Automatic generation of polynomial-based hardware architectures for function evaluation

Florent de Dinechin, Mioara Joldes, Bogdan Pasca
LIP (CNRS/INRIA/ENS-Lyon/UCBL)
Université de Lyon
{*Florent.de.Dinechin, Mioara.Joldes, Bogdan.Pasca*}*@ens-lyon.fr*

***Abstract*—Polynomial approximation is a very general technique for the evaluation of a wide class of numerical functions of one variable. This article details an architecture generator that inputs the specification of a function and outputs a synthesizable description of an architecture evaluating this function with guaranteed accuracy. It improves upon the literature in two aspects. Firstly, it uses better polynomials, thanks to recent advances related to constrained-coefficient polynomial approximation. Secondly, it refines the error analysis of polynomial evaluation to reduce the size of the multipliers used. An open-source implementation is provided in the FloPoCo project, including architecture exploration heuristics designed to use efficiently the embedded memories and multipliers of high-end FPGAs. High-performance pipelined architectures for precisions up to 64 bits can be obtained in seconds.**

***Keywords*-elementary function; hardware evaluator; polynomial approximation; FPGA;**

I. Introduction and motivation

In this article, we consider real functions $f(x)$ of one real variable x, and we are interested in a fixed-point implementation of this function over some interval. We assume that f is continuously differentiable over this interval up to a certain order. The literature provides many examples of such functions for which a hardware implementation is required.

- Fixed-point sine, cosine, exponential and logarithms are routinely used in signal processing algorithms.
- Random number generators with a Gaussian distribution may be built using the Box-Muller method, which requires logarithm, square root, sine and cosine [1]. Arbitrary distributions may be obtained by the inversion method, in which case one needs a fixed-point evaluator for the inverse cumulative distribution function (ICDF) of the required distribution [2]. There are as many ICDF as there are statistical distributions.
- Approximations of the inverse $1/x$ and inverse square root $1/\sqrt{x}$ functions are used in recent floating-point units to bootstrap division and square root computation [3].
- $f_{\log}(x) = \log(x+1/2)/(x-1/2)$ over $[0,1]$, and $f_{\exp}(x) = e^x - 1 - x$ over $[0, 2^{-k}]$ for some small k, are used to build hardware floating-point logarithm and exponential in [4].
- $f_{\cos}(x) = 1 - \cos\left(\frac{\pi}{4}x\right)$, and $f_{\sin}(x) = \frac{\pi}{4} - \frac{\sin\left(\frac{\pi}{4}x\right)}{x}$ over $[0,1]$, are used to build hardware floating-point trigonometric functions in [5].
- $s_2(x) = \log_2(1+2^x)$ and $d_2(x) = \log_2(1+2^x)$ are used to build adders and subtracters in the Logarithm Number System (LNS), and many more functions are needed for Complex LNS [6].

Many function-specific algorithms exist, for example variations on the CORDIC algorithm provide low-area, long-latency evaluation of most elementary functions [7]. Our purpose here is to provide a generic method, that is a method that works for a very large class of functions. The main motivation of this work is to facilitate the implementation of a full hardware mathematical library (libm) in FloPoCo, a core generator for high-performance computing on FPGAs[1]. We present a complete implementation in this context, however, most of the methodology is independent of the FPGA target and could apply to other hardware targets such as ASIC circuits.

A. Related work and contributions

Articles describing specific polynomial evaluators are too numerous to be mentioned here, and we just review works that describe generic methods.

Several table-based, multiplier-less methods for linear (or degree-1) approximation have evolved from the original paper by Sunderland et al [8]. See [9] or [7] for a review. These methods have very low latency but do not scale well beyond 20 bits: the table sizes scale exponentially, and so does the design-space exploration time.

The High-Order Table-Based Method (HOTBM) by Detrey and one of us [10] extended the previous methods to higher-degree polynomial approximation. An open-source implementation is available in FloPoCo. However it is not suited to recent FPGAs with powerful DSP blocks and large embedded memories. In addition, it doesn't scale beyond 32 bits.

Lee et al [11] have published many variations on a generic datapath optimization tool called MiniBit to optimize polynomial approximation. They use ad-hoc mixes of analytical

[1] www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/

techniques such as interval analysis, and heuristics such as simulated annealing to explore the design space. However, the design space explored in these articles does not include the architectures we describe in the present paper: All the multipliers in these papers are larger than strictly needed, therefore they miss the optimal. In addition, this tool is closed-source and difficult to evaluate from the publications, in particular it is unclear if it scales beyond 32 bits.

Tisserand studied the optimization of low-precision (less than 10 bits) polynomial evaluators [12]. He finetunes a rounded minimax approximation using an exhaustive exploration of neighboring polynomials. He also uses other tricks on smaller (5-bit or less) coefficients to replace the multiplication by such a coefficient by very few additions. Such tricks do not scale to larger precisions.

Compared to these publications, the present work has the following distinctive features.

- This approach scales to precisions of 64 bits or more, while being equivalent or better than the previous approaches for smaller precisions.
- We use for polynomial approximation minimax polynomials provided by the Sollya tool[2], which is the state-of-the-art for this application, as detailed in Section II-B.
- We attempt to use the smallest possible multipliers. As others, we attempt to minimize the coefficient sizes. In addition, we also truncate, at each computation step, the input argument to the bare minimum of bits that are needed at this step. Besides, we will use truncated multipliers [19], [14] in the near future.
- This approach is fully automated, from the parsing of an expression describing the function to VHDL generation. An open-source implementation is available as the FunctionEvaluator class in FloPoCo, starting with version 2.0.0. This implementation is fully operational, to the point that Table II was obtained in less one hour.
- The resulting architecture evaluates the function with last-bit accuracy. It may be automatically pipelined to a user-specified frequency thanks FloPoCo's pipelining framework [13].

B. Relevant features of recent FPGAs

Here are some of the features of recent FPGAs that can be used in polynomial evaluators.

- Embedded multipliers features are summed up in Table. I It is possible to build larger multipliers by assembling these embedded multipliers. The DSP blocks include specific adders and shifters designed for this purpose [14].
- Memories have a capacity of 9Kbit or 144Kbit (Altera) or 18Kbit (Xilinx) and can be configured in shape, for instance from $2^{16} \times 1$ to $2^9 \times 36$ for the Virtex-4.

[2]http://sollya.gforge.inria.fr/

Family	*Multipliers*
Virtex II to Virtex-4	18x18 signed or 17x17 unsigned
Virtex-5/Virtex-6	18x25 signed or 17x24 unsigned
Stratix II/III/IV	18x18 signed or unsigned

Table I
MULTIPLIER BLOCKS IN RECENT FPGAS

A given FPGA typically contains a comparable number of memory blocks and multipliers. It therefore makes sense to try and balance the consumption of these two resources. However, the availability of these resources also depends on the wider context of the application, and it is even better to expose a range of trade-offs between them.

II. FUNCTION EVALUATION BY POLYNOMIAL APPROXIMATION

Polynomial approximation is the generic mathematical tool that reduces the evaluation of a function to additions and multiplications. For these operations, we can either build architectures (in FPGAs or ASICs), or use built-in operators (in processors or DSP-enabled FPGAs). A good primer on polynomial approximation for function evaluation is Muller's book [7].

Building a polynomial evaluator for a function may be decomposed into two subproblems: 1/ *approximation*: finding a good approximation polynomial, and 2/ *evaluation*: evaluating it using adders and multipliers. The smaller the input argument, the better these two steps will behave, therefore a *range reduction* may be applied first if the input interval is large.

We now discuss each of these steps in more detail, to build the implementation flow depicted on Figure 1. In this paper we will consider, without loss of generality, a function f over the input interval $x \in [0, 1)$.

In our implementation, the user inputs a function (assumed on $[0, 1)$, the input and output precisions (both expressed as LSB weight), and the degree d of the polynomials used. This last parameter could be determined heuristically, but we leave it as a means for the user to trade-off multipliers and latency for memory size.

A. Range reduction

In this work, we use the simple range reduction that consists in splitting the input interval in 2^k sub-intervals, indexed by $i \in \{0, 1, ..., 2^k - 1\}$. The index i may be obtained as the leading bits of the binary representation of the input: $x = 2^{-k}i + y$ with $y \in [0, 2^{-k})$. This decomposition comes at no hardware cost. We now have $\forall i \in \{0, \ldots, 2^k - 1\} \quad f(x) = f_i(y)$, and we may approximate each f_i by a polynomial p_i. A table will hold the coefficients of all these polynomials, and the evaluation of each polynomial will share the same hardware (adders

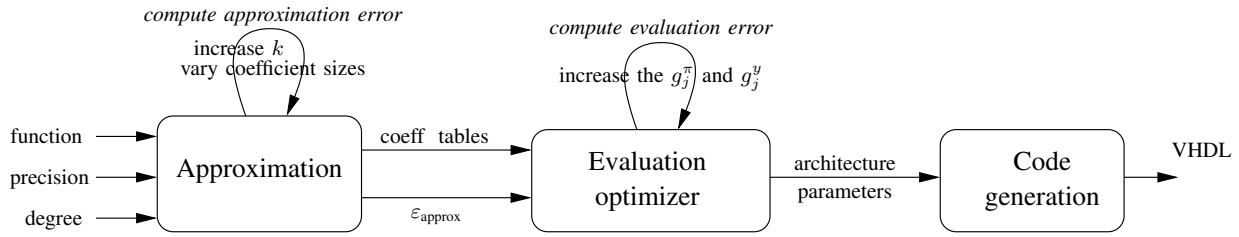

Figure 1. Automated implementation flow

and multipliers), which therefore have to be built to accommodate the worst-case among these polynomial. Figure 3 describes the resulting architecture.

Compared to a single polynomial on the interval, this range reduction increases the storage space required, but decreases the cost of the evaluation hardware for two reasons. First, for a given target accuracy $\varepsilon_{\text{total}}$, the degree of each of the p_i decreases with k. There is a strong threshold effect here, and for a given degree there is a minimal k that allows to achieve the accuracy. Second, the reduced argument y has k bits less than the input argument x, which will reduce the input size of the corresponding multipliers. If we target an FPGA with DSP blocks, there will also be a threshold effect here on the number of DSP blocks used.

Many other range reductions are possible, most related to a given function or class of functions, like the logarithmic segmentation used in [2]. For an overview, see Muller [7]. Most of our contributions are independent of the range reduction used.

B. Polynomial approximation

One may use the well-known Taylor or Chebyshev approximation polynomials of arbitrary degree d [7]. These polynomials can be obtained analytically, or using computer algebra systems. A third method of polynomial approximation is Remez' algorithm, a numerical process that, under some conditions, converges to the minimax approximation: the polynomial of degree d that minimizes the maximal difference between the polynomial and the function. We denote $\varepsilon_{\text{approx}}$ the approximation error, defined as the maximum absolute difference between the polynomial and the function.

Between approximation and evaluation, for an efficient machine implementation, one has to round the coefficients of the minimax polynomial (which has real numbers in theory, and are computed with large precision in practice) to smaller-precision numbers suitable for efficient evaluation. On a processor, one will typically try to round to single- or double-precision numbers. On an FPGA, we may build adders and multipliers of arbitrary size, so we have one more question to answer: what is the optimal size of these coefficients? In [11], this question is answered by an error analysis that considers separately the error of rounding each coefficient of the minimax polynomial (considered as a real-coefficient one) and tries to minimize the bit-width of the rounded coefficients while remaining within acceptable error bounds.

However, there is no guarantee that the polynomial obtained by rounding the coefficients of the real minimax polynomial is the minimax among the polynomials with coefficients constrained to these bit-width. Indeed, this assumption is generally wrong. One may obtain much more accurate polynomials for the same coefficient bit-width using a modified Remez algorithm due to Brisebarre and Chevillard [15] and implemented as the `fpminimax` command of the Sollya tool. This command inputs a function, an interval and a list of constraints on the coefficient (e.g. constraints on bitwidths), and returns a polynomial that is very close to the best minimax approximation polynomial among those with such constrained coefficients.

Since the approximation polynomial now has constrained coefficients, we will not round these coefficients anymore. In other words, we have merged the approximation error and the coefficient truncation error of [11] into a single error, which we still denote $\varepsilon_{\text{approx}}$. The only remaining rounding or truncation errors to consider are those that happen during the evaluation of the polynomial.

Let us now provide a good heuristic for determining the coefficient constraints. Let $p(y) = a_0 + a_1 y + a_2 y^2 + ... + a_d y^d$ be the polynomial on one of the sub-intervals (for clarity, we remove the indices corresponding to the sub-interval). The constraints taken by `fpminimax` are the minimal weights

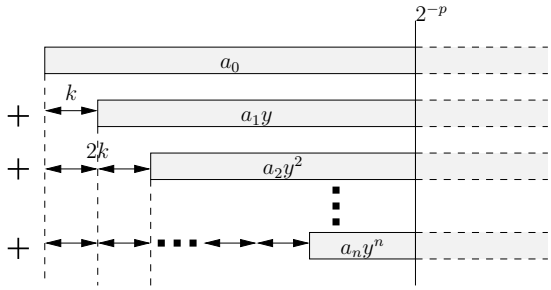

Figure 2. Alignment of the monomials

of the least significant bit (LSB) of each coefficient. To reach some target precision 2^{-p}, we need the LSB of a_0 to be of weight at most 2^{-p}. This provides the constraint on a_0. Now consider the developed form of the polynomial, as illustrated by Figure 2. As coefficient a_j is multiplied by y^j which is smaller than 2^{-kj}, the accuracy of the monomial $a_j y^j$ will be aligned on that of the monomial a_0 if its LSB is of weight 2^{-p+kj}. This provides a constraint on a_j.

The heuristic used is therefore the following. Remember that the degree d is provided by the user. The constraints on the $d+1$ coefficients are set as just explained. For increasing k, we try to find 2^k approximation polynomials p_i of degree d respecting the constraints, and fulfilling the target approximation error (which will be defined in Section II-D). We stop at the first k that succeeds. Then, the 2^k polynomials are scanned, and the maximum magnitude of all the coefficients of degree j provides the most significant bit that must be tabulated, hence the memory consumed by this coefficient.

C. Polynomial evaluation

Given a polynomial, there are many possible ways to evaluate it. The HOTBM method [10] uses the developed form $p(y) = a_0 + a_1 y + a_2 y^2 + ... + a_d y^d$ and attempts to tabulate as much of the computation as possible. This leads to short-latency architecture since each of the $a_i y^i$ may be evaluated in parallel and added thanks to an adder tree, but at a high hardware cost.

In this article, we chose a more classical Horner evaluation scheme, which minimizes the number of operations, at the expense of the latency: $p(y) = a_0 + y \times (a_1 + y \times (a_2 + ... + y \times a_d)...)$. Our contribution is essentially a fine error analysis that allows us to minimize the size of each of the operations. It is presented below in II-D.

There are intermediate schemes that could be explored. For large degrees, the polynomial may be decomposed into an odd and an even part: $p(y) = p_e(y^2) + y \times p_o(y^2)$. The two sub-polynomial may be evaluated in parallel, so this scheme has a shorter latency than Horner, at the expense of the precomputation of x^2 and a slightly degraded accuracy. Many variations on this idea, e.g. the Estrin scheme, exist [7], and this should be the subject of future work. A polynomial may also be refactored to trade multiplications for more additions [16], but this idea is mostly incompatible with range reduction.

D. Accuracy and error analysis

The maximal error target $\varepsilon_{\text{total}}$ is an input to the algorithm. Typically, we aim at *faithful rounding*, which means that $\varepsilon_{\text{total}}$ must be smaller than the weight of the LSB of the result, noted u. In other words, all the bits returned hold useful information. This error is decomposed as follows: $\varepsilon_{\text{total}} = \varepsilon_{\text{approx}} + \varepsilon_{\text{eval}} + \varepsilon_{\text{finalround}}$ where

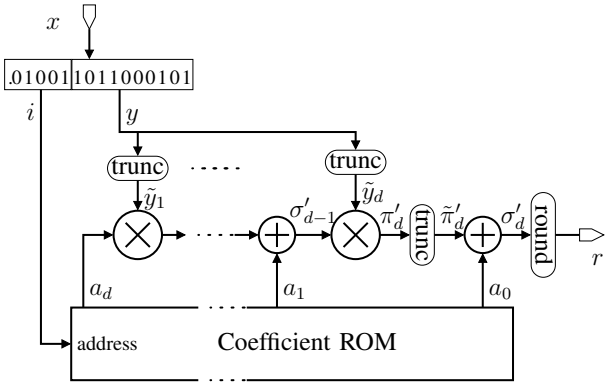

Figure 3. The function evaluation architecture

- $\varepsilon_{\text{approx}}$ is the approximation error, the maximum absolute difference between any of the p_i and the corresponding f_i over their respective intervals. This computation belongs to the approximation step and is also performed using Sollya [17].
- $\varepsilon_{\text{eval}}$ is the total of all rounding errors during the evaluation;
- $\varepsilon_{\text{finalround}}$ is the error corresponding to the final rounding of the evaluated polynomial to the target format. It is bounded by $u/2$.

We therefore need to ensure $\varepsilon_{\text{approx}} + \varepsilon_{\text{eval}} < u/2$. The polynomial approximation algorithm iterates until $\varepsilon_{\text{approx}} < u/4$, then reports $\varepsilon_{\text{approx}}$. The error budget that remains for the evaluation is therefore $\varepsilon_{\text{eval}} < u/2 - \varepsilon_{\text{approx}}$ and is between $u/4$ and $u/2$.

In $p(y) = a_0 + a_1 y + a_2 y^2 + ... + a_d y^d$, the input y is considered exact, so $p(y)$ is the value of the polynomial if evaluated in infinite precision. What the architecture evaluates is $p'(y)$, and our purpose here is to compute a bound on $\varepsilon_{\text{eval}}(y) = p'(y) - p(y)$.

Let us decompose the Horner evaluation of p as a recurrence:

$$\begin{cases} \sigma_0 = a_d \\ \pi_j = y \times \sigma_{j-1} & \forall j \in \{1...d\} \\ \sigma_j = a_{d-j} + \pi_j & \forall j \in \{1...d\} \\ p(y) = \sigma_d \end{cases}$$

This would compute the exact value of the polynomial, but at each evaluation step, we may perform two truncations, one on y, and one on π_j. As a rule of thumb, each step should balance the effect of these two truncations on the final error. For instance, in an addition, if one of the addends is much more accurate than the other one, it probably means that it was computed too accurately, wasting resources.

To understand what is going on, consider step j. In the addition $\sigma_j = a_{d-j} + \pi_j$, the π_j should be at least as accurate as a_{d-j}, but not much more accurate: let us keep g_j^π bits to the right of the LSB of a_{d-j}, where g_j^π is a small positive integer ($0 \leq g_j^\pi < 5$ in our experiments). The parameter g_j^π

defines the truncation of π_j, and also the size of σ_j (which also depends on the weight of the MSB of a_{d-j}).

Now since we are going to truncate $\pi_j = y \times \sigma_{j-1}$, there is no need to input to this computation a fully accurate y. Instead, y should be truncated to the size of the truncated π_j, plus a small number g_j^y of guard bits.

The computation actually performed is therefore the following:

$$\begin{cases} \sigma'_0 = a_d \\ \pi'_j = \tilde{y}_j \times \sigma'_{j-1} & \forall j \in \{1...d\} \\ \sigma'_j = a_{d-j} + \tilde{\pi}'_j & \forall j \in \{1...d\} \\ p'(y) = \sigma'_d \end{cases}$$

In both previous equations, the additions and multiplications should be viewed as exact: the truncations are explicited by the tilded variables, e.g. $\tilde{\pi}'_j$ is the truncation of π'_j to g_j^π bits beyond the LSB of a_{d-j}. There is no need to truncate the result of the addition, as the truncation of π'_j serves this purpose already.

We may now compute the rounding error:

$$\varepsilon_{\text{eval}} = p'(y) - p(y) = \sigma'_d - \sigma_d$$

where

$$\begin{aligned} \sigma'_j - \sigma_j &= \tilde{\pi}'_j - \pi_j \\ &= (\tilde{\pi}'_j - \pi'_j) \ + \ (\pi'_j - \pi_j) \end{aligned}$$

Here we have a sum of two errors. The first, $\tilde{\pi}'_j - \pi'_j$, is the truncation error on π' and is bounded by a power of two depending on the parameter g_j^π. The second is computed as

$$\begin{aligned} \pi'_j - \pi_j &= \tilde{y}_j \times \sigma'_{j-1} - y \times \sigma_{j-1} \\ &= (\tilde{y}_j \sigma'_{j-1} - y\sigma'_{j-1}) \ + \ (y\sigma'_{j-1} - y\sigma_{j-1}) \\ &= (\tilde{y}_j - y)\sigma'_{j-1} \quad + \quad y \times (\sigma'_{j-1} - \sigma_{j-1}) \end{aligned}$$

Again, we have two error terms which we may bound separately. The first bound is the truncation error on y, which depends on the parameter g_j^y, and is multiplied by a bound on σ'_{j-1} which has to be computed recursively itself. The second term recursively uses the computation of $\sigma'_j - \sigma_j$, and the bound $y < 2^{-k}$.

The previous error computation is implemented in C++. From the values of the parameters g_j^π and g_j^y, it decides if the architecture defined by these parameters is accurate enough.

E. Parameter space exploration for the FPGA target

The last problem to solve is to find values of these parameters that minimize the cost of an implementation. This optimization problem is very dependent on the target technology, and we now present an exploration heuristic that is specific to DSP-enabled FPGAs: our objective will be to minimize the number of DSP blocks.

Let us first consider the g_j^y parameter. The size of this truncation directly influences the DSP count. Here, we observe that once a DSP block is used, it saves us almost nothing to under-use it. We therefore favor truncations which reduce the size of y to the smallest multiple of a multiplier input size that allows us to reach the target accuracy. For Virtex4 and StratixII, the size of y should target a multiple of 17 and 18 respectively. On Virtex5 and Virtex6, multiples of 17 or 24 should be investigated. Consequently, each g_j^y can take a maximum of three possible values: 0, corresponding to no truncation, and one or two soft spots corresponding to multiples of multiplier input size.

The determination of the possible values of g_j^π also depends on the DSP multiplier size, as the truncation of π'_j defines the size of the sum σ'_j, which is input to a multiplier. There are two considerations to be made: First, it makes no sense to keep guard bits to the right of the LSB of $\tilde{\pi}'_j$. This gives us an upper bound on g_j^π. Secondly, as we are trying to reduce DSP count, we should not allow a number of guard bits that increases the size of σ'_j over a multiple of the multiplier input size. This gives us a second upper bound on g_j^π. The real upper-bound in computed as a minimum of the two precomputed upper-bounds.

These upper bounds define the parameter space to explore. We also observe that the size of the multiplications increases with j in our Horner evaluation scheme. We therefore favor truncations in the last Horner steps, as these truncations can save more DSP blocks. This defines the order of exploration of the parameter space. The parameters g_j^π and g_j^y are explored using the above rules until the error $\varepsilon_{\text{eval}}$ satisfies the bound $\varepsilon_{\text{eval}} < u/2 - \varepsilon_{\text{approx}}$.

This is a fairly small parameter space exploration, and its execution time is negligible with respect to the few seconds it may take to compute all the constrained minimax approximations.

III. Examples and comparisons

Table II presents the input and output parameters for obtaining the approximation polynomials for several representative functions mentioned in the introduction. The functions f are all considered over $[0, 1]$, with identical input and output precision. Three precisions are given in Table 1. Table 2 provides synthesis results for the same experiments.

It is difficult to compare to previous works, especially as none of them reaches the large precisions we attain. Our approach brings no savings in terms of DSP blocks for precisions below 17 bits. We may compare to the logarithm unit in [1] which computes $\log(1 + x)$ on 27 bits using a degree-2 approximation. Our tool instantly finds the same coefficient sizes of 30, 22 and 13, and our implementation uses 5 DSP blocks where [1] uses 6: one multiplier is saved thanks to the truncation of y. For larger precisions, the savings would also be larger.

We should compare the polynomial approach to the CORDIC family of algorithm which can be used for many elementary functions [7], [18]. Table IV compares implementations for 32-bit sine and cosine, using for CORDIC

$f(x)$	I	S	23 bits (single prec.)			36 bits			52 bits (double prec.)		
			d	k	Coeffs size	d	k	Coeffs size	d	k	Coeffs size
$\sqrt{1+x}$	$[0,1]$	$\frac{1}{2}$	2	64	$26,20,14$	3	128	$39,32,25,18$	4	512	$55,46,37,28,19$
			1	2048	$26,15$	2	2048	$39,28,17$	3	2048	$55,44,33,22$
$\frac{\pi}{4} - \frac{\sin(\frac{\pi}{4}x)}{x}$	$[0,1]$	2^3	2	128	$26,19,12$	3	128	$39,32,25,18$	4	256	$55,47,39,31,23$
			1	4096	$26,14$	2	2048	$39,28,17$	3	2048	$55,44,33,22$
$1-\cos(\frac{\pi}{4}x)$	$[0,1]$	2	2	128	$26,19,12$	3	256	$39,31,23,15$	4	256	$55,47,39,31,23$
			1	4096	$26,14$	2	2048	$39,28,17$	3	4096	$55,43,31,19$
$\log_2(1+x)$	$[0,1]$	1	2	128	$26,19,12$	3	256	$39,31,23,15$	4	256	$55,45,35,25,15$
			1	4096	$26,14$	2	4096	$39,27,15$	3	4096	$55,43,31,19$
$\frac{\log(x+1/2)}{x-1/2}$	$[0,1]$	$\frac{1}{2}$	2	256	$26,18,10$	3	512	$39,30,21,12$	4	1024	$55,45,35,25,15$
			1	4096	$26,14$	2	4096	$39,27,15$	3	8192	$55,42,29,16$

Table II
EXAMPLES OF POLYNOMIAL APPROXIMATIONS OBTAINED FOR SEVERAL FUNCTIONS. S REPRESENTS THE SCALING FACTOR SO THAT THE FUNCTION IMAGE IS IN $[0,1]$

$f(x)$	I	23 bits (single prec.)					36 bits					52 bits (double prec.)				
		d	l	slices	DSP	BRAM	$d.$	l	slices	DSP	BRAM	d	l	slices	DSP	BRAM
$\sqrt{1+x}$	$[0,1]$	2	9	118	3	2*	3	18	351	9	3	4	32	893	21	5
		1	5	62	1	5	2	12	231	5	9	3	24	668	15	17
$\frac{\pi}{4} - \frac{\sin(\frac{\pi}{4}x)}{x}$	$[0,1]$	2	9	119	3	2*	3	20	435	11	4	4	36	1196	29	6
		1	5	64	1	11	2	12	238	5	10	3	28	809	19	18
$1-\cos(\frac{\pi}{4}x)$	$[0,1]$	2	9	119	3	2*	3	20	427	11	4*	4	36	1197	29	6
		1	5	64	1	11	2	13	240	5	10	3	24	672	15	38
$\log_2(1+x)$	$[0,1]$	2	9	119	3	2*	3	20	425	11	4*	4	33	1039	24	10
		1	5	64	1	11	2	11	214	5	22	3	26	722	17	38
$\frac{\log(x+1/2)}{x-1/2}$	$[0,1]$	2	9	116	3	2*	3	18	349	9	3	4	33	1036	24	11
		1	5	64	1	11	2	12	232	5	21	3	23	657	15	74

Table III
SYNTHESIS RESULTS USING ISE 11.1 ON VIRTEXIV XC4VFX100-12. l IS THE LATENCY OF THE OPERATOR IN CYCLES. ALL THE OPERATORS OPERATE AT A FREQUENCY OF 320 MHZ. A STAR INDICATES THAT A BLOCKRAM IS SEVERELY UNDERUSED.

LogiCore CORDIC 4.0 sin+cos 32 cyles@296MHz, 3812 LUT, 3812 FF
This work, sin alone 14 cycles@386MHz, 3 BlockRam, 7 DSP48E, 335 FF, 407 LUT
This work, cos alone 14 cycles@386MHz, 3 BlockRam, 7 DSP48E, 332 FF, 398 LUT

Table IV
COMPARISON WITH CORDIC FOR 32-BIT SINE/COSINE FUNCTIONS ON VIRTEX5

the implementation from Xilinx LogiCore [18]. This table illustrates that these two approaches address different ends of the implementation spectrum. The polynomial approach provides smaller latency, higher frequency and low logic consumption (hence predictability in performance independently of routing pressure). The CORDIC approach consumes no DSP nor memory block. Variations on CORDIC using higher radices could improve frequency and reduce latency, but at the expense of an even higher logic cost. A deeper comparison remains to be done.

IV. CONCLUSION, OPEN ISSUES AND FUTURE WORK

Application-specific systems sometimes need application-specific operators, and this includes operators for function evaluation. This work has presented a fully automatic design tool that allows one to quickly obtain architectures for the evaluation of a polynomial approximation with a uniform range reduction for large precisions, up to 64 bits. The resulting architectures are better optimized than what the literature offers, firstly thanks to state-of-the-art polynomial approximation tools, and secondly thanks to a finer error analysis that allows for truncating the reduced argument. They may be fully pipelined to a frequency close to the nominal frequency of current FPGAs.

This work will enable the design, in the near future, of elementary function libraries for reconfigurable computing that scale to double precision. However, we also wish to offer to the designer a tool that goes beyond a library: a generator that produces carefully optimized hardware for his very function. Such application-specific hardware may be more efficient than the composition of library components.

Towards this goal, this work can be extended in several directions.

- There is one simple way to further reduce the multiplier cost, by the careful use of truncated multipliers [19], [14]. Technically, this only changes the bound on the multiplier truncation error in the error analysis of II-D. This improvement should be implemented soon.

- Another way, for large multiplications, is the use of the Karatsuba technique, which is also implemented in FloPoCo [20]. It is even compatible with the previous one.
- Non-uniform range reduction schemes should be explored. The power-of-two segmentation of the input interval used in [2] has a fairly simple hardware implementation using a leading zero or one counter. This will enable more efficient implementation of some functions.
- More parallel versions of the Horner scheme should be explored to reduce the latency.
- Parameter space exploration is tuned for minimizing DSP usage, it should also be tuned to make the best possible usage of available configurations of embedded memory blocks.
- Our tools could attempt to detect if the function is odd or even [21], and consider only odd or even polynomials for such case [7], [21]. Whether this works along with range reduction remains to be explored.
- We currently only consider a constant target error corresponding to faithful rounding, but a target error function could also be input.
- Designing a pleasant and universal interface for such a tool is a surprisingly difficult task. Currently, we require the user to input a function on $[0, 1)$, and the input and output LSB weight. Most functions can be trivially scaled to fit in this framework, but many other specific situations exist.

Acknowledgements

This work was partly supported by the ANR EVAFlo project and Stone Ridge Technology.

REFERENCES

[1] D.-U. Lee, J. Villasenor, W. Luk, and P. Leong, "A hardware Gaussian noise generator using the Box-Muller method and its error analysis," *IEEE Transactions on Computers*, vol. 55, no. 6, 2006.

[2] R. Cheung, D.-U. Lee, W. Luk, and J. Villasenor, "Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method," *IEEE Transactions on VLSI Systems*, vol. 8, no. 15, 2007.

[3] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*, ser. Hewlett-Packard Professional Books. Prentice Hall, 2000.

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

[5] ——, "Floating-point trigonometric functions for FPGAs," in *Field-Programmable Logic and Applications*. IEEE, 2007, pp. 29–34.

[6] M. G. Arnold and S. Collange, "A dual-purpose real/complex logarithmic number system ALU," in *Proceedings of the 19th IEEE Symposium on Computer Arithmetic*, 2009, pp. 15–24.

[7] J.-M. Muller, *Elementary Functions, Algorithms and Implementation*, 2nd ed. Birkhäuser, 2006.

[8] D. A. Sunderland, R. A. Strauch, S. S. Wharfield, H. T. Peterson, and C. R. Role, "CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications," *IEEE Journal of Solid-State Circuits*, vol. 19, no. 4, pp. 497–506, 1984.

[9] F. de Dinechin and A. Tisserand, "Multipartite table methods," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 319–330, 2005.

[10] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2005, pp. 328–333.

[11] D. Lee, A. Gaffar, O. Mencer, and W. Luk, "Optimizing hardware function evaluation," *IEEE Transactions on Computers*, vol. 54, no. 12, pp. 1520–1531, 2005.

[12] A. Tisserand, "High-performance hardware operators for polynomial evaluation," *Int. J. High Performance Systems Architecture*, vol. 1, no. 1, pp. 14–23, 2007.

[13] F. de Dinechin, C. Klein, and B. Pasca, "Generating high-performance custom floating-point pipelines," in *Field Programmable Logic and Applications*. IEEE, Aug. 2009, pp. 59–64.

[14] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, "Multipliers for floating-point double precision and beyond on FPGAs," in *Highly-Efficient Accelerators and Reconfigurable Technologies*, 2010.

[15] N. Brisebarre and S. Chevillard, "Efficient polynomial L^∞-approximations," in *18th Symposium on Computer Arithmetic*. IEEE, 2007, pp. 169–176.

[16] D. Knuth, *The Art of Computer Programming, vol.2: Seminumerical Algorithms*, 3rd ed. Addison Wesley, 1997.

[17] S. Chevillard, M. Joldes, and C. Lauter, "Certified and fast computation of supremum norms of approximation errors," in *19th Symposium on Computer Arithmetic*. IEEE, 2009, pp. 169–176.

[18] *CORDIC v4.0 (DSD249)*, Xilinx Corporation, 2009.

[19] M. Schulte and E. Swartzlander, "Truncated multiplication with correction constant," in *Workshop on VLSI Signal Processing*, 1993, pp. 388–396.

[20] F. de Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," in *Field Programmable Logic and Applications*. IEEE, Aug. 2009, pp. 250–255.

[21] C. Lauter and F. de Dinechin, "Optimising polynomials for floating-point implementation," in *Real Numbers and Computers*, 2008, pp. 7–16.