we could ignore the least significant bit (LSB) of the input, as it did not affect the output by more than ±1. The other was that for all values of the input above 16,384, the output changed only twice, decreasing from 2 to 1 at 17,471, and then to 0 at 20,077.

The result of this was that a table 8,192 entries deep and 11 bits wide (a total of 11 Kb) was sufficient to represent all values of the input from 0 to 16,384 (discarding the LSB), with the two values above this handled using a couple of comparators, as shown in Fig. 1.

The only other processing required was a comparator for the two inputs $K\ln(A)$ and $K\ln(B)$, a subtractor to compute their difference, and another subtractor to subtract the smaller (i.e. more negative) input from the output of the look-up table (which is equivalent to adding the smaller input to the negative of the value from the look-up table, required because the numbers stored in the look-up table are positive, the minus sign having been discarded). The architecture of the log-add block is shown in Fig. 2.

Domain conversion

The alternative to this algorithm is to convert the data from the log domain to the normal domain (i.e. take the exponential), perform the summation, and then take the

log of the result.

Two different approaches have been previously implemented on an FPGA. In [2], a RAM-based look-up table is used to perform the conversion on 16-bit log-domain values, producing 16-bit results; 128 Kb of storage is required for this. The reverse operation is performed in software; were it to be implemented in hardware, we would assume that another similarly sized table would be required. This compares to one 11 Kb look-up table in our design.

Alternatively, CORDIC [3] provides a very resource-efficient method for performing non-linear operations like these. An iterative implementation requires a very small look-up table (just one entry per bit of accuracy), and incurs a delay of one clock cycle per bit. A fully pipelined version does not use a look-up table at all, and incurs a latency of one clock cycle per bit. Two CORDIC blocks would be required for the two conversions, with a couple of additional cycles for the summation itself. Our design, using 24-bit numbers, has a total latency of 4 cycles.

If we are converting between domains, there is also the issue of data representation to consider. As mentioned above, if we use the scaling factor *K*, a 16-bit value in the log domain translates to a number between 10^{-12} and 1, which would need to be represented as a 40-bit fixed-point value if complete accuracy were required, or floating-point otherwise. With 24 bits, the range is 10^{-3072} to 1, which cannot be sensibly represented in fixed-point. Hence avoiding a domain conversion circumvents these issues.

Conclusion

We have shown one way in which the log-add algorithm can be implemented in hardware, in this case as part of a speech recognition system. We have shown that this method requires significantly less data storage than domain conversion based on look-up tables, while having a shorter latency than CORDIC, and in both cases avoiding the problem of how best to represent data in the normal domain.

While the size of the look-up table and the cutoff values are specific to our implementation, the nature of the data ensures that optimisations can be made in the manner described.

Using an existing theory, our design demonstrates that what would otherwise be a complex calculation can be reduced to elements well-suited to hardware, with no significant loss of accuracy.

References

[1] MELNIKOFF, S.J., QUIGLEY, S.F. and RUSSELL, M.J.: 'Performing speech recognition on multiple parallel files using continuous hidden Markov models on an FPGA'. Proc. IEEE International Conference on Field-Programmable Technology, FPT 2002, 2002, pp. 399-402

[2] STOGIANNOS, P., DOLLAS, A. and DIGALAKIS, V.: 'A configurable logic based architecture for real-time continuous speech recognition using hidden

Markov models', *Journal of VLSI Signal Processing Systems*, 2000, **24** (2-3), pp. 223-240

[3] ANDRAKA, R.: 'A survey of CORDIC algorithms for FPGA based computers'. ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '98, 1998, pp. 191–200 and http://www.fpga-guru.com/papers.htm

[4] HOLMES, J. N and HOLMES, W.J.: 'Speech synthesis and recognition' (Taylor & Francis, London, 2001 2nd edn.)

[5] YOUNG, S., EVERMANN, G., KERSHAW, D., MOORE, G., ODELL, J., OLLASON, D., POVEY, D., VALTCHEV, V. and WOODLAND, P.: 'The HTK Book' (Cambridge University Engineering Department, 2002) and http://htk.eng.cam.ac.uk/docs/docs.shtml

Figures

Fig. 1. Log-add table structure. The 'and' gate with negated inputs is equivalent to a comparator checking if the input is less than 16384, but requires fewer resources. The multiplexor outputs zero (option 'd') if the input is greater than or equal to 20077

Fig. 2. Log-add structure

Fig. 1.

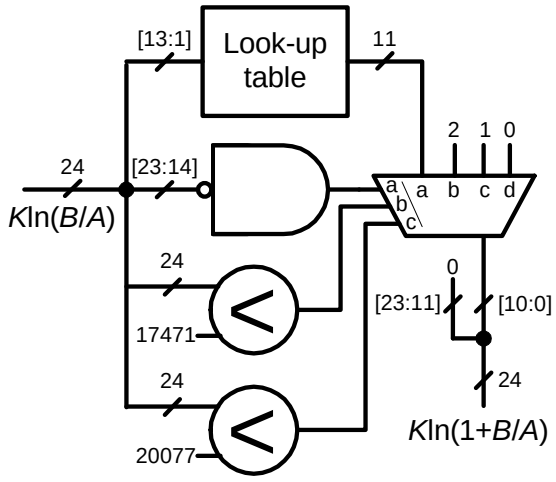
[13:1]
Look-up table
11
2 1 0
24
[23:14]
a
b
c
a b c d
K ln(B/A)
24
17471
0
[23:11]
[10:0]
24
24
20077
K ln(1+B/A)

Fig. 2.