INTERNATIONAL
COMPUTER SCIENCE
INSTITUTE

Revisiting a basic function on current CPUs: A fast logarithm implementation with adjustable accuracy

O. Vinyals, G. Friedland, N. Mirghafori

TR-07-002

June, 2007

Abstract

In this report, we present an implementation of the logarithm function that takes better advantage of the architecture of current processors than previous implementations. The proposed C-language function is a fast single precision approximation of the natural logarithm with adjustable accuracy. Given an IEEE 754 floating point number, the main idea is to use a quantized version of the mantissa as a pointer into a lookup table. The amount of quantization of the mantissa determines the table size and therefore the accuracy. Current processors are able to store relatively large lookup tables in cache memory. Therefore an acceptable accuracy can be reached without too many main memory accesses. We measured a speed up of about factor 6 with respect to the standard C-library implementation while keeping the absolute error as low as 10e−6. This article presents and discusses our proposed implementation with respect to other logarithm realizations on different platforms. Measurements are performed using a dedicated benchmark and by testing the performance of the function as part of a real application.

Revisiting a basic function on current CPUs: A fast logarithm implementation with adjustable accuracy

Oriol Vinyals, Gerald Friedland, Nikki Mirghafori
{vinyals,fractor,nikki}@icsi.berkeley.edu

June 21, 2007

Abstract

In this report, we present an implementation of the logarithm function that takes better advantage of the architecture of current processors than previous implementations. The proposed C-language function is a fast single precision approximation of the natural logarithm with adjustable accuracy. Given an IEEE 754 floating point number, the main idea is to use a quantized version of the mantissa as a pointer into a lookup table. The amount of quantization of the mantissa determines the table size and therefore the accuracy. Current processors are able to store relatively large lookup tables in cache memory. Therefore an acceptable accuracy can be reached without too many main memory accesses. We measured a speed up of about factor 6 with respect to the standard C-library implementation while keeping the absolute error as low as 10^{-6}. This article presents and discusses our proposed implementation with respect to other logarithm realizations on different platforms. Measurements are performed using a dedicated benchmark and by testing the performance of the function as part of a real application.

1 Motivation

During our research on speeding up a machine learning algorithm, namely the ICSI Speaker Diarization Engine[1] [7, 1], we found that a major bottleneck was the computation of the natural logarithm. This comes at no surprise because many machine learning systems, for example when using Gaussian Mixture Models combined with Hidden Markov Models, rely heavily on the computation of logarithms because they use log-likelihoods as a basic similarity measure. Profiling the ICSI Speaker Diarization Engine, we found that computing the log-likelihood took about 80 % of the total runtime.

[1]The task of a speaker diarization is to segment an audio recording into speaker-homogeneous regions. That means, given a single-source recording, the engine is to determine "who spoke when".

Of course, two strategies can be followed to improve the speed of such a bottleneck: One can either change the structure of the algorithm and reduce the number of log-likelihood calculations or one can reduce the execution time of the logarithm function itself. The second option is often not considered because one assumes that compilers and standard libraries already have a very optimized version of these basic functions. We found, however, that many implementations of the logarithm function are either too slow, too inaccurate, or require special hardware.

This article presents the realization of a platform independent, fast C-language implementation of the logarithm function. The idea behind the approach is to take advantage of the large amount of cache available in current processors. We demonstrate that using CPU caches for storing a lookup table helps speed up the logarithm computation dramatically, without the requirement of specialized hardware. We call the approach *ICSILog*.

Section 2 introduces the currently most common logarithm implementations before Section 3 describes the idea of our approach. Section 4 presents speed and accuracy measurements on different platforms. We conclude with Section 5 followed by the references. An Appendix contains the source code of the current version of the ICSILog.

2 Related Work

Apart from the dependency on the IEEE 754 [6] floating point standard we did not want to assume any hardware requirements, so our resulting code would be portable. A search for different fast logarithm implementations results in mostly special purpose solutions. Many of them require additional hardware.

The default *math.h* logarithm function computes a high-order Taylor approximation to achieve floating point precision. This involves a large number of multiplications and sums of floating point numbers. Throughout the article we will refer to the standard GCC 4.0.1 implementation of the logarithm as our baseline.

Laurent de Soras published an algorithm called *FastLog* [2] in 2001. His algorithm basically computes an order-3 Taylor approximation of any given IEEE 754 floating point number. The algorithm is fast but less accurate (compare Table 1). We found that our approach using a lookup table was as fast as this implementation with better accuracy.

Many compilers offer an option to trade off floating point accuracy for speed. The GNU Compiler Collection (GCC) [3] for example offers a flag called *-ffast-math*. When this compiler flag is on, the compiler uses speed optimizations that can result in incorrect output for programs which depend on an exact implementation of IEEE or ISO specifications for math functions. When applying this flag to compile the algorithm described in Section 1, we achieve only a little speed up but the performance of the overall system noticeably decreases.

Advanced Micro Devices, Inc (AMD) offers the *AMD Core Math Library* (ACML) [4]. ACML is a performance-tuned math library relying on the current processors series produced by AMD. These include AMD Opteron and AMD Athlon 64. AMD claims that computing a natural logarithm with floating point precision takes only 94 CPU cycles. Although this library is dependent on the use of processors by AMD, we included it in our comparison because we do have access to this kind of hardware. The results are described in Section 4.2.

State-of-the-art 3D graphics cards are equipped with so-called *Graphic Processor Units* (GPUs). They offer a significant amount of processing power also for floating point math operations. NVIDIA, Inc for example offers the so-called Compute Unified Device Architecture (CUDA) [5] on their recent models GeForce 8800 GTX and GTS. The idea is to give computationally intensive applications access to highly-parallelized processing through an easy-to-use programming interface. NVIDIA claims the log function can be computed in only four GPU cycles. However, this does not take into account that GPUs are usually clocked at lower frequencies than current CPUs. Most importantly, there is a rather large communication overhead when using CUDA only for some basic computations. Therefore using CUDA requires a complete re-design of any given algorithm and in the end one relies on a proprietary hardware solution.

3 What is ICSILog?

The core idea of the approach described here is to increase the performance of the logarithm computation by relying on a lookup table that can easily reside in CPU cache. A pre-calculation of all logarithms for the entire floating point number domain would take prohibitive amounts of memory (about 8 GB). Of course, a table of this size would neither fit into the cache memory of current CPUs.

Fortunately, the size of the look up table can be reduced by exploiting the way floating point numbers are represented in memory. Conceptually, a 32-bit IEEE 754 floating point number is stored as follows. A value *val* of a number is the product of a 23-bit mantissa *man* and an 8-bit exponent *exp*. One bit is reserved for the sign s. If $s = 0$, the sign is positive, otherwise, it is negative. Since the real-valued logarithm is only defined for positive numbers, the sign bit can be ignored. We get:

$$val = 2^{exp} \cdot man$$

We can use the multiplicative property of the logarithm function to decompose the logarithm computation as:

$$\log_2(val) = \log_2(2^{exp} \cdot man) = exp + \log_2(man)$$

In order to calculate the natural logarithm, we can take advantage of the property that all logarithms are proportional to each other. This results in the following equation:

$$\log_e(val) = (exp + \log_2(man)) \cdot \log_e(2) = exp \cdot \log_e(2) + \log_2(man) \cdot \log_e(2)$$

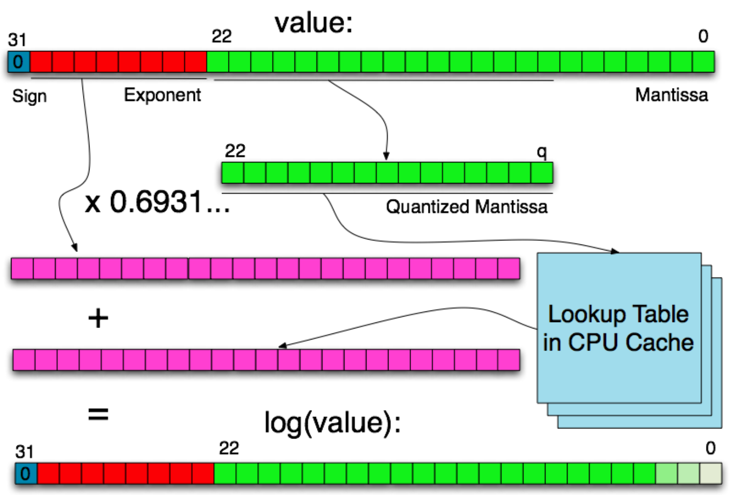

Figure 1: Concept of ICSILog algorithm. An IEEE 754 floating point number is decomposed into mantissa and exponent. The mantissa is quantized and used as a pointer into a lookup table that should fit into CPU cache. The result of the look up can be easily composed with the downscaled exponent using one addition.

Of course, $\log_e(2) = 0.6931471805...$ is a constant. Calculating the logarithm with respect to any other base only requires multiplying with a different constant.

Extracting the exponent and the mantissa of a floating point number can be performed quickly using bit shift operations. Therefore, in order to calculate the left part of the sum, only one multiplication is required. To calculate the right part of the sum, we store the results of the computation

$$\log_2(man) \cdot \log_e(2)$$

in a lookup table. Unfortunately, this still requires a table with 2^{23} entries with each entry needing 4 bytes, thus 32 MB. In our experiments (see Section 4.1), we found that using a table of this size increases the performance of the logarithm computation only very slightly since memory accesses take about the same time as the computation of the Taylor approximation. In order for the look up table to fit into cache, we quantize the mantissa, i.e. we ignore q least significant bits of the mantissa.

The table is then indexed using the $23 - q$ most significant bits of the mantissa. The result is calculated by adding the value looked up in the table and the downscaled exponent. Figure 1 shows a diagram illustrating the steps explained in this section.

Of course, accuracy is lost because of the quantization of the mantissa, as will be discussed in the next section.

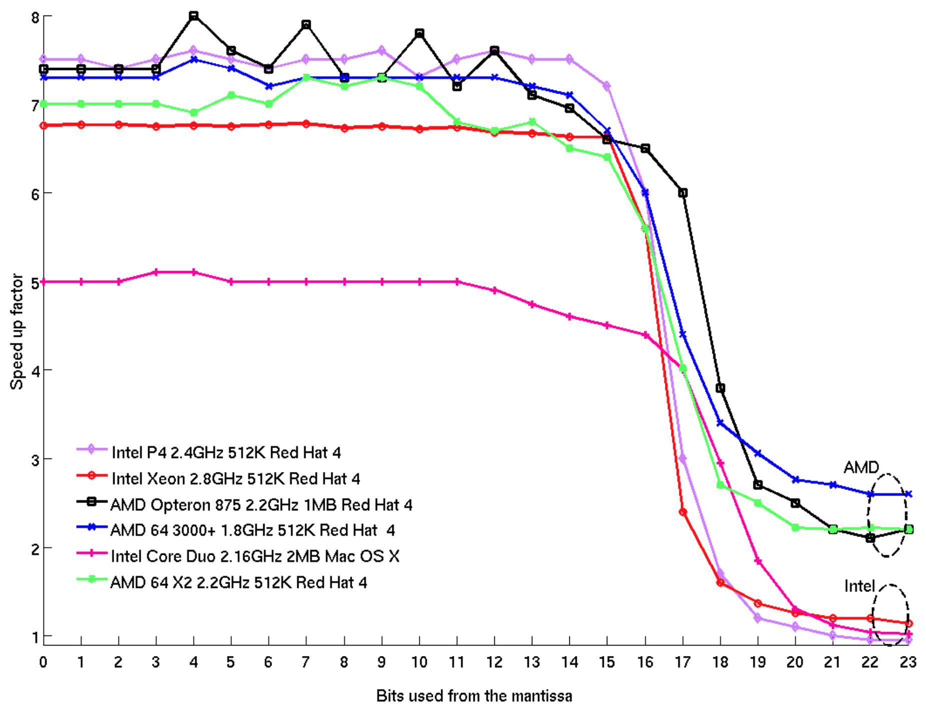

Figure 2: Speed-accuracy trade off of the ICSILog on different systems relative to the GCC standard log implementation. Indicated is processor type, cache size, and operating system. The speed drops at a point where there are too many cache misses. It is also observed that when all bits are used, AMD machines still mantain a speed up of 2.0. On Intel machines the speed up factor drops to 1.0.

4 Performance of ICSILog

This section discusses the accuracy-performance trade-off for ICSILog and compares our proposed implementation with different state-of-the-art logarithm realizations, both using a simple benchmark application and in a real application.

4.1 Speed-Accuracy Trade-Off

In order to find the best trade-off between accuracy (level of quantization of the mantissa) and speed, we measured the time it takes to calculate the logarithm of 10 million random numbers on different CPUs and operating systems. The speed is compared to the execution time of the standard implementation of the logarithm.

Figure 2 shows the speed of the ICSILog relative to the standard log on different CPUs and platform for different table sizes. It can be observed that

Method	Speed up	Error
Standard Log	1.0	0.00
AMD ACML Log	1.7	$1.10\ 10^{-7}$
-ffast-math Log	2.0	$1.42\ 10^{-7}$
FastLog	6.0 – 7.0	$4.26\ 10^{-3}$
ICSILog (q=0)	2.0	0.00
ICSILog (q=7)	6.0	$6.55\ 10^{-6}$
ICSILog (q=8)	6.0 – 8.0	$1.31\ 10^{-5}$

Table 1: Speed up table and error of several logarithm implementations performed on an AMD machine. q is the number of least significant bits ignored from the mantissa.

the execution speed starts to decrease when we use about 16 bits from the mantissa (i.e., $q = 7$). This is the point where the look up table is too big to be constantly held in cache memory.

Using no quantization of the mantissa, which results in no loss of accuracy, the speed up on current Intel systems is about 1.0 and still 2.0 on AMD systems. There are many factors that could cause this behavior. A possible explanation is that main memory access is more optimized on AMD architecture mainboards.

4.2 Benchmark Results

Table 1 shows the performance of ICSILog compared to other logarithm implementations. We measured the time it takes to calculate the log using different implementations for 10 million random numbers. All the experiments were performed on an AMD Opteron 875 (64 bits) 2.2 GHz dual core with 1024 KB cache. This made it possible to compare the performance of the ICSILog against the logarithm implementation of the ACML library (see Section 2). The operating system was Red Hat Enterprise 4 and the benchmark application was compiled using GCC 4.0.1. With $q = 7$, ICSILog was faster than any other tested logarithm implementations while maintaining an accuracy of $6.55\ 10^{-6}$ compared to the standard implementation.

4.3 ICSILog in a Real Application

Since ICSILog is based on cache utilization, it is important to measure the performance of the algorithm in a real application where the CPUs cache memory is also used for other purposes. Since our initial objective was speeding up a speaker diarization algorithm [7], Table 2 shows the results of using different logarithm implementations inside this engine. Inside a real application, there are many factors to be taken into account that influence speed and/or accuracy. However, the example shows that our proposed ICSILog is able to increase the performance of a real application significantly while maintaining a better accuracy than FastLog.

	Standard Log	FastLog	ICSILog (q=11)
Time needed	100%	49%	45%
Diarization Error Rate	11.74%	12.14%	11.74%

Table 2: Speed up and error of ICSILog in a real application. For more details refer to the text.

5 Conclusion

We propose a new implementation of the logarithm function, called ICSILog. This platform and hardware independent realization of the logarithm function achieves a better speed-accuracy trade-off than any other current implementation. The goal is achieved by taking advantage of the large and fast cache memories of current CPUs. With cache memories growing, ICSILog can be used with increased table sizes. Then the function will become even more accurate without a loss in performance. In the future, we expect that more and more basic functionality can be optimized towards cache utilization.

Credits

Oriol Vinyals implemented ICSILog and conducted all experiments.

Gerald Friedland had the initial idea for the ICSILog. He supervised the development and the experiments.

Nikki Mirghafori inspired the development of faster Speaker Diarization algorithms and gave helpful advice on many aspects of the project.

We would like to thank Chuck Wooters and Marijn Huijbregts for their advice on the ICSI Speaker Diarization Engine. We would like to thank Nelson Morgan, Chuck Wooters, Marijn Huijbregts, Adam Janin, Yan Huang, David Johnson, and Benoit Favre for their participation in discussions around the ICSILog. This work was (partly) funded by DTO VACE program. Gerald Friedland was supported by a fellowship within the postdoc program of the German Academic Exchange Service (DAAD). Oriol Vinyals was supported by the European Union 6th FWP IST Integrated Project AMIDA (Augmented Multiparty Interaction with Distant Access).

References

[1] J. Ajmera and C. Wooters. A robust speaker clustering algorithm. In *Proceedings of IEEE Automatic Speech Recognition and Understanding (ASRU)*, pages 411–416, St. Thomas, U.S. Virgin Islands, December 2003.

[2] Laurent de Soras. Fastlog Function (last visited: 06-18-2007). `http://www.flipcode.com/cgi-bin/fcarticles.cgi?show=63828`.

[3] GNU Foundation. GCC, the GNU Compiler Collection (last visited: 06-18-2007). http://gcc.gnu.org/.

[4] AMD Inc. AMD Core Math Library (ACML) (last visited: 06-18-2007). http://developer.amd.com/acml.jsp.

[5] NVidia. NVIDIA CUDA. Revolutionary GPU Computing (last visited: 06-18-2007). http://developer.nvidia.com/cuda.

[6] Institute of Electrical and Electronics Engineers. IEEE 754-1985: Standard for Binary Floating-Point Arithmetic, 1985. http://grouper.ieee.org/groups/754/.

[7] Chuck Wooters and Marijn Huijbregts. The ICSI RT07s Speaker Diarization System. *Lecture Notes in Computer Science*, 2007 (to appear).

Source Code

```
/* Creates the ICSILog lookup table. Must be called
   once before any call to icsi_log().
   n is the number of bits to be taken from the mantissa
     (0<=n<=23)
   lookup_table is a pointer to a floating point array
    (memory has to be allocated by the user) of 2^n positions.
*/
void fill_icsi_log_table(const int n, float *lookup_table)
{
   float numlog;
   int *const exp_ptr = ((int*)&numlog);
   int x = *exp_ptr; //x is the float treated as an integer
   x = 0x3F800000; //set the exponent to 0 so numlog=1.0
   *exp_ptr = x;
   int incr = 1 << (23-n); //amount to increase the mantissa
   int p=pow(2,n);
   for(int i=0;i<p;++i)
   {
       lookup_table[i] = log2(numlog); //save the log value
       x += incr;
       *exp_ptr = x; //update the float value
   }
}

/* Computes an approximation of log(val) quickly.
   val is a IEEE 754 float value, must be >0.
   lookup_table and n must be the same values as
    provided to fill_icsi_table.
   returns: log(val). No input checking performed.
*/
inline float icsi_log(register float val,
        register const float *lookup_table, register const int n)
{
  register int *const  exp_ptr = ((int*)&val);
  register int         x = *exp_ptr; //x is treated as integer
  register const int   log_2 = ((x >> 23) & 255) - 127;//exponent
  x &= 0x7FFFFF; //mantissa
  x = x >> (23-n); //quantize mantissa
  val = lookup_table[x]; //lookup precomputed value
  return ((val + log_2)* 0.69314718); //natural logarithm
}
```