Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 341510, 12 pages
doi:10.1155/2011/341510

Research Article

A Vector-Like Reconfigurable Floating-Point Unit for the Logarithm

Nikolaos Alachiotis and Alexandros Stamatakis

The Exelixis Lab, Scientific Computing Group, Heidelberg Institute for Theoretical Studies, 69118 Heidelberg, Germany

Correspondence should be addressed to Nikolaos Alachiotis, n.alachiotis@gmail.com

Received 27 July 2010; Accepted 14 January 2011

Academic Editor: Aravind Dasu

The use of reconfigurable computing for accelerating floating-point intensive codes is becoming common due to the availability of DSPs in new-generation FPGAs. We present the design of an efficient, pipelined floating-point datapath for calculating the logarithm function on reconfigurable devices. We integrate the datapath into a stand-alone LUT-based (Lookup Table) component, the LAU (Logarithm Approximation Unit). We extended the LAU, by integrating two architecturally independent, LAU-based datapaths into a larger component, the VLAU (vector-like LAU). The VLAU produces 2 results/cycle, while occupying the same amount of memory as the LAU. Under single precision, one LAU is 12 and 1.7 times faster than the GNU and Intel Math Kernel Library (MKL) implementations, respectively. The LAU is also 1.6 times faster than the FloPoCo reconfigurable logarithm architecture. Under double precision, one LAU is 20 and 2.6 times faster than the respective GNU and MKL functions and 1.4 times faster than the FloPoCo logarithm. The VLAU is approximately twice as fast as the LAU, both under single and double precision.

1. Introduction

The use of FPGAs as accelerators for compute-intensive codes is driven by their potential for implementing deeply pipelined architectures and for executing hundreds of operations in parallel. As the devices become larger, new fabrics, in particular DSPs, allow for a wider range of applications, in particular floating-point intensive codes, to be efficiently executed/accelerated on FPGAs.

A large number of scientific applications rely on the frequent and efficient computation of the logarithm function. For instance, multimedia codes need to estimate 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 commonly used to avoid numerical underflow (especially in statistics) 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 exhibit real-time constraints, such as real-time image processing applications [5] or skin segmentation algorithms [6]. 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 to meet real-time requirements of time-critical applications.

When an FPGA is used for accelerating floating-point intensive applications, a thorough exploration of the performance and precision tuning parameter space for the required arithmetic operators can eventually lead to significant performance improvements. In fact, implementations of simple floating-point operators like adders or multipliers may require fairly complex reconfigurable architectures. Furthermore, the amount of hardware resources used by a floating-point operator generally increases with precision/accuracy requirements. However, accuracy requirements of a specific operator may depend on the application at hand.

In previous work on calculating the logarithm function [7] using reconfigurable logic, we focused on the design of a pipelined Logarithm Approximation Unit (LAU). We demonstrated that the LAU is sufficiently accurate for

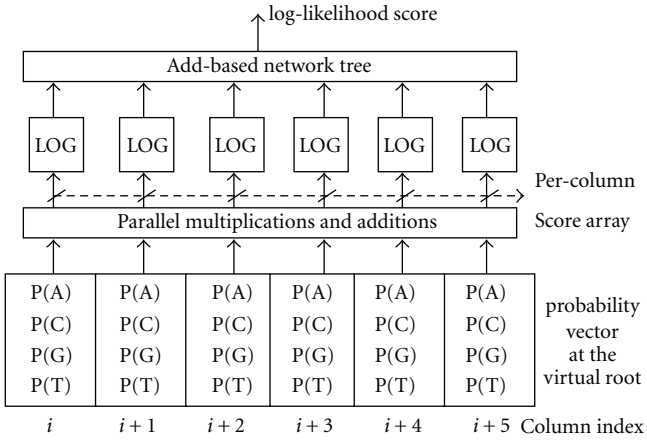

Figure 1: Parallel placement of logarithm units for the calculation of the log-likelihood score for a phylogenetic tree topology.

computing the phylogenetic maximum likelihood (ML) function on a reconfigurable coprocessor for RAxML [8]. RAxML is a widely used bioinformatics code for reconstructing evolutionary trees (evolutionary histories or simply phylogenies) from DNA or protein data under the ML criterion. The LAU architecture utilizes look up tables (LUTs) for calculating the logarithm and can be conveniently adjusted to provide the desired/required application-specific accuracy. The LAU is based on the ICSILog approximation method (Vinyals and Friedland in [9]) that is available as open-source code. If not stated otherwise, in this paper, we use the term LUT for referring to the look up table required by the ICSILog approximation method rather than to the low-level hardware LUTs on the FPGA device.

As already mentioned, several computational units must be placed on the FPGA and operate in parallel to efficiently exploit the available computational resources. Thus, the resource requirements of a component/unit (e.g., a simple floating-point operator or a more complex arithmetic function) need to be minimized to allow for placing several instances on the chip that will then operate in parallel. The input/output (I/O) requirements can be accommodated by parallel I/O ports, for instance, by organizing the embedded memory blocks of a device into several smaller parallel blocks that can provide a sufficient throughput with respect to the arrangement of the parallel components. In Figure 1, we provide a representative example for the potential arrangement of logarithm components and the respective parallel execution of the logarithm function. The block diagram depicts a fine-grain parallelization of log-likelihood score computations for a given evolutionary tree topology under a likelihood-based model (used, e.g., in maximum likelihood or Bayesian phylogeny programs).

In the current paper, we present a significantly extended and optimized vector-like LAU implementation. The vector-like Logarithm Approximation Unit (VLAU) can calculate two logarithms within the same clock cycle. Using the VLAU is more resource-efficient compared to instantiating and using two simple independent LAUs in parallel. The underlying idea of the VLAU consists of exploiting the dual-port configuration option of embedded memory blocks. This implementation option allows for sharing LUTs between two, otherwise completely independent, LAU-based pipelines. Furthermore, a detailed analysis of resources requirements and performance impact with respect to the latency of the LAU has been conducted for the single and double precision versions. We also extended the C implementation of the ICSILog algorithm (International Computer Science Institute) to support double precision (DP) arithmetics.

Throughout the paper, we denote IEEE-754 single precision arithmetics as SP and IEEE-754 double precision arithmetics as DP. We denote the single precision software implementation of ICSILog (version 0.6 beta) as SP-ICSILog and our DP software implementation as DP-ICSILog. By SP-LAU, DP-LAU, SP-VLAU, and DP-VLAU we denote the SP and DP FPGA implementations of the LAUs and VLAUs, respectively.

The DP-ICSILog C code as well as the hardware descriptions for the LAUs and the VLAUs (including all available latency variants) are available as an 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 an LUT with 4,096 entries. We also provide several COE files for different LUT sizes, such that the LAU/VLAU can be conveniently reconfigured and adapted to the precision required by the respective target application.

The remainder of this 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. The LAU architecture is described in Section 4, and the VLAU architecture is introduced in Section 5. In the following Section 6, we present speed and accuracy measurements for LAUs and VLAUs with a LUT-size of 4,096 and provide a detailed evaluation of LAU implementations with latencies ranging between 5 and 22 clock cycles. We also analyze performance and resource utilization of the FPLog implementations and assess the numerical stability of RAxML in software using DP-ICSILog. We conclude in Section 7.

2. The ICSILog Algorithm

The underlying idea of the ICSILog algorithm consists of increasing the speed of the logarithm computation by using an LUT that resides entirely in the CPU cache. The algorithm exploits the floating point number representation of 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}:

$$\text{num} = (\pm)2^{\text{exp}} * \text{man}. \quad (1)$$

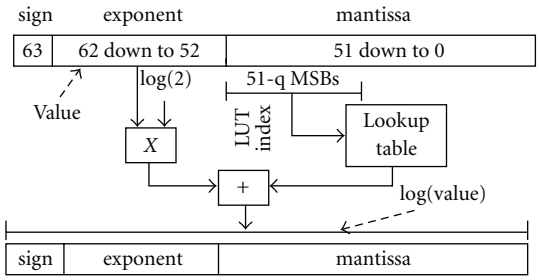

Figure 2: Outline of the ICSILog Algorithm.

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

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

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 $\text{exp} * \log(2)$ requires the floating-point representation for the decimal value of the exponent field. One can use the Xilinx floating-point operator (FPO) [10] 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 6, 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, that is, the logarithm of the mantissa, requires the use of an LUT. A naïve LUT will thus need to contain all precomputed values for *log(man)* which requires 32 MB of memory for the SP number range. Vinyals and Friedland found that, the usage of a 32 MB 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 2, we provide a schematic outline of the Vinyals and Friedland 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, respectively. The variable q is the number of least significant bits of the mantissa that will be ignored by the quantization process. Thus, the variable q can be used to appropriately adapt the accuracy and LUT size to the specific requirements of an application. A detailed study of the accuracy loss that is induced by quantizing the mantissa can be found in [9]. The tradeoff between accuracy and embedded memory hardware resources used will be discussed in Section 6.

3. Related Work

A thorough bibliographical search revealed that alternative implementations of fast logarithm algorithms mostly represent special purpose solutions that are tailored to a specific application or hardware platform, that is, there is a lack of a generally applicable solution.

Dedicated software implementations that entail approximation algorithms for the logarithmic function have been developed for accelerating multimedia applications [9, 11]. In 2001, 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/number ranges [9]. The LUT-based approach of ICSILog, which we implemented in reconfigurable logic in the LAU and VLAU components, is as fast as fast log but provides better accuracy [9].

De Dinechin et al. have developed FloPoCo (floating-point cores), an open-source arithmetic core generator for FPGAs [12]. The logarithmic unit generated by FloPoCo (FPLog) supports SP (SP-FPLog), DP (DP-FPLog), and user-defined number formats. The FPLog units can be configured to 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. The algorithms and implementation techniques that are deployed in FloPoCo for generating the FPLog unit are described in [13, 14].

Section 6 includes a direct comparison between the LAU and FPLog units (using the most recent version 2.0.0 of FloPoCo) in terms of speed and resource utilization on a Virtex 5 FPGA. Note that the FPLog input format slightly differs from the IEEE-754 standard. Two additional bits in every input number indicate whether the input should be treated as special number (*zero, nan,* (+/−) *inf*) or as normal number. Thus, in order to integrate the FPLog component into a design that complies with the IEEE-754 standard, this dedicated input number format specification requires additional logic (which can also be separately generated by FloPoCo) to appropriately set these bits. Furthermore, a common FPGA design paradigm is event-driven architectures. Unfortunately, the FPLog interface does not provide any additional ports for validity input signals, that is, signals that indicate whether the current values at the input and output ports are valid or not. Consequently, FPLog is harder to integrate with event-driven architectures.

National instruments [15] have designed a high-throughput natural logarithm function for FPGAs. The specific design is only available commercially, and only a limited amount of information is provided regarding unit performance. The implementation only supports fixed-point arithmetics, and the input arguments must be unsigned

(input number range: $[1/e, 1)$). For numbers outside this input range, the unit generates undefined results. The interface of the logarithm component provides all necessary ports for validity input signals for easy integration with and use in event-driven environments. The CORDIC algorithm (COordinate Rotation DIgital Computer [16]) is deployed for the specific implementation, and the user can set the desired accuracy level by defining the number of iterations in the CORDIC algorithm. Because this logarithm implementation is not available (not even for a short evaluation period), we were not able to conduct a respective performance comparison with the LAU/VLAU architectures.

Tropea [17] has also presented an area-optimized FPGA implementation to compute the base-N logarithm function. An important aspect of the specific implementation is that it can be mapped on FPGAs from any vendor. Performance results for Xilinx [18] and Actel [19] FPGAs are provided in [17]. The error analysis section in [17] reveals that highly accurate results can be obtained, while only using a small fraction of the overall hardware resources. The unit utilizes the multiplicative normalization method [20] to calculate the logarithm, and several configurations with various precision levels are evaluated.

Recently, Chrysos et al. [21] presented a general reconfigurable architecture for a Bioinformatics algorithm that uses Interpolated Markov Models (IMMs) for gene finding, known as the Glimmer algorithm. The Glimmer algorithm also requires logarithm calculations. The respective hardware architecture contains 6 logarithm unit instances that operate in parallel. The design of the logarithm component in the Glimmer architecture [21] deploys a similar strategy as the LAU.

In 2008, Raygoza-Panduro et al. [22] presented an automatically generated mathematical unit. The hardware description is automatically generated by a JAVA program and can be synthesized. The framework supports a wide range of complex arithmetic operations. The mathematical unit was used to implement a sliding mode controller for a magnetic levitation system. The sytem provides operators for the natural logarithm and the base-10 logarithm. The automatically generated mathematical unit was mapped to a Virtex II FPGA; the logarithm functions (natural and base-10) only occupy 1% of available FPGA slices and 1% of available LUTs on the device. The resource-efficiency of the unit appears to be mainly induced by the usage of bit-width reduced floating-point arithmetics, that is, only 3 bits are used for the exponent field and only 14 bits for the mantissa field, respectively.

4. The LAU Architecture

In the following, we describe the design of a reconfigurable architecture for the ICSILog algorithm. In Figure 3, 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 LAU input 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

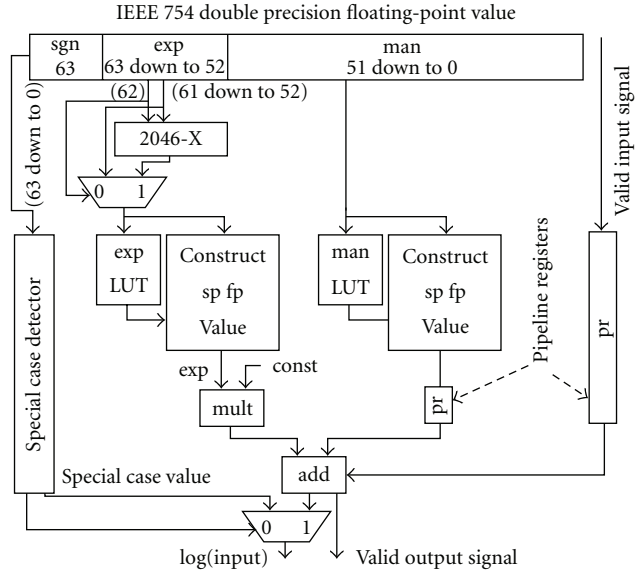

Figure 3: Block diagram of LAU.

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 corresponding output. The module also outputs a selection signal for the final 2 to 1 multiplexer (bottom left in Figure 3) that is connected to the output port of the LAU.

To the right of the *special_case_detector* in Figure 3, we have integrated a group of modules that operate on the exponent bits of the input. 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 [10] (fixed-to-float) operator. However, we deploy an LUT-based approach to carry out this transformation more efficiently. The *exp_LUT* lookup table in Figure 3 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*). Details about the performance and resource tradeoffs between our approach and the alternative design using the Xilinx FPO are provided in Section 6.4.

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 1,024 entries (2^{11-1}), where 8 and 11 are the number of bits that represent the exponent field of an SP and a 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, an 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. This correspondence 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, an 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. The overall result produced by this part of the architecture is the first operand of the final addition: $\exp * \log(2) + \log(\text{man})$.

The *man_LUT* module in Figure 3 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 an SP floating-point number. As outlined in Section 6, one can increase the accuracy of the LAU by increasing the size of *man_LUT*. For example, in 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 SP numbers in the IEEE standard). In this case, the mantissa LUT would require 32 MB 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.

5. The VLAU Architecture

An additional optimization can be applied to the LAU architecture (Section 4), when several parallel LAUs shall be placed on an FPGA device. This optimization is based on a special feature of embedded memory blocks in new-generation FPGAs, which can be configured as so-called dual-port memories.

Each memory block provides two fully independent ports that yield access to a shared memory space. An appropriate reconfiguration of the LAU look up tables (*exp_LUT*, *man_LUT*) for usage as dual-port ROMs (Read Only Memories) allows two independent LAUs to use the same memory blocks for lookups.

Figure 4 depicts the optimized VLAU architecture (vector-like LAU). The shared memory area in the middle of Figure 4 (denoted as *shared LUTs*) contains the *exp_LUT* and *man_LUT* look up tables. The two LAU-based pipelines are located to the left and the right of the *shared LUTs* in Figure 4. These two pipelines are exact copies of the LAU architecture (Section 4), but the LUTs have been moved to a *shared* memory area. The individual LAU pipelines are architecturally completely independent from each other, since they only share a read-only memory area. Each LAU pipeline only accesses one of the two ports of the shared LUTs.

The VLAU architecture is well suited for vector-processing, since it can accommodate the computation of two logarithms in one cycle. Because the same pipeline design is used for the LAU and VLAU, a two-unit VLAU is as fast as two independent LAUs. The main advantage of a two-unit VLAU over two independent LAUs is that the VLAU only requires 50% of the memory blocks.

The FPGA-based coprocessor for gene finding by Chrysos et al. [21] represents an example of an architecture that could potentially benefit from the memory-efficient VLAU implementation. The Glimmer architecture is memory intensive and the attained level of parallelism was limited by the number of available embedded memory blocks in the device (personal communication with G. Chrysos; June 14th, 2010). The deployment of VLAUs can thus help to reduce the number of memory blocks required for computing the logarithm and thereby increase the degree of parallelism in the Glimmer architecture.

6. Experimental Results

Initially, we verified the functionality of the LAU/VLAU architectures (Section 6.1). In the following two sections, we

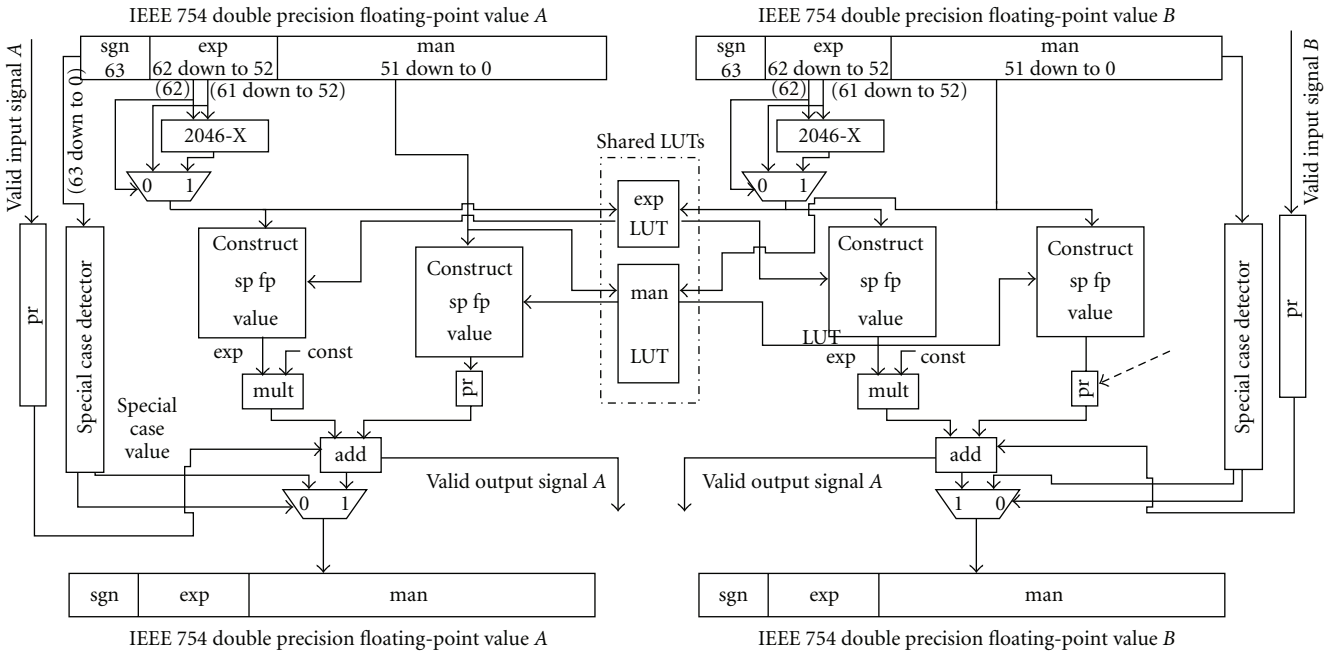

Figure 4: Top-level design of the VLAU architecture.

investigate the behavior of RAxML [8] using DP-ICSILog (Section 6.2) and assess the accuracy of the implementation (Section 6.3). Thereafter, Section 6.4 provides a detailed resource usage and performance evaluation for LAUs with various latencies and also for the VLAU architecture. We also compare performance and resource utilization with the FloPoCo logarithm [12]. A thorough run time comparison between the LAU/VLAU architectures and respective software implementations (GNU and MKL [23]) is presented in Section 6.5. Note that all results in Section 6 refer to Xilinx reports as obtained after the implementation process (post-place and route).

6.1. Verification. In order to verify the correctness of the proposed architectures, we conducted extensive post-place and route simulations as well as tests on an actual FPGA. As a 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-1 FPGA.

Initially, the advanced verification tool Chipscope Pro Analyzer was used to monitor the output ports of the SP/DP-LAUs and SP/DP-VLAUs, and the expected signals for given input numbers were tracked. Thereafter, an experimental PC-FPGA platform was set up. We use Gigabit Ethernet to communicate between the PC and the FPGA board based on the optimized unit for direct UDP/IP-based PC-FPGA communication that we recently made available [24]. A JAVA test application was used on the PC side to generate random SP and DP input values (using the standard `java.util.Random` class), organize the numbers into bytes, and transmit them to the FPGA. On the FPGA side, the floating-point representations were reconstructed from the incoming bytes and forwarded to the LAU/VLAU components. The logarithms of the inputs were then sent back to the JAVA test application on the PC from the FPGA and printed to screen.

Table 1: Log-likelihood score deviation with DP-ICSILog.

Dataset	DP-GNU	DP-ICSILog
44 organisms	−11231.35	−11231.29
90 organisms	−54078.01	−54078.18
150 organisms	−39606.31	−39606.60
218 organisms	−134173.86	−134167.56
140 organisms	−124777.22	−124780.10

6.2. DP-ICSILog in a Real-World Application. We integrated DP-ICSILog into RAxML [8], 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 of alternative tree topologies are computed. We found that an LUT with 4,096 entries is sufficient to guarantee numerical stability of RAxML and yield accurate results (see below). Table 1 indicates the respective log likelihood scores for tree searches using the GNU and DP-ICSILog implementations on various DNA datasets with 40, 90, 150, and 218 organisms (sequences) as well as a protein dataset with 140 organisms. Based on standard statistical significance tests for comparing log likelihood scores of phylogenetic trees as implemented in the CONSEL tool suite [25], we found that the score differences among the respective trees are not statistically significant. In other words, the trees computed (under the same starting conditions) using the

GNU implementation and the DP-ICSILog (LUT size: 4,096) cannot be statistically distinguished from each other. Hence, DP-ICSILog with an LUT size of 4,096 provides sufficient application- and domain-specific accuracy for RAxML.

6.3. Accuracy Assessment. Initially, we used the standard C `rand()` function to generate benchmarks with $2 * 10^7$ random numbers in order to measure the average error introduced by the logarithm approximation as a function of LUT size with respect to the GNU function. 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 an LUT with 4,096 entries represents a good tradeoff between accuracy and LUT size for our purposes (developing a hardware architecture for RAxML), since an LUT of this size only requires 3 block rams (36 Kb 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 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, that is, before modifying the reconfigurable architecture. For instance, the overall RAxML hardware architecture requires a huge amount of memory and reconfigurable fabric for other purposes. Therefore, we chose to minimize the hardware resources consumed for the logarithmic function to the largest possible extent.

Finally, 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, the respective logarithmic approximation implementations (SP-/DP-ICSILog, DP-LAU), and the SP-/DP-MKL library functions. Table 3 provides these errors for 10^6 random input numbers ranging from 10^{-20} to 10^{20}.

6.4. Performance Assessment versus Hardware. The LAUs and VLAUs were mapped to a Xilinx Virtex 5 SX95T-2 FPGA. In Figure 5, we provide resource usage and performance data for LAU (SP on the left and DP on the right) implementations with different latencies. We tested different latency-specific configuration settings for the Xilinx Floating-Point adders and multipliers that are generated. The variation of these settings allowed us to generate LAU implementations with latencies that range between 5 and 22 clock cycles. Note that, all measurements in this Section refer to LAUs and VLAUs with a *man_LUT* size of 4,096 entries.

The respective clock frequencies of the LAUs were obtained using the Xilinx Tools (ADVANCED 1.53 speed file) and are also provided in Figure 5. The clock frequencies are obtained from the static timing report, and the default Xilinx `Balanced` optimization strategy was selected. All implementations (SP-LAU, DP-LAU, SP-VLAU, and DP-VLAU) are fully pipelined with a throughput of one result per clock cycle and per pipeline datapath. Since the LAU only comprises a single pipeline datapath, a throughput of one result per clock cycle is achieved, while the VLAU (with two independent pipeline datapaths) can compute two results per clock cycle.

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

# Block rams (18 Kb)	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 3: Min, max, average, and mean squared error of logarithm implementations with respect to GNU functions.

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

In Figure 6, we provide the clock frequencies of the SP-LAU and DP-LAU for *man_LUT* sizes ranging between 512 and 32,768 entries. The frequency reduction with increasing LUT size is due to the additional logic (mostly block rams for the LUT) that is required by the LAU. The number of block rams required increases exponentially for every bit that is added to the mantissa field, which is used as an index for *man_LUT*. The increase of other reconfigurable resources is significantly lower, that is, a LAU with a 32,768 entry *man_LUT* size occupies 700% more 36 Kb 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.

In Table 4, we compare the hardware resources used by our custom LUT-based module and the Xilinx FPO [10] (configured in fixed-to-float mode) for transforming the exponent value into a floating point value. The numbers in parentheses next to the names in the first line of Table 4 represent the latency (number of clock cycles) for alternative configurations. 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 60 MHz lower than for our LUT-based approach. The LUT-based module occupies 1 BRAM (18 Kb) while the FPO solution does not use BRAM memory. For some applications, trading some memory for a substantially higher clock speed is acceptable, since it can yield a higher overall clock frequency and thereby improved overall system performance. When the FPO is configured with the maximum latency of 6 cycles, the LAU is only 5 MHz

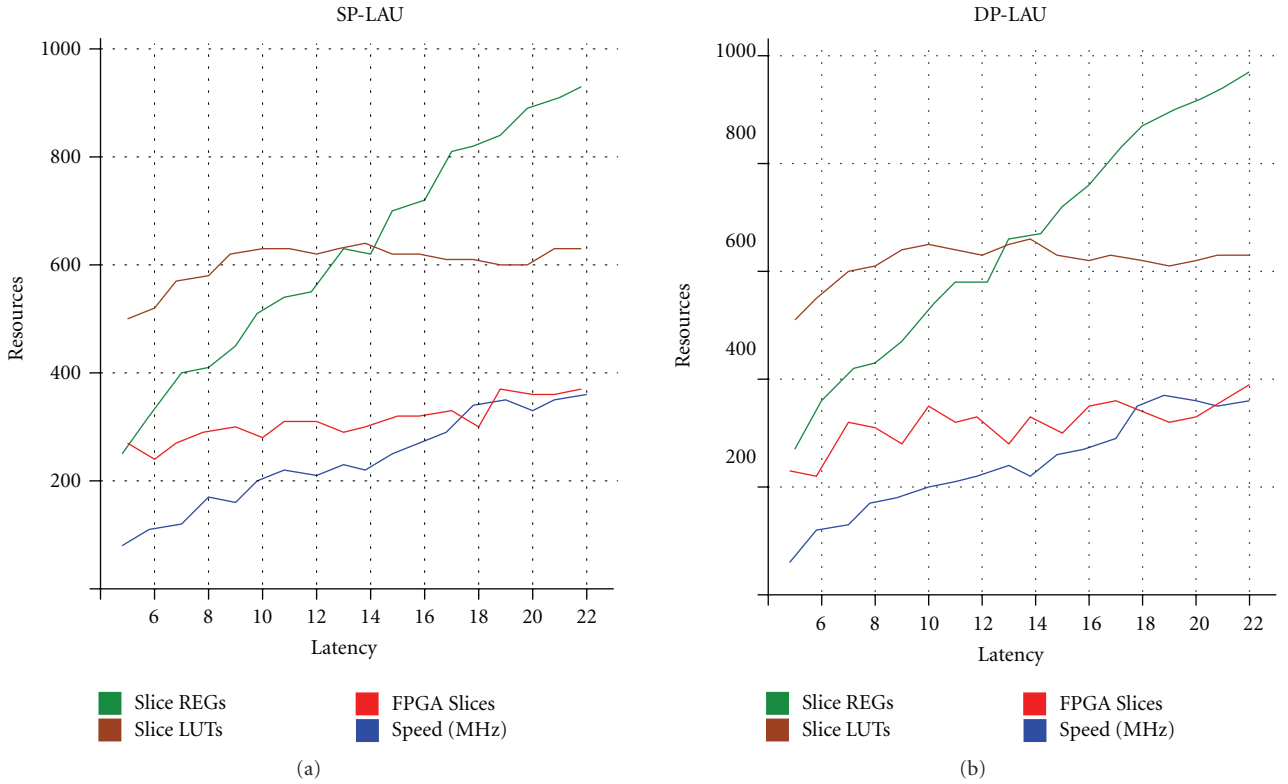

Figure 5: Resources and performance of alternative SP and DP LAU implementations.

Table 4: Resource usage by the LUT-based approach (latency of two cycles) and Xilinx FPO (latency of two and six cycles) for transformation of exponent to SP number.

Resources	LUT(2)	FPO(2)	FPO(6)
Slice registers	32	46	115
Slice LUTs	19	64	88
Occupied slices	20	19	42
# of LUT-flip flop pairs	48	45	126
# of BRAMS (18 Kb)	1	0	0
DP-LAU frequency (MHz)	334	274	339
DP-LAU latency (cycles)	22	22	26

faster than with our LUT-based approach. However, the total latency of the LAU increases by 4 cycles, and the FPO requires a larger amount (see Table 4) of hardware resources.

To the best of our knowledge, the only other open-source logarithm for FPGAs is provided by FloPoCo [12] framework. All FloPoCo operators can be fully parameterized, that is, the user can select the desired precision of the result and define desired performance parameters. To conduct a fair comparison between LAU/VLAU and FPLog units, we used the latest release of FloPoCo (version 2.0.0) and mapped the LAUs/VLAUs and FPLogs to the same FPGA (Virtex 5 SX95T-2). Table 5 provides a performance and resource usage comparison after the implementation process (post-place and route). The reduced FPLog implementations (denoted as (*red.*) in the table) offer the same accuracy as the LAU/VLAU implementations, while the full (precise) FPLog implementations (denoted as (*full*)) yield the same results as the GNU function.

In addition, all available Xilinx optimization strategies were explored to determine the most efficient strategy for each implementation. The available optimization strategies for Virtex 5 devices are `Balanced`, `Area Reduction`, `Minimum`, `Runtime`, `Power Optimization`, and `Timing Performance`. For each implementation/architecture, we only provide the data for the best optimization strategy with respect to clock frequencies in Table 5. The SP-LAU occupies slightly less hardware than the full (high precision) SP-FPLog, while DP-LAU requires significantly less resources than the full DP-FPLog. When the SP/DP FPLogs are configured to yield the same accuracy as the SP/DP LAUs, the FPLog implementations are more resource efficient but exhibit significantly lower maximum clock frequencies. Thus, reduced-precision FPLog units are more likely to lie on the critical path, when embedded into a larger, more complex architecture that needs to calculate logarithms. The VLAUs outperform all other implementations in terms of throughput, since they can produce 2 results per cycle. To achieve performance that is comparable to the VLAU with the FPLog(red.) implementation, two FPLog(red.) instances are required. Therefore, a single VLAU is more resource efficient than two FPLog(red.) units, since the total number of occupied slices and the number of slice LUTs used by the VLAU is smaller than the total amount required for two FPLog(red.) instances. At the same time, the resource

Table 5: Resources, performance, and accuracy of LAUs, VLAUs, and FPLogs.

Resources total	SP				DP			
	LAU	VLAU	FPLog (red.)	FPLog (full)	LAU	VLAU	FPLog (red.)	FPLog (full)
Slice registers 58,800	932	1,864	782	992	970	1,912	809	2,568
Slice LUTs-58,800	551	1,099	712	873	634	1,107	735	1,910
Occupied slices-14,720	298	569	294	330	341	576	307	711
# 36 k block RAM-244	3	3	1	2	3	3	1	2
# 18 k block RAM-488	1	1	1	2	1	1	1	21
# DSP48Es-640	3	6	3	5	3	6	3	14
Frequency (MHz)	370	351	233	240	334	330	233	198
Latency	22	22	18	20	22	22	18	34
Results/cycle	1	2	1	1	1	2	1	1
Error	2^{-17}	2^{-17}	2^{-17}	2^{-23}	2^{-17}	2^{-17}	2^{-17}	2^{-52}

consumption for all other resources, that is, slice registers, brams, and DSPs, is the same. Nevertheless, the single VLAU still outperforms the two FPLog(red.) instances with respect to clock frequency.

As far as the LAU implementation by Chrysos et al. [21] is concerned, two LUTs are deployed, but the LUTs are not initialized as efficiently as in our LAU. Consequently, additional operations, that is, a concatenation, a floating-point multiplication, a float-to-fixed operation, and a fixed-point subtraction, are required for calculating the LUT index. The respective LUT entry is then used to calculate the final output which also represents an approximation of the logarithm function. Since the paper by Chrysos et al. [21] focuses on the overall architecture for Glimmer, only a limited amount of information is provided with respect to implementation and performance of the logarithm unit.

Finally, the configurations presented in [17] by Tropea were mapped to a Virtex 4 LX15-12 FPGA by Tropea. The highest clock frequency reported in [17] is 191 MHz (the architecture has not been made available by the author). Thus, to conduct a fair comparison, we also mapped the LAU to a Virtex4 LX15-12 FPGA. We obtained a clock frequency of 345 MHz for the SP version and of 344 MHz for the DP version.

6.5. Performance Assessment versus Software. We also compared LAU and VLAU performance 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 arithmetic component, that is, only one LAU and only one VLAU were instantiated, respectively. The software implementations were executed on an Intel Core2 Duo T9600 processor running at 2.8 GHz with 6 MB of L2 Cache. All software (SP-/DP-ICSILog) and hardware implementations (SP-/DP-LAU) we tested used a mantissa LUT with 4,096 entries.

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

# samples	GNU	ICSILog	LAU	VLAU
10^3	0.03290	0.00620	0.0027	0.0015
10^6	32.40	6.31	2.7	1.42
10^8	3315	595	270.2	142.45

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 aggressive 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 6 shows the execution

Table 7: Execution times (in ms) of GNU, ICSILog, LAU, and VLAU DP implementations for 10^3 to 10^8 invocations.

# samples	GNU	ICSILog	LAU	VLAU
10^3	0.0722	0.0119	0.003	0.0016
10^6	58.40	9.47	2.99	1.51
10^8	5909	899	299.4	151.5

Table 8: Execution times (in ms) of MKL, ICSILog, LAU, and VLAU SP implementations (The `icc` compiler is used.)

# samples	MKL	ICSILog	LAU	VLAU
10^6	4.7	5.3	2.7	1.42
10^7	46.9	50.2	27.0	14.25
10^8	342.9	486.6	270.2	142.45

Table 9: Execution times (in ms) of MKL, ICSILog, LAU, and VLAU DP implementations (The `icc` compiler is used.)

# samples	MKL	ICSILog	LAU	VLAU
10^6	8.0	8.8	2.99	1.51
10^7	77.2	85.1	29.94	15.15
10^8	668.4	839.7	299.4	151.5

times for the GNU implementation, SP-ICSILog, the SP-LAU, and the SP-VLAU. The SP LAU is 12 times faster than the GNU function and 2.2 times faster than SP-ICSILog, while the SP-VLAU is 23 times faster than the GNU functions and 4.1 times faster than SP-ICSILog.

As already mentioned, the standard release of ICSILog only provides an SP logarithm function. 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 reimplemented 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/VLAU architectures.

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, DP-LAU, and DP-VLAU logarithm functions (Table 7). The DP-LAU is 20 times faster than the GNU math library and 3.1 times faster than DP-ICSILog which in turn is up to 6.5 times faster than the GNU implementation. The DP-VLAU is 40 times faster than the GNU math library implementation and 6 times faster than DP-ICSILog.

For our second set of experiments, we used the Intel `icc` compiler (version 11.1, optimization flag `−O1`). We also tested the fast logarithm implementation provided by the Intel Math Kernel Library (MKL [23]) for 106 to 10^8 invocations on random input numbers as in the preceding experiments.

Tables 8 and 9 provide the execution times for the SP and DP MKL, ICSILog, and LAU implementations, respectively.

Table 10: Execution times (in ms) of GNU and ICSILog DP implementations compiled with `gcc` and `−O2`/`−O3`.

# 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 11: Execution times (in ms) of MKL and ICSILog DP implementations compiled with `icc` 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

The SP-LAU is 1.7 times faster than the MKL logarithm and 1.8 times faster than SP-ICSILog, while the respective speedups for the SP-VLAU are 3.3 and 3.5. Unfortunately, a detailed description of the MKL logarithm implementation is currently not available. The DP-LAU is 2.6 times faster than the respective MKL implementation and 2.8 times faster than DP-ICSILog which is almost as fast as the DP-MKL function (speedups vary between 0.8 and 0.9). The DP-VLAU is 5.2 times faster than the MKL implementation and 5.6 times faster than DP-ICSILog.

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 times with all alternative DP implementations: DP-GNU, DP-MKL, and DP-LAU. Table 10 provides the execution times for DP-GNU and DP-ICSILog for 10^3 to 10^8 invocations of the `gcc`-compiled code. The DP-LAU is 19 times faster than the GNU math library and 2.7 times faster than DP-ICSILog, which in turn is up to 7 times faster than the GNU implementation (for `−O2` as well as `−O3`). The DP-VLAU is 37.5 times faster than the GNU math library and 5.3 times faster than DP-ICSILog. Table 11 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, the DP-VLAU is 4 times faster than the MKL function and 5 times faster than DP-ICSILog.

7. Conclusion and Future Work

We presented an architecture that efficiently calculates an approximation of the logarithm in reconfigurable logic under SP and DP arithmetics and only uses 2% of the computational resources on medium-size FPGAs. The SP-/DP-LAUs (LUT size: 4,096) as well as the DP software are freely available for download. To the best of our knowledge,

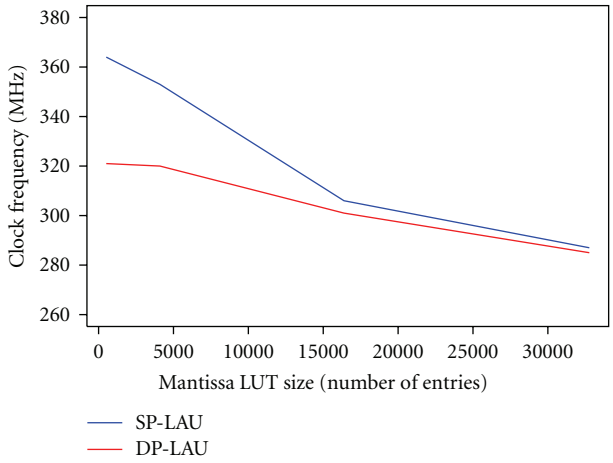

Figure 6: LAU frequencies with respect to LUT size.

this represents the only IEEE-754 compatible open-source 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 6.4), if the LUT size is increased and the speed will only slightly decrease (see Figure 6).

Finally, we designed a memory-efficient VLAU architecture that exploits the dual-port option of embedded memory blocks. The VLAU utilizes two pipelined LAU datapaths but only requires one instance of the read-only lookup tables. This feature allows a VLAU to calculate two results per cycle while requiring half the LUT memory than two independent parallel LAUs. The VLAU can therefore be used for designing large architectures that require the computation of logarithms on vectors. The SP/DP-VLAUs are freely available for download.

Acknowledgment

Part of this work was funded under the auspices of the Emmy-Noether program by the German Science Foundation (DFG).

References

[1] D. Ververidis and C. Kotropoulos, "Gaussian mixture modeling by exploiting the Mahalanobis distance," *IEEE Transactions on Signal Processing*, vol. 56, no. 7 I, pp. 2797–2811, 2008.

[2] J. Felsenstein, "Evolutionary trees from DNA sequences: a maximum likelihood approach," *Journal of Molecular Evolution*, vol. 17, no. 6, 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 the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, pp. 1–8, Rome, Italy, May 2009.

[4] N. Alachiotis, A. Stamatakis, E. Sotiriades, and A. Dollas, "A reconfigurable architecture for the phylogenetic likelihood function," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 674–678, Prague, Czech Republic, September 2009.

[5] V. M. Preciado, *Real-Time Wavelent Transform for Image Processing on the Cellular Neural Network Universal Machine*, vol. 2085 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, 2001.

[6] B. De Ruijsscher, G. N. Gaydadjiev, J. Lichtenauer, and E. Hendriks, "FPGA accelerator for real-time skin segmentation," in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia (ESTIMEDIA '06)*, pp. 93–97, October 2006.

[7] N. Alachiotis and A. Stamatakis, "Efficient floating-point logarithm unit for FPGAS," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, Atlanta, Ga, USA, April 2010.

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

[9] O. Vinyals and G. Friedland, "A hardware-independent fast logarithm approximation with adjustable accuracy," in *Proceedings of the 10th IEEE International Symposium on Multimedia (ISM '08)*, pp. 61–65, December 2008.

[10] Xilinx, "Floating Point Operator v4.0," July 2009, http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf.

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

[12] F. De Dinechin, C. Klein, and B. Pasca, "Generating high-performance custom floating-point pipelines," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications(FPL '09)*, pp. 59–64, September 2009.

[13] J. Detrey and F. de Dinechin, "Parameterized floating-point logarithmand exponential functions for FPGAs," *Microprocessors and Microsystems, Special Issue on FPGA-Based Reconfigurable Computing*, vol. 31, no. 8, pp. 537–545, 2007.

[14] J. Detrey and F. De Dinechin, "A parameterizable floating-point logarithm operator for FPGAs," in *Proceedings of the 39th Asilomar Conference on Signals, Systems and Computers*, vol. 2005, pp. 1186–1190, IEEE Signal Processing Society, November 2005.

[15] National Instruments, "High Throughput Natural Logarithm Function," http://www.ni.com/.

[16] J. E. Volder, "The CORDIC trigonometric computing technique," in *Proceedings of IRE Transactions on Electronic Computers*, pp. 330–334, 1959.

[17] S. E. Tropea, "FPGA implementation of base-N logarithm," in *Proceedings of the 3rd Southern Conference on Programmable Logic (SPL '07)*, pp. 27–32, February 2007.

[18] Xilinx, http://www.xilinx.com/.

[19] Actel, January 2009, http://www.actel.com/.

[20] B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, 2000.

[21] G. Chrysos, E. Sotiriades, I. Papaefstathiou, and A. Dollas, "A FPGA based coprocessor for gene finding using Interpolated Markov Model (IMM)," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 683–686, August 2009.

[22] J. J. Raygoza-Panduro, S. Ortega-Cisneros, J. Rivera, and A. de la Mora, "Design of a mathematical unit in FPGA for

the implementation of the control of a magnetic levitation system," *International Journal of Reconfigurable Computing*, vol. 2008, Article ID 634306, p. 9, 2008.

[23] Intel, "Intel Math Kernel LibraryReference Manual," http://software.intel.com/en-us/articles/intel-mkl/.

[24] N. Alachiotis, S. A. Berger, and A. Stamatakis, "Efficient PCF-PGA communication over Gigabit Ethernet," in *Proceedings of the International Conferences on Embedded Software and Systems (ICESS '10)*, pp. 1727–1734, Bradford, UK, 2010.

[25] H. Shimodaira and M. Hasegawa, "CONSEL: for assessing the confidence of phylogenetic tree selection," *Bioinformatics*, vol. 17, no. 12, pp. 1246–1247, 2002.