Automatic generation of polynomial-based hardware architectures for function evaluation

Florent de Dinechin, Mioara Joldes, Bogdan Pasca

Automatic generation of polynomial-based hardware architectures for function evaluation
LIP research report 2010-14

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

Many applications require the evaluation of some function through polynomial approximation. This article details an architecture generator for this class of problems that improves upon the literature in two aspects. Firstly, it benefits from 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. As a result, architectures for evaluating arbitrary functions with precisions up to 64 bits, making efficient use of the resources of recent FPGAs, can be obtained in seconds. An open-source implementation is provided in the FloPoCo project.

1 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 such a function over some interval. We assume that f is continuously differentiable over some 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 [11]. 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 [3]. 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 [12].
- $f_{\log}(x) = \log(x + 1/2)/(x - 1/2)$ 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 [8].
- $f_{\cos}(x) = 1 - \cos\left(\frac{\pi}{4}x\right)$, 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 [7].
- $s_2(x) = \log_2(1 + 2^x)$ 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 [1].

Many function-specific algorithms exist, for example variations on the CORDIC alorithm provide low-area, long-latency evaluation of most elementary functions [13]. 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].

1.1 Related work and contributions

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

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

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

Lee et al [10] have published many variations on a generic datapath optimisation tool called MiniBit to optimize polynomial approximation. They use ad-hoc mixes of analytical 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.

The High-Order Table-Based Method (HOTBM) by Detrey and Dinechin [6] is based on polynomial approximation. Their implementation is available as open-source 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: the table sizes scale exponentially, and so does the design-space exploration time.

Tisserand studied the optimisation of low-precision (less than 10 bits) polynomial evaluators [15]. He finetunes a rounded minimax approximation using an exhaustive exploration of neighbouring polynomials. He also use other tricks on smaller (5-bit or less) coefficients that 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 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 2.2.
- 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 also use truncated multipliers.
- 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 the FloPoCo open subversion repository (it will be part of the next release of FloPoCo). This implementation is fully operational, to the point that Table 2 was obtained in less one hour.

[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 1. Multiplier blocks in recent FPGAs

- The resulting architecture may be automatically pipelined to a user-specified frequency thanks FloPoCo's pipelining framework [**?**].
- This implementation provides an easy to use interface: it inputs an arbitrary function expression, a polynomial degree, and input and output bit-width. It produces an architecture in synthesizable VHDL evaluating the function with faithful accuracy.

1.2 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. 1 It is possible to build larger multipliers by assembling these embedded multipliers [5]. Besides, these multipliers are embedded in more complex DSP blocks that also include specific adders and shifters, which the synthesis tools will use efficiently.
- 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.
- A given FPGA typically contains a comparable number of memory blocks and multipliers. When designing an algorithm for an operator, 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 provide a range of trade-offs between them.

2 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 [13].

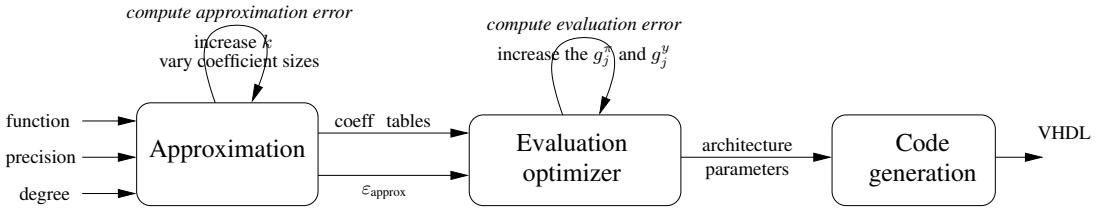

Figure 1. Automated implementation flow

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 all the following, we will consider, without loss of generality a function f over the input interval $x \in [0, 1)$.

In our implementation, the user inputs the function, input and output precisions, 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.

2.1 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 $\forall i \in \{0, \dots, 2^k - 1\} \quad f(x) = f_i(y)$, 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 and multipliers), which therefore have to be built to accomodate 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 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 size of the multipliers inputting it. 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 [3]. For an overview, see Muller [13]. Most of our contributions are independent of the range reduction used.

2.2 Polynomial approximation

One may use the well-known Taylor or Chebyshev approximation polynomials of arbitrary degree d [13]. 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. In all the following, we will call approximation error, and note $\varepsilon_{\text{approx}}$, this 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 or an ASIC, we may build adders and multipliers of arbitrary size with a one-bit granularity, so, we have one more question to answer: what is the optimal size of these coefficients? In [10], 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 asumption 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 [2] 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 [10] 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. Actually, the constraints taken by `fpminimax` are the minimal weights 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 2.4). 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.

2.3 Polynomial evaluation

Given a polynomial, there are many possible ways to evaluate it. The HOTBM method [6] 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.

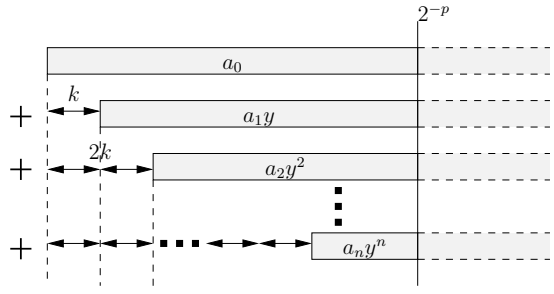

Figure 2. Alignment of the monomials

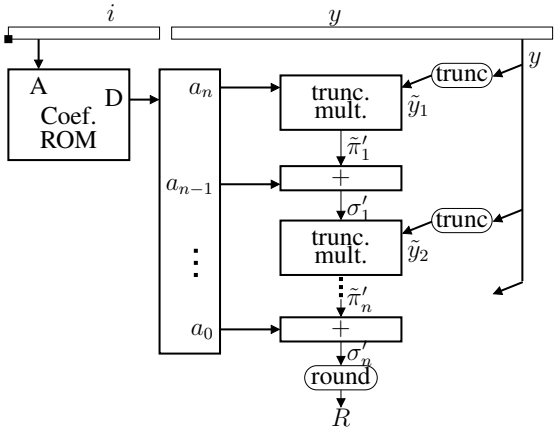

Figure 3. The function evaluation architecture

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 2.4.

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 exist [13], and this should be the subject of future work. A polynomial may also be refactored to trade multiplications for more additions [9], but this idea is mostly incompatible with range reduction.

2.4 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

- $\varepsilon_{\text{approx}}$ is the approximation error, the maximum absolute difference between any of the p_i and the corresponding f_i over their respective intervals. This com-

putation belongs to the approximation step and is also performed using Sollya [4].

- $\varepsilon_{\text{eval}}$ 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 $u/2$.

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 in all this section). 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 $y < 2^{-k}$.

2.5 Parameter space exploration

The previous error computation is implemented in C++. The parameters g_j^π and g_j^y are set to zero, then increased 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. There are many ways of improving it, in particular we should favor truncations of y to sizes that are soft spots for DSP block implementations: multiples of 17 bits for Virtex 4, multiples of 18 bits for Stratix II and later, multiples of 17 or 24 bits for Virtex-5 and Virtex-6. This is under investigation.

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 [11] 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 [11] uses 6: one multiplier is saved thanks to the truncation of y. For larger precisions, the savings would also be larger.

$f(x)$	I	23 bits (single prec.)			36 bits			52 bits (double prec.)		
		d	k	Coeffs size	d	k	Coeffs size	d	k	Coeffs size
$\sqrt{1+x}$	$[0,1]$	2	64	$27,19,11$	3	128	$40,31,22,14$	4	512	$56,45,34,24,15$
		1	2048	$27,14$	2	2048	$40,27,14$	3	2048	$56,43,30,18$
$\frac{\pi}{4} - \frac{\sin(\frac{\pi}{4}x)}{x}$	$[0,1]$	2	64	$27,21,14$	3	128	$40,33,25,15$	4	256	$56,48,39,28,18$
		1	4096	$27,15$	2	2048	$40,29,17$	3	2048	$56,45,33,19$
$1-\cos(\frac{\pi}{4}x)$	$[0,1]$	2	64	$27,21,14$	3	128	$40,33,25,15$	4	256	$56,48,39,28,19$
		1	4096	$27,15$	2	2048	$40,29,17$	3	2048	$56,45,33,19$
$\log_2(1+x)$	$[0,1]$	2	128	$27,20,12$	3	256	$40,32,23,14$	4	512	$56,47,37,27,18$
		1	4096	$27,15$	2	2048	$40,29,17$	3	4096	$44,31,18$
$\frac{\log(x+1/2)}{x-1/2}$	$[0,1]$	2	256	$27,19,11$	3	512	$40,31,22,14$	4	1024	$56,46,36,27,17$
		1	4096	$27,15$	2	4096	$40,28,16$	3	8192	$56,43,30,18$

Table 2. Examples of polynomial approximations obtained for several functions

$f(x)$	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	8	82	3	2*	3	16	282	9	3	4	29	864	23	5
		1	4	31	1	5	2	10	170	5	9	3	22	580	15	17
$\frac{\pi}{4} - \frac{\sin(\frac{\pi}{4}x)}{x}$	$[0,1]$	2	8	86	3	2*	3	18	365	11	4*	4	33	1045	27	6
		1	4	33	1	11	2	10	173	5	10	3	26	713	19	17
$1-\cos(\frac{\pi}{4}x)$	$[0,1]$	2	8	86	3	2*	3	18	365	11	4*	4	34	1084	29	6
		1	4	33	1	11	2	10	173	5	10	3	29	708	19	17
$\log_2(1+x)$	$[0,1]$	2	8	83	3	2*	3	18	358	11	4*	4	31	997	26	6
		1	4	33	1	11	2	11	170	5	10	3	21	562	14	38
$\frac{\log(x+1/2)}{x-1/2}$	$[0,1]$	2	8	81	3	2*	3	18	352	11	3	4	29	887	23	12
		1	4	33	1	11	2	10	171	5	21	3	21	558	14	74

Table 3. 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 above 250 MHz. A star indicates that a BlockRAM is severely underused.

3 Examples of application

Table 2 presents the input and output parameters for obtaining the approximation polynomials for several representative functions mentioned in the introduction. The function f 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.

4 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 will 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 [14]. Technically, this only changes the bound on the multiplier truncation error in the error analysis of 2.4. The implementation in next FloPoCo release will include this further optimization.

- Another way, for large multiplications, is the use of the Karatsuba-Ofman scheme, which is also implemented in FloPoCo [5]. 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 [3] 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 should be tuned to find soft spots related to specific features of the target hardware, in particular available configurations of embedded memory blocks, embedded multiplier input width, etc.
- Our tools should attempt to detect if the function is odd or even, and consider only odd or even polynomials for such case. Whether this works along with range reduction remains to be explored.
- Designing a pleasant and universal interface for such a tool is a surprisingly difficult task. Currently, we require the user to input a function from $[0, 1)$ to $[0, 1)$ – any function can be trivially scaled to fit in this framework. Besides, the tool should also detect if some bits of the output are constantly 1 or 0, and avoid computing them, or raise a warning.

References

[1] 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*, pages 15–24, 2009.

[2] N. Brisebarre and S. Chevillard. Efficient polynomial L^∞-approximations. In *18th Symposium on Computer Arithmetic*, pages 169–176. IEEE Computer Society Press, 2007.

[3] 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*, 8(15), 2007.

[4] S. Chevillard, M. Joldes, and C. Lauter. Certified and fast computation of supremum norms of approximation errors. In *19th IEEE SYMPOSIUM on Computer Arithmetic*, pages 169–176, 2009.

[5] F. de Dinechin and B. Pasca. Large multipliers with fewer DSP blocks. In *Field Programmable Logic and Applications*. IEEE, Aug. 2009.

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

[7] J. Detrey and F. de Dinechin. Floating-point trigonometric functions for FPGAs. In *Field-Programmable Logic and Applications*, pages 29–34. IEEE, 2007.

[8] 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*, 31(8):537–545, 2007.

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

[10] D. Lee, A. Gaffar, O. Mencer, and W. Luk. Optimizing hardware function evaluation. *IEEE Transactions on Computers*, 54(12):1520–1531, 2005.

[11] 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*, 55(6), 2006.

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

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

[14] M. Schulte and E. Swartzlander. Truncated multiplication with correction constant. In *Workshop on VLSI Signal Processing*, pages 388–396, 1993.

[15] A. Tisserand. High-performance hardware operators for polynomial evaluation. *Int. J. High Performance Systems Architecture*, 1(1):14–23, 2007.