EFFICIENT FLOATING-POINT LOGARITHM UNIT FOR FPGAS

Nikolaos Alachiotis, Alexandros Stamatakis[*]

The Exelixis Lab
Dept. of Computer Science
Technische Universität München
email: {alachiot,stamatak}@in.tum.de

ABSTRACT

As FPGAs become larger, new fabrics, in particular DSPs, allow for a wider range of applications, specifically floating-point intensive codes, to be efficiently executed.

The logarithm is a widely used function in many scientific applications. We present the design of an efficient and sufficiently accurate Logarithm Approximation Unit (LAU) that uses a Look-Up Table (LUT) based approximation, in reconfigurable logic. The LAU has been verified through post place and route simulations, tested on actual FPGA, and is freely available for download. An important property of the LAU architecture is, that it only requires 2% of overall hardware resources on a medium-size FPGA (Xilinx V5SX95T) and thereby allows for easy integration with more complex architectures. Under single precision (SP) the LAU is 11 and 1.6 times faster than the GNU and Intel Math Kernel Library (MKL) implementations and up to 1.44 times faster than the FloPoCo reconfigurable logarithm unit, while occupying slightly less resources. Under double precision (DP) the LAU is 18 and 2.5 times faster than the GNU and Intel MKL implementations and up to 1.66 times faster than the FloPoCo logarithm while occupying significantly less resources.

The LUT-based approximation is sufficiently accurate for our target application and provides a flexible mechanism to adapt the LAU to specific accuracy requirements.

1. INTRODUCTION

A wide range of scientific applications rely on the computation of logarithms. Multimedia codes need to estimate, e.g., log likelihood scores for Gaussian Mixture Models [1] or Bioinformatics programs for evolutionary reconstruction under the Maximum Likelihood model [2] need to compute log likelihood scores of evolutionary trees. The logarithm is also frequently used to avoid underflow by replacing multiplications via additions.

Many of the applications that rely on the logarithm are either highly compute-intensive, such as the phylogenetic likelihood function which represents an important computational kernel in computational Biology [3, 4] or have real-time constraints, such as real-time image processing applications or skin segmentation algorithms [5]. Irrespective of the specific type of application, the deployment of reconfigurable logic (FPGAs) represents a common technique to either speed up applications, prototype hardware designs, or meet real-time requirements of time-critical applications.

Within the framework of our research on designing a reconfigurable Maximum Likelihood (ML) based phylogenetic co-processor for RAxML [6], we found that only one FPGA implementation for the logarithmic function on new-generation FPGAs is available [7]. One can use FloPoCo [8] to generate the aforementioned logarithm implementation. While this implementation provides high numerical precision, that might nonetheless not always be required, it has a relatively low clock frequency. Therefore, depending on the application, a significantly faster and slightly less accurate logarithm implementation may be preferable. To this end, we designed a reconfigurable floating point logarithm unit that operates at a significantly higher frequency than the existing FPGA implementation and returns an approximation of the logarithm. Since the logarithm is a fundamental function, we are convinced that this unit will be useful for a wide range of applications, beyond the scope of our own research.

We present the implementation of a pipelined LUT-based LAU in reconfigurable logic. The LAU design is highly efficient, both in terms of amount of reconfigurable fabric used (2% of hardware resources on a medium-size FPGA) as well as with respect to clock frequency and latency.

In addition, the LAU can be easily adjusted to the desired accuracy, based on the ICSILog approximation method (implemented in software) that has originally been proposed by Vinyals and Friedland in [9]. The speed of the original algorithm is achieved via a small LUT, that entirely fits in the cache of the CPU. The LUT-based approach led to a significant reduction in execution times [9] attaining speedups of up to a factor of 11 on different CPU architectures (In-

[*]This work is funded under the auspices of the Emmy-Noether program by the German Science Foundation (DFG)

tel Pentium 4, Intel Xeon, AMD Opteron 875, AMD 64 3000+, Intel Core Duo, AMD 64 X2) compared to the standard GNU library logarithm implementation.

The architecture we present here, represents a first step towards an efficient library of basic arithmetic floating-point functions for new-generation FPGAs. In addition, we extended the C implementation of the ICSILog algorithm (International Computer Science Institute) to support double precision (DP) arithmetic. We henceforth denote the single precision software implementation of ICSILog (version 0.6 beta) as SP-ICSILog and our DP software implementation as DP-ICSILog. Throughout the paper, we denote IEEE-754 single precision arithmetic as SP and IEEE-754 double precision arithmetic as DP. By SP-LAU and DP-LAU we denote the SP and DP FPGA implementations of the LAUs.

We compared the performance of our LAU to ICSILog, the GNU mathematical library, and the INTEL MKL (Math Kernel Library [10]) library implementations under SP and DP. Under SP, we measured speedups of 2 for the SP-LAU compared to SP-ICSILog, speedups of 11 compared to the GNU library function, and a speedup of 1.6 with respect to the Intel MKL logarithm function. Under DP, the DP-LAU achieves a speedup of 2.6 with respect to DP-ICSILog, a speedup of 18 compared to the GNU library, and a speedup of 2.5 compared to the respective MKL function. Furthermore, we compared the SP- and DP-LAUs to the FloPoCo logarithm (denoted as SP- and DP-FPLog) [7]. The SP-LAU occupies slightly less computational resources than SP-FPLog and is 1.44 times faster while DP-LAU occupies significantly less resources than DP-FPLog and is 1.66 times faster.

The DP-ICSILog C code as well as the hardware description are available as open source code for download at: `http://wwwkramer.in.tum.de/exelixis/logFPGA.tar.bz2`. The default hardware configuration that supports both, Virtex 4 and Virtex 5 FPGAs, uses a LUT with 4,096 entries. We also provide several COE files for different LUT sizes, such that the LAU can be conveniently reconfigured and adapted to the precision required by the respective target application.

The rest of the paper is organized as follows: Section 2 describes the underlying ideas of the ICSILog algorithm. In Section 3 we review related work on logarithmic units for FPGAs. Our FPGA architecture is described in Section 4. In the following Section 5, we present speed and accuracy measurements for the LAUs with a LUT-size of 4,096 entries. We also compare performance and resource utilization to the fast FPLog implementations, and assess numerical stability of RAxML in software using DP-ICSILog. We conclude in Section 6.

2. THE ICSILOG ALGORITHM

The underlying idea of the ICSILog algorithm consists of increasing the speed of the logarithm computation by using a LUT that resides entirely in the CPU cache. The algorithm exploits the way, by which floating point numbers are represented in the IEEE-754 standard. An IEEE floating point number consists of three fields: the sign (*sgn*), the exponent (*exp*), and the mantissa (*man*). The decimal floating point value of a number (*num*) is represented by the sign, followed by the product of the mantissa and the factor 2^{exp}:

$$num = (+/-)2^{exp} * man \tag{1}$$

In order to calculate the logarithm of *num*, one can use the multiplicative property of the logarithmic function and decompose the computation as follows:

$$\begin{aligned} log(num) &= log(2^{exp} * man) \\ &= log(2^{exp}) + log(man) \\ &= exp * log(2) + log(man) \end{aligned}$$

Since the real-valued logarithm is only defined for positive numbers, the sign bit can be discarded. The factor by which exp is multiplied is a constant and only depends on the base of the logarithm; one may use $log_e(2)$, $log_2(2)$, or $log_{10}(2)$ for instance. Thus, the calculation of the logarithm for an arbitrary base x, only requires the constant $log_x(2)$ and an appropriately initialized full-size LUT (comprising all values) for the base x.

The calculation of the first part of the sum: $exp * log(2)$ requires the floating point representation for the decimal value of the exponent field. One can use the Xilinx Floating Point Operator (FPO) [12] to obtain this value. However, we use a faster LUT-based method (this is a separate LUT that is exclusively used for this conversion) to obtain the floating point value which is described in Section 4. In Section 5 we provide a performance comparison between the Floating Point Operator provided by Xilinx and our approach. Once the floating point value of the exponent is available, the first operand of the final addition is calculated by conducting the multiplication with the constant floating point value.

The calculation of the second part of the sum, i.e., the logarithm of the mantissa requires the use of a LUT. A naïve LUT will thus need to contain *all* pre-computed values for $log(man)$ which requires 32MB of memory for the SP number range. Vinyals and Friedland found that the usage of a 32MB full-size LUT only yields insignificant performance improvements with respect to the GNU implementation [9]. To improve performance and reduce LUT size, they deploy a quantized mantissa that entirely fits into cache memory. In Figure 1 we provide a schematic outline of their algorithm at the bit level.

The mantissa LUT is indexed by using the $23 - q$ most significant bits of the mantissa under SP and the $52 - q$ most significant bits under DP. The variable q is the number of least significant bits of the mantissa that will be ignored by the quantization process. Thus, q can be used to appropriately adapt the accuracy and LUT size to the specific requirements of an application. An in-depth study about the accuracy loss induced by quantizing the mantissa can be

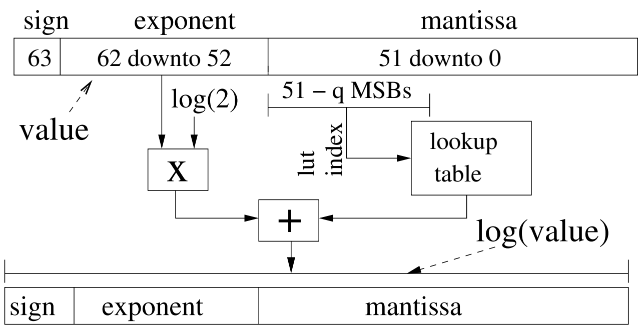

Fig. 1. Outline of the ICSILog Algorithm.

found in [9]. The trade-off between accuracy and embedded memory hardware resources will be discussed in Section 5.

3. RELATED WORK

A thorough bibliographical search revealed that alternative implementations of fast logarithm algorithms mostly represent special purpose solutions that are specifically targeted at a respective application or hardware platform, i.e., there is a lack of a generally applicable solution. Thus, to the best of our knowledge, our work represents the first implementation of a reconfigurable logarithmic approximation unit on new-generation FPGAs with DSPs as well as the first implementation of the ICSILog algorithm on reconfigurable logic.

There have been reported algorithms that calculate approximations of the logarithmic function [9, 11] in software in order to speed up specific multimedia applications. This underlines that the trade-off between precision and computational resources used is still an open problem and also strongly depends on the target application. In 2001 L. de Soras proposed and made available an algorithm called Fast-Log [11]. This algorithm computes a 3rd order Taylor series approximation of the logarithm for any given IEEE-754 floating point number. The algorithm is fast, but lacks accuracy in certain cases [9]. The LUT-based approach of ICSILog, which we implemented in reconfigurable logic, is as fast as Fast-Log, but provides better accuracy [9].

Recently, Dinechin *et al.* presented FloPoCo (Floating-Point Cores), an arithmetic core generator for FPGAs [8]. The logarithmic unit generated by FloPoCo (FPLog) supports SP (SP-FPLog), DP (DP-FPLog), as well as user-defined number formats. The FPLog units yield *exactly* the same results as the respective GNU functions, hence accuracy comparisons between our LAU and FPLog are identical to comparisons between the LAU and the GNU library.

Section 5 includes a direct comparison between the LAU and FPLog units (using the most recent version 1.15.1 of FloPoCo) in terms of speed and resource utilization on a Virtex 5 FPGA. It is worthwhile to point out that, the FPLog input coding slightly differs from the IEEE-754 standard.

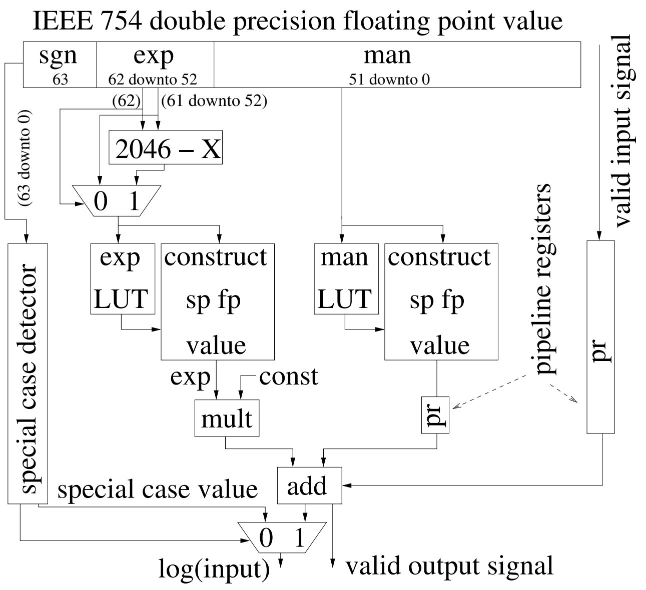

Fig. 2. Block diagram of LAU.

Two extra bits in every input vector indicate whether the input should be treated as one of the values: *zero*, *nan*, (+/-)*inf* or as a normal number. This partial lack of compliance with the IEEE-754 standard requires additional logic (which can also be separately generated by FloPoCo) to appropriately set these bits, in order to integrate an FPLog component into a design that relies on the IEEE-754 standard. Furthermore, a common FPGA design technique relies on developing event-driven architectures. The FPCLog interface does not include signals to indicate whether the values of the input and output ports are valid or not. As a consequence, the lack of such signals yields FPLog harder to integrate with event-driven architectures.

4. THE LAU ARCHITECTURE

In the following we describe the design of a reconfigurable architecture for the ICSILog algorithm. In Figure 2 we provide the block diagram of the top-level unit.

The leftmost module is the *special_case_detector*. As the name suggests, this module assesses whether the input to the LAU is valid or not. Special cases are: negative numbers, nan, $-inf$ and inf as defined by the IEEE standard. Since the logarithm is not defined on negative numbers the result is nan. For nan and $-inf$ inputs the result is defined as nan as well. For an inf input the unit will return inf again. The module consists of comparators, logic gates, and pipeline registers that detect the special case inputs and produce the respective output. The module also outputs a selection signal for the final 2to1 multiplexer (bottom left in Figure 2) that is connected to the output port of the LAU.

To the right of the *special_case_detector* in Figure 2, we have integrated a group of modules that operate on the input exponent bits. These modules compute the first operand of the addition that returns the approximation of the logarithm.

Initially, the decimal value of the exponent field needs to be transformed into a floating point number. The straightforward approach to implement this operation is to use the Xilinx FPO [12] (fixed-to-float) operator. However, we deploy a LUT-based approach to carry out this transformation more efficiently. The *exp_LUT* lookup table in Figure 2 is used for this purpose. Note that, this LUT is a special component of our hardware implementation and should not be confused with the mantissa LUT of the ICSILog algorithm (*man_LUT*). More details about the specific performance and resource trade-offs between our approach and the alternative design using the Xilinx FPO are provided in Section 5.3.

Internally, all operations are conducted under SP. For the SP-LAU the *exp_LUT* contains 128 entries (2^{8-1}), while for the DP-LAU there are 1024 entries (2^{11-1}), where 8 and 11 is the number of bits that represent the exponent field of a SP and DP value respectively. The reason why the size of the *exp_LUT* can be reduced by 50% is explained in the next paragraph. Each entry of the *exp_LUT* contains a total of 9 bits in the SP-LAU and 13 bits in the DP-LAU. The first 3 bits under SP and the first 4 bits under DP are the least significant bits of the exponent field of the floating point number representation we intend to construct. The remaining 6 (SP) and 9 bits (DP) are the most significant bits of the mantissa. The remaining bits of the exponent field are always set to `10000` for SP and `1000` for DP. Note that, at this point a SP value is being constructed for the DP-LAU as well. The remaining bits of the mantissa are all set to zero.

One can observe that there is a correspondence between the decimal values of the exponent field and the exponents themselves. For DP, while the decimal value ranges from 0 to 2,047 the exponent ranges from -1,023 to 1,024. Table 1 illustrates this correspondence which can be used to reduce the size of *exp_LUT* by 50%, via only storing the bits required to represent floating point numbers in the range 0-1,023. To support the full range (0-2,047) we use additional logic. More specifically, the 11-bit mantissa is transformed into a 10-bit index for *exp_LUT* by subtracting the 11-bit value from 2,046. For example, a 11-bit index with a decimal value X in which the most significant bit is set, indexes a lookup table entry $> 1,023$. Hence, $X - 1,023$ provides the distance from the last entry of the lookup table with 1,024 entries. Thus, $1,023-(X-1,023) = 2,046-X$ will yield the correct 10-bit index for a *exp_LUT* with half the size. The most significant bit of the exponent field (discarded from the index) becomes the sign of the newly constructed floating point value. After this transformation, the resulting floating point number becomes the first operand of the multiplication; the second operand is a constant value.

decimal value of exponent field	-1023	exponent
0	-1,023	-1,023
. . .	-1,023	. . .
1,023	-1,023	0
. . .	-1,023	. . .
2,047	-1,023	1,024

Table 1. Correspondence between the decimal value of the exponent field and the exponent.

The overall result produced by this part of the architecture is the first operand of the final addition: $exp * log(2) + log(man)$. The *man_LUT* module in Figure 2 is the standard quantized LUT of the ICSILog algorithm and contains pre-calculated values of logarithms. We therefore used ICSILog to generate the contents of *man_LUT*. As previously described, the most significant bits of the mantissa are used for indexing the *man_LUT*. Each entry of the table (for SP and DP values) consists of a SP floating point number. As outlined in the next Section one can increase the accuracy of the LAU by increasing the size of *man_LUT*. For example, for a *man_LUT* of size 4,096, only the 12 most significant bits of the mantissa field of the input value will be used for indexing. Both lookup tables (*exp_LUT* and *man_LUT*) are enhanced by a *construct_sp_fp_value* unit. These units consist of logic gates, registers, and multiplexers which are used to construct the correct floating point representations from the respective LUT entries. Finally, the sum of the two values generated by *exp_LUT*, *man_LUT*, and the respective *construct_sp_fp_value* units will return an approximation of the logarithm that is identical to the ICSILog software.

As already mentioned, all operations are conducted under SP. Thus, for the SP-LAU, the result is simply the output of the final adder. For DP, the result is transformed into DP by appropriately adapting the bit indices of the SP representation. The least significant bits of the mantissa are set to zero, and a bit extension for the most significant bits of the exponent is conducted while maintaining its sign.

The usage of SP arithmetic, even for the DP-LAU, does not affect the precision of the output because of the approximation strategy that is being used. DP will only be affected if a *man_LUT* with more than 2^{23} entries is used (23 is the number of bits in the mantissa field of IEEE SP numbers). In this case the mantissa LUT would require 32MB of memory. Currently, there is no FPGA available with such a large amount of embedded memory. Clearly, the savings in terms of FPGA resources (embedded memory and DSP slices) by internally using SP for our LAU design are significant. Note that, in our DP-ICSILog software implementation, we transformed the entire algorithm to DP, because the SP algorithm with a type casting operation from `float` to `double` in C was slower than a direct implementation under DP.

# block rams (18Kb)	LUT entries	average error
1	512	0.000352
2	1,024	0.000176
3	2,048	0.000088
6	4,096	0.000044
12	8,192	0.000022
24	16,384	0.000011
48	32,768	0.000005

Table 2. Average LAU error and *man_LUT* # of block rams.

5. EXPERIMENTAL RESULTS

Initially, we verified the functionality of the LAU (Section 5.1) and assessed its accuracy (Section 5.2). Thereafter, we provide a detailed resource usage analysis (Section 5.3). A thorough run time comparison is presented in Section 5.4. Section 5.5 presents a performance and resource utilization comparison with the FloPoCo logarithm [7]. Finally, in Section 5.6 we investigate the behavior of RAxML [6] using DP-ICSILog. Note that, all results in Section 5 refer to Xilinx reports after the implementation process (post place and route).

5.1. Verification

In order to verify correctness of the proposed architecture, we conducted extensive post place and route simulations as well as tests on an actual FPGA. As simulation tool we used Modelsim 6.3f by Mentor Graphics. For hardware verification we used the HTG-V5-PCIE development platform equipped with a Xilinx Virtex 5 SX95T-2 FPGA. The advanced verification tool Chipscope Pro Analyzer was used to monitor the output port of the SP- and DP-LAUs and the expected signals for given input numbers were tracked.

5.2. Accuracy Assessment

Initially, we used benchmarks with $2 * 10^7$ random numbers to measure the average error introduced by the logarithm approximation with respect to the GNU function for implementations of the LAU with various LUT sizes. The results are provided in Table 2. We used the ICSILog software to generate the contents of *man_LUT*, such that it yields *exactly* the same results as ICSILog. From Table 2 we deduce that a LUT with 4,096 entries represents a good trade-off between accuracy and LUT size for our purposes, since a LUT of this size only requires 3 block rams (36Kb each). For a medium-size new-generation FPGA, like the Xilinx Virtex 5 SX95T, 3 block rams correspond to only 1% of the total block memory available. As discussed in [9] the size of the LUT increases exponentially for every additional correct bit in the mantissa. Clearly, a specific target application

Program/Unit	Min	Max	Avg	MSE
SP-ICSILog	4.228e-7	1.210e-4	4.438e-5	2.689e-9
DP-ICSILog	3.140e-9	1.205e-4	4.437e-5	2.688e-9
DP-LAU	4.228e-7	1.210e-4	4.437e-5	2.690e-9
SP-MKL	0.0e-0	3.815e-6	5.003e-7	1.65e-14
DP-MKL	0.0e-0	4.44e-16	4.52e-22	4.93e-38

Table 3. Min, max, average, and mean squared error of logarithm implementations with respect to GNU functions.

Resources	LUT	FPO
Slice registers	32	46
Slice LUTs	19	64
Occupied slices	20	19
# of LUT Flip Flop pairs	48	45
# of BRAMS (18Kb)	1	0

Table 4. Resource usage by LUT-based approach and Xilinx FPO for transformation of exponent to SP number.

as well as a global view of the entire reconfigurable system that will use the LAU is required to determine the ideal *man_LUT* size. Since the software implementation is available as open-source code, it is easy to assess the required mantissa LUT size a priori, i.e., before modifying the reconfigurable architecture.

In our specific case (RAxML) we found that a size of 4,096 entries is sufficient to ensure numerical stability of the code and accurate results (see Subsection 5.6). The overall architecture for RAxML requires memory and reconfigurable fabric for other purposes. Therefore, we chose to minimize the hardware resources for the execution of the logarithmic function to the largest possible extent.

For a *man_LUT* with 4,096 entries we also measured the minimum, maximum, average, and mean squared error between the GNU SP and DP library functions and the respective logarithmic approximation implementations: SP-/DP-ICSILog, DP-LAU, SP-/DP-MKL library functions. Table 3 provides these errors for 10^6 random input numbers ranging from 10^{-20} to 10^{20}.

5.3. Resource Requirements

The LAU was mapped to a Xilinx Virtex 5 SX95T-2 FPGA. The columns of Table 11 that refer to the LAUs provide the amount of computational resources occupied by the SP-LAU and DP-LAU with a LUT size of 4,096. Table 11 demonstrates that we have achieved one of our major design goals, i.e., to devise a sufficiently accurate logarithmic unit by using a minimum of resources.

The clock frequencies of the LAUs were measured using the respective Xilinx Tools (ADVANCED 1.53 speed file)

# samples	SP-GNU	SP-ICSILog	SP-LAU
10^3	0.03290	0.00620	0.00301
10^6	32.40	6.31	2.93
10^8	3315	595	293

Table 5. Execution times (in ms) of GNU, ICSILog, and LAU SP implementations for 10^3 up to 10^8 invocations.

# samples	DP-GNU	DP-ICSILog	DP-LAU
10^3	0.07220	0.01191	0.00319
10^6	58.40	9.47	3.12
10^8	5909	899	312

Table 6. Execution times (in ms) of GNU, ICSILog, and LAU DP implementations for 10^3 to 10^8 invocations.

# samples	SP-MKL	SP-ICSILog	SP-LAU
10^6	4.7	5.3	2.9
10^7	46.9	50.2	29.3
10^8	342.9	486.6	292.7

Table 7. Execution times (in ms) of MKL, ICSILog, and LAU SP implementations compiled with icc.

# samples	DP-MKL	DP-ICSILog	DP-LAU
10^6	8.0	8.8	3.1
10^7	77.2	85.1	31.2
10^8	668.4	839.7	311.9

Table 8. Execution times (in ms) of MKL, ICSILog, and LAU DP implementations compiled with icc.

and are shown in Table 12. The clock frequencies are obtained from the static timing report and refer to LAUs with *man_LUT* size of 4,096 entries. The SP- and DP-LAUs have the same latency in terms of clock cycles (22) and are fully pipelined with a throughput of one result per cycle.

In Table 4 we compare the hardware resources used by our custom LUT-based module and the Xilinx FPO [12] (configured in fixed-to-float mode) for transforming the exponent value into a floating point value.

Since the LUT-based approach has a latency of two cycles, we configured the Floating Point Operator to have the same latency and integrated it into the LAU. We also added an 11-bit subtractor such that the LAU produces correct results. The clock frequency of the LAU using the Floating Point Operator was 60MHz slower than for our LUT-based approach. When the FPO is configured with the maximum latency of 6 cycles, the LAU is 5MHz faster than with the LUT-based approach. However, in this case the total latency of the LAU increases from 22 to 26 cycles and the FPO requires a larger amount of hardware resources.

5.4. Performance Assessment versus Software

In order to conduct a fair performance assessment of the LAU, we compared it to a wide range of software implementations: the SP-/DP-GNU logarithms: `logf()`/`log()`, the SP-/DP-MKL logarithms: `vsLn()`/`vdLn()`, and the SP-/DP-ICSILog algorithms. As hardware platform we used a V5SX95T-2 FPGA (speed grade -2) with one LAU. The software implementations were executed on an Intel Core2 Duo T9600 processor running at 2.8GHz with 6MB of L2 Cache. All software (SP-/DP-ICSILog) and hardware implementations (SP-/DP-LAU) tested used a mantissa LUT with 4,096 entries.

For software tests, we used the GNU `gcc` compiler (version 4.3.2) as well as the Intel `icc` compiler (version 11.1) in order to fully exploit the capabilities of the Intel CPU. We only used `-O1` for optimization with `gcc` because with more aggressive optimizations (`-O2` and `-O3`) the current SP-ICSILog version yields an average error that is 10^5 times larger than the error obtained by compiling the code with `-O1`. Thus, the `gcc` compiler optimizations applied under `-O2` and `-O3` yield numerically unstable code. When `icc` is used, SP-ICSILog produces the expected average error, which is in the range of 10^{-5} for all optimization levels (`-O1`, `-O2`, `-O3`). When `-O2` or `-O3` is used with `icc`, SP-ICSILog is only 1.09 times faster on average than the GNU math library. However, when `-O1` is used, SP-ICSILog is on average 4.5 times faster.

Initially, we used the GNU `gcc` compiler (version 4.3.2, with `-O1`) and measured the execution times for 10^3 up to 10^8 invocations of the GNU library SP function as well as SP-ICSILog. Note that, we used the most recent version of the SP-ICSILog algorithm, which is faster than the initial release of the ICSILog software. According to the benchmark that is made available by the authors the current version is approximately 1.7 times faster than the initial version (when compiled with `gcc` and `-O1`). Table 5 shows the execution times for the GNU implementation, SP-ICSILog, and the SP-LAU. The SP-LAU is 11 times faster than the GNU function and two times faster than SP-ICSILog.

As already mentioned, the standard release of ICSILog is only available for SP input values. Furthermore, it does not provide built-in error detection/correction for special-case inputs like $nan, inf, -inf$ or negative numbers which is critical for applications like RAxML. In order to conduct a fair performance evaluation of the DP-LAU, we therefore re-implemented the ICSILog algorithm to support DP inputs and invalid input detection. Our new DP version of ICSILog (DP-ICSILog) is only 1.5 times slower than the official SP release by Vinyals and Friedland. DP-ICSILog is also freely available for download together with the LAU architecture.

# samples	DP-GNU		DP-ICSILog	
	-O2	-O3	-O2	-O3
10^3	0.0779	0.0758	0.0119	0.0119
10^6	57.82	57.34	8.50	8.42
10^8	5,692	5,678	799	798

Table 9. Execution times (in ms) of GNU, ICSILog, and LAU DP implementations compiled with gcc and -O2/-O3.

# samples	DP-MKL		DP-ICSILog	
	-O2	-O3	-O2	-O3
10^6	8.0	8.0	8.1	8.1
10^7	64.1	60.9	77.9	77.7
10^8	619.6	601.8	769.8	769.6

Table 10. Execution times (in ms) of MKL, ICSILog, and LAU DP implementations compiled with icc and -O2/-O3.

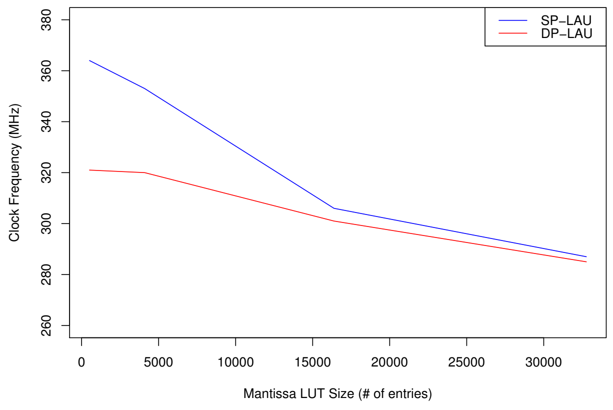

Fig. 3. LAU frequencies with respect to LUT size.

For assessing DP performance we used `gcc` (`-O1`) and measured execution times for 10^3 up to 10^8 invocations of the GNU, DP-ICSILog, and DP-LAU logarithm functions (Table 6). The DP-LAU is 18.8 times faster than the GNU math library and 3 times faster than DP-ICSILog which in turn is up to 6.5 times faster than the GNU implementation.

For our second set of experiments, we used the Intel `icc` compiler (version 11.1, optimization flag `-O1`). We also used the fast logarithm implementation provided by the Intel Math Kernel Library (MKL) for 10^6 to 10^8 invocations on random numbers as in the preceding experiments.

Tables 7 and 8 show the execution times for the SP and DP MKL, ICSILog and LAU implementations respectively. The SP-LAU is 1.6 times faster than the MKL logarithm and 1.8 times faster than SP-ICSILog. Unfortunately, a detailed description of the MKL logarithm implementation is currently not available. The DP-LAU is 2.4 times faster than the respective MKL implementation and 2.7 times faster than DP-ICSILog which is almost as fast as the DP-MKL function (speedups vary from 0.8 to 0.9).

As already mentioned, SP-ICSILog becomes unstable when optimization flags `-O2` or `-O3` are used with `gcc`. Therefore, we only assessed the performance impact of using `-O2` and `-O3` with `gcc` on DP-ICSILog. We compare DP-ICSILog execution time against the remaining DP implementations: DP-GNU, DP-MKL, and DP-LAU. Table 9 provides the execution times of DP-GNU and DP-ICSILog for 10^3 to 10^8 invocations of the `gcc`-compiled code. The DP-LAU is 18.8 times faster than the GNU math library and 2.6 times faster than DP-ICSILog which in turn is up to 7 times faster than the GNU implementation (for `-O2` as well as `-O3`). Table 10 provides respective execution times under DP for the same experimental setup, but using the Intel `icc` compiler instead. The DP-LAU is 2.2 times faster than the respective MKL implementation and 2.5 times faster than DP-ICSILog which is as fast as the DP-MKL function. Speedups between DP-ICSILog and the DP-MKL function vary between 0.83 and 0.98 for both optimization levels `-O2` and `-O3`.

Finally, Figure 3 shows the clock frequencies of the SP-LAU and DP-LAU for *man_LUT* sizes ranging between 512 up to 32,768 entries. The observed frequency reduction is because of the increasing logic requirements of the LAU, basically block rams. The number of block rams required increases exponentially for every extra bit added to the mantissa field which is used as an index for *man_LUT* while the increase of the other resources is significantly slighter, i.e., a LAU with a 32,768 entry *man_LUT* size occupies 700% more 36Kb block rams than a LAU with a 4,096 entry *man_LUT* size while only 15% more slices and 9% more Slice LUTs are required.

5.5. Performance Assessment versus Hardware

To the best of our knowledge, the only open-source logarithm for FPGAs is the one provided by FloPoCo [8]. As mentioned in Section 3 the FPLog operator yields the same results as the GNU logarithm, thus we only compared the SP-/DP-LAUs to the SP-/DP-FPLogs in terms of performance and resource efficiency. Table 11 provides the hardware resources occupied on a Virtex 5 SX95T-2 FPGA. The SP-LAU occupies slightly less resources than SP-FPLog while DP-LAU needs significantly less resources than DP-FPLog. Table 12 provides a performance comparison between the logarithm implementations. The SP- and DP-LAUs operate at significantly higher frequencies than SP-/DP-FPLogs. Thus, the FPLog units are more likely to lie on the critical path of a larger architecture that needs to calculate logarithms.

Resources-Total	SP		DP	
	FPLog	LAU	FPLog	LAU
slice registers-58,800	992	932	2,763	970
slice LUTs-58,800	909	621	2,482	634
occupied slices-14,720	375	363	795	360
# 36k blockRAM-244	2	3	4	3
# 18k blockRAM-488	2	1	18	1
# DSP48Es-640	5	3	14	3

Table 11. Resource usage: LAUs vs FPLogs.

Performance	SP		DP	
	FPLog	LAU	FPLog	LAU
Clock Latency	20	22	34	22
Frequency	244.7	353.5	192.3	320.6

Table 12. Performance comparison: LAUs vs FPLogs.

5.6. DP-ICSILog in a Real-World Application

We integrated DP-ICSILog into RAxML [6], which is a widely used tool for inferring phylogenies (evolutionary trees) from molecular data that has been developed in our group. The vast majority of logarithm invocations is conducted when the log likelihood scores for alternative tree topologies are computed. Table 13 indicates the respective log likelihood scores for tree searches using the GNU and DP-ICSILog implementations on various DNA datasets with 150, and 218 organisms (sequences) as well as a protein dataset with 140 organisms. Based on the standard statistical significance tests used in phylogenetics, the difference of log likelihood scores among the respective trees is not statistically significant, hence DP-ICSILog with a LUT size of 4,096 provides sufficient accuracy for our application.

6. CONCLUSION & FUTURE WORK

We presented a new architecture that efficiently calculates an approximation of the logarithm in reconfigurable logic under SP and DP arithmetic and only uses 2% of resources on medium-size FPGAs. The SP-/DP-LAUs (LUT size:4,096) as well as the DP software are freely available. To the best of our knowledge, this represents the only IEEE-754 compatible implementation of a resource-efficient logarithm approximation unit in reconfigurable logic. Since the accuracy demands of such a basic unit strongly depend on the target application, we also make available several COE files that can be used to initialize LUTs of various sizes and hence easily adapt the LAUs to the desired accuracy level. Except for an increase of block ram usage to hold the mantissa LUT, the proportion of required hardware resources will only slightly increase (see Section 5.4) if the LUT size is increased and the speed will only slightly decrease (see Figure 3). Future work will focus on the development of a resource-efficient unit for the exponential function.

Dataset	DP-GNU	DP-ICSILog
150 organisms	-39606.31	-39606.60
218 organisms	-134173.86	-134167.56
140 organisms	-124777.22	-124780.10

Table 13. Log-likelihood score deviation with DP-ICSILog.

7. REFERENCES

[1] D. Ververidis, C. Kotropoulos, "Gaussian Mixture Modeling by Exploiting the Mahalanobis Distance," in *Proc. of IEEE trans. on Signal Proc.*, vol.56, no.7, pp.2797-2811, July 2008.

[2] J. Felsenstein, "Evolutionary trees from DNA sequences: a maximum likelihood approach," *Journal of Molecular Evolution.*, vol. 17, pp. 368–376, 1981.

[3] N. Alachiotis, E. Sotiriades, A. Dollas, and A. Stamatakis, "Exploring FPGAs for Accelerating the Phylogenetic Likelihood Function," in *Proceedings of HICOMB2009*, Rome, Italy, May 2009.

[4] N. Alachiotis, A. Stamatakis, E. Sotiriades, and A. Dollas, "A Reconfigurable Architecture for the Phylogenetic Likelihood Function," in *Proc. of FPL 2009*, Prague, September 2009.

[5] B. Ruijsscher, G. Gaydadjiev, J. Lichtenauer, and E. Hendriks, "FPGA accelerator for real-time skin segmentation," in *Proc. of the IEEE Workshop on Embedded Systems for Real Time Multimedia* , vol., no., pp. 93–97, Octomber 2006.

[6] A. Stamatakis, "RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models," *Bioinf.*, vol. 22, no. 21, pp. 2688–2690, 2006.

[7] J. Detrey, F. de Dinechin, "Parameterized floating-point logarithm and exponential functions for FPGAs," *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing, 31(8):537-545, 2007.*

[8] F. de Dinechin, C. Klein, B. Pasca, "Generating high-performance custom floating-point pipelines," *Proc. of FPL 2009.*

[9] O. Vinyals, G. Friedland, "A Hardware-Independent Fast Logarithm Approximation with Adjustable Accuracy," *Tenth IEEE Inter. Symposium on Multimedia*, pp. 61–65, 2008.

[10] Intel, "Intel Math Kernel Library Reference Manual," www.intel.com/software/products/mkl/docs/WebHelp/mkl.htm

[11] L. de Soras, "Fast log() Function," (last visited: 03-07-2009), www.flipcode.com/cgi-bin/fcarticles.cgi?show=63828.

[12] Xilinx, "Floating Point Operator v4.0, " (last visited: 03-07-2009), http://www.xilinx.com/support/ip_documentation/floating_point_ds335.