Floating Point Calculation of the Cube Function on FPGAs

Roberto R. Osorio

***Abstract*—Specialized arithmetic units allow fast and efficient computation of lesser used mathematical functions. The overall impact of those units would be negligible in a general purpose processor, as added circuitry makes chips more complex despite most software would seldom make use of it. On the opposite side, custom computing machines are built for a specific task, and they can always benefit from specialized units if they are available. In this work, floating point architectures are proposed for computing the cube on Intel and Xilinx FPGAs. Those implementations reduce the cost and latency compared to using simple floating point multiplications and squarers.**

I. Introduction

SPECIALIZED arithmetic units may be faster and smaller than an equivalent implementation using generic operations. Within the scope of this paper, the best example is squaring. Computing the square as a simple multiplication is inefficient in both time and area compared to custom implementations, which may exploit the redundancy exhibited by the operands [1]. Provided that computing the cube is rarer than squaring, few examples exist in the literature of possible implementations beyond multiplying by the square. In [2], the cube is used to approximate the sine function, and in [3], the general case of powering is addressed. However, there exist many more cases in which this operation is required, such as the Hodgkin-Huxley model of potentials in neurons [4]; Duffing equation of damped and driven oscillators [5]; or Painlevé equations, useful in the study of quantum gravity, plasma physics, or non-linear optics [6].

As it happens with many other differential equations, computationally intensive numerical integration is required to solve the problems listed above. Also, evolutionary computing techniques are often used to fine tune the parameters in those equations, which also needs substantial computing time. Examples of the application of evolutionary computing to the problems proposed above may be found in the literature [7] [8] [9].

Custom computing machines (CCM) implement circuits specifically designed to carry out a single task [10]. At present time, FPGAs are the more versatile platform to implement CCMs. Numerical integration on a CCM achieves high computational efficiency as a large number of operations are executed concurrently by parallel pipelines with little control dependencies. If a given function is used in a CCM, the question is not why to custom-implement it but, why not.

In this work, the specific case of computing the cube in floating point is addressed. In Section II, the optimization of products and powers on FPGAs is discussed. Section III introduces the proposed architectures for computing the cube of floating point numbers in single and double precision using the most widely adopted FPGAs, and reviews some previous works. Additionally, an analysis is presented on the implementation of higher powers. Section IV deals with the implementation cost in terms of resource usage and latency, showing the potential benefits of the direct implementation of the cube compared to the use of normal multiplications, and other works found in the literature. Section V shows the maximum and mean errors for each proposal. Finally, the main conclusions are shown.

R.R. Osorio is with CITIC, Computer Architecture Group, Universidade da Coruña, Spain e-mail: roberto.osorio@udc.es

II. Computation on FPGAs

The considerations that will guide the design process are explained now. Presicion is a key factor when designing arithmetic functional units. The trade-offs between precision and complexity are analyzed in this section. Then, the basics of reconfigurable logic are introduced, with a focus on the importance of hardwired multipliers. This paper focuses in reducing the use of multipliers in order to save resources and allow for increased parallelism. In Section II-D, the case of squaring is put as a example of the strategy to follow in the remainder of the article. Explanations of computations at bit level will follow this notation: all bits are indexed according to their weight, so that a[-3] has a weight of 2^{-3}, and $a[0:-2]$ encompasses 3 bits with weights 2^0 downto 2^{-2}. This notation may require some initial effort to grasp it, but it allows to identify how operands are aligned to each other, and which bits can be truncated. Moreover, it works both for parts of the original mantissa and further calculations.

A. Precision

Full precision is desirable in many applications, mainly for result reproducibility across different computing devices. However, full precision is also highly expensive to implement, and allowing for small differences in the less significant bits may reduce the complexity of the implementation, producing smaller and faster circuits. In the context of CCMs, absolute precision is not always required and, in the particular case of evolutionary computing, randomness plays an important role in convergence, and results are not expected to be reproducible. Therefore, partial results that lay in low-weight positions may be ignored if the error of doing so is low.

Also, implementing all the rounding modes adds substantial complexity to arithmetic circuits, as an additional normalization may be necessary after rounding. Therefore, in this paper we choose to implement only truncation, and keep the

precision in the operations so that the deviation from the accurate result is never larger than 1 unit in the last place (*ulp*).

Another source of unnecessary complexity is managing subnormal numbers, as aligning the mantissa would require the use of full-length barrel shifters. The cube of any subnormal number, would always produce zero, which simplifies the implementation. However, it is possible to start with a normal operand and obtaining a subnormal result. There are two possible solutions to address this problem. First, it is possible to detect subnormal results and round them down to zero. Second, some authors [11] [12] propose to extend the size of the exponent in 1 bit. This makes the use of subnormal numbers unnecessary for all floating point operations without loosing precision. In an FPGA, adding one bit to the standard format is not a problem, and only the final results would require a conversion to the IEEE 754 format. In this paper, we have opted for the first solution as, despite some precision is lost and some error may accumulate, extending the exponent would require reimplementing all floating point operations and, moreover, commercial tools like Xilinx Vivado HLS [13] also round down subnormal numbers.

B. The technology in FPGAs

The heart of FPGAs is made of reconfigurable logic blocks, which go under different names depending on the manufacturer. Those elements store the results of multi-variable boolean operations in look-up tables (LUT) instead of computing them. Reconfigurability is achieved by updating the stored results without having to change the hardware. This simple concept, paired with sophisticated and also reconfigurable connection networks, allows implementing any digital circuit.

However, common components such as multipliers or RAM memories would consume large numbers of logic blocks if implemented on an FGPA. Therefore, FPGAs include hundreds or thousands of hardwired multipliers (currently DSP blocks) and RAM blocks that are both faster and consume less area on chip than purely reconfigurable implementations.

In floating point intensive applications, DSP usage is a limiting factor, as once all the DSPs have been used, no more operations can be implemented concurrently. Also, the more resources are used, the harder it becomes to implement an efficient routing.

Therefore, reducing the number of DSP blocks may be a priority, either by improving the algorithms or by implementing small products using reconfigurable logic. The decision of implementing a multiplier with logic is mainly empirical. A quick study conducted on Xilinx most recent FPGAs showed that, up to 9x9 bit products, Xilinx HLS prefers using logic to DSPs. Moreover, by computing the ratio between the number of logic and DSP blocks for different FPGAs, it has been found that DSPs consume a lower share of resources for product sizes above 14x14 bits. That is, implementing multipliers larger than 14x14 bits using reconfigurable logic may result in the consumption of all the available LUTs, whereas many DSPs would still be available. However, this is highly variable, as different chip models implement different mixes of logic and DSP blocks.

C. Multiplication on Intel and Xilinx FPGAs

The size of a product is the sum of the sizes of the operands (but 1 bit if both operands are signed). The complexity of the implementation, however, is proportional to the product of the sizes. In this way, doubling the precision multiplies the size of the circuit by 4. Therefore, exploiting any redundancy in the operators or operations may reduce the cost of the implementation. This is firstly explained for the well know case of squaring (Section II-D), before dealing with the cube.

Modern FPGAs implement hardwired multipliers of different sizes. Last Intel's FPGAs (formerly Altera) implement 27×27 bit signed/unsigned multipliers, that can be split in two 18×18 ones. In this way, single precision product (24-bit mantissa) may be implemented using a single multiplier, while double precision (53-bit mantissa) requires 4 multipliers. Most recent Xilinx's FPGAs implement 27×18 bit signed multipliers, so implementing single and double precision (SP and DP) products requires 2 and 8 multipliers respectively. More multipliers are needed on Xilinx FPGAs, however, their size is roughly 70% that of Intel ones and, potentially, they allow for finer granularity. If complete precision is not needed, however, DP can be computed using 6 multipliers. Multipliers on FPGAs are actually part of a DSP circuit that also implements adders before and after the multiplication. This is of great interest for splitting operands between several multipliers and combine the partial results afterwards. Finally, selected Intel's devices in the Arria 10 family implement hardwired single precision adders and multipliers. Each of those multipliers consumes one 27×27 bit integer multiplier, and the additional circuitry for implementing exponent calculation and results normalization are also hardwired. In this paper, integer multipliers are used directly, as it saves resources for single precision cube and it enables us to implement also double precision.

D. Squaring

Compared to general multiplication, squaring conceals some redundancy that can be exploited. Let's observe that $(a+b)\times(c+d) = ac+ad+bc+bd$, but $(a+b)^2 = a^2+2ab+b^2$, which saves computing one partial product. Given a number $M[0:-n+1]$ of n bits, and making $a = M[0:-n/2+1]$ and $b = M[-n/2:-n+1]$, each partial product has $1/4$ of the complexity of the full product, and a reduction in circuit area of nearly 25% is achieved. For large values of n, breaking up M in 3 or more chunks allows for additional gains as shown for Xilinx FPGAs in [14].

Intel FPGAs can compute the square in SP as a normal product, using a single multiplier; but DP would require 3 instead of 4, a 25% saving. Xilinx FPGAs use also 1 multiplier for SP, but DP can be achieved with 4. Figure 1 shows the architecture for single precision, in which the only DSP computes $a\cdot(a+2b)$ using the pre-adder, while b^2 is skipped due to its low significance. This case is a good example of the kind of exploits that will allow us to obtained optimized architectures for the cube.

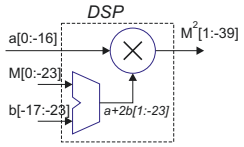


Fig. 1. Architecture for single precision square on Xilinx FPGAs.

TABLE I
TERMS FOR COMPUTING THE CUBE OF A SINGLE PRECISION MANTISSA. SIZE IS GIVEN IN BITS. THE POSITIONS OF THE LEAST AND MOST SIGNIFICANT BITS ARE ALSO GIVEN.

	$\mathbf{a}$	$\mathbf{b}$	$\mathbf{a^2}$	$\mathbf{a^3}$	$\mathbf{a^2b}$	$\mathbf{ab^2}$	$\mathbf{b^3}$
size	17	7	34	51	39	30	21
LSB	-16	-23	-32	-48	-52	-59	-68
MSB	0	-17	1	2	-14	-30	-48

III. ARCHITECTURES FOR CUBE COMPUTATION

Usually, the cube would be obtained by squaring followed by an additional multiplication. However, by splitting the operand M in 2 or 3 parts, the cube can be computed as shown in Equations 1 and 2:

$$(a+b)^3 = a^3+b^3+3\cdot(ab^2+a^2b) \quad (1)$$

$$(a+b+c)^3 = a^3+b^3+c^3+3\cdot(ab^2+a^2b+ac^2+a^2c+bc^2+b^2c)+6abc \quad (2)$$

If complete precision is required, all terms must be computed. However, if a small error is acceptable, and M is split in a wise way, some terms can be eliminated keeping the error lower or equal than 1 *ulp*.

The procedure employed to evaluate the architectures is the following: all the architectures are non-pipelined. This simplifies comparing results, as pipelining introduces significant overhead that is highly dependent on the number of stages in the pipeline. In Section IV, high-level synthesis (HLS) is used to pipeline the architectures for several target frequencies. The normal work-flow is followed in HLS and no third-party tools are used for synthesis.

A. Single precision in Xilinx FPGAs

The proposal for single precision is dividing the mantissa M in 2 parts $a[0:s]$ and $b[s\text{-}1:\text{-}23]$. By choosing s in the [-17:-14] range, terms ab^2 and b^3 have a weight much lower than the *ulp* and some computations can be avoided. For the sake of concreteness, let's assume $s = $ -16. Then, the size of the different terms, and the weight of the LSBs and MSBs, are shown in Table I.

The following explanations refer to the circuit in Figure 2. The way to compute the significant terms (that is, a^3 and $3a^2b$) is the following: a^2 is obtained first, using one 27x18 multiplier ($DSP1$). Then, M and $2b$ are added together to obtain $a+3b$. This sum is computed at no cost at the pre-adders before multiplying by a^2, obtaining a^3+3a^2b.

The term a^2 must be divided in 2 parts, 17-bit each. Thus, the product requires two multipliers to obtain two partial

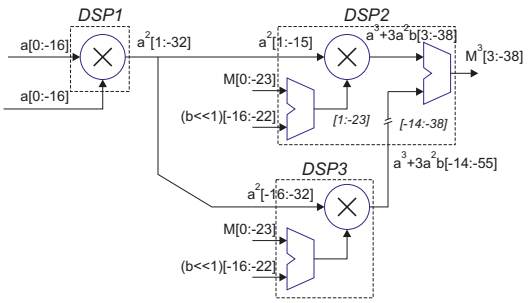


Fig. 2. Architecture for single precision cube on Xilinx FPGAs.

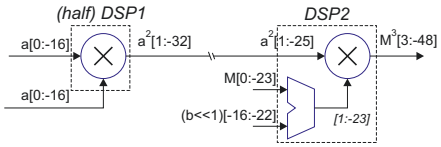


Fig. 3. Architecture for single precision cube on Intel FPGAs.

products: $DSP2[3:\text{-}38]$ and $DSP3[\text{-}14:\text{-}55]$, that must be added, discarding the 17 LSBs of $DSP3$. The sum of $DSP2$ and $DSP3$ is calculated in the post-adder of $DSP2$. Finally, the sum is normalized, which is not shown in Figure 2.

Regarding the error, there is no truncation, and the only differences with other algorithms would be due to not applying rounding. All the bits introduced in $DSP1$ and $DSP2$ are needed to obtain the desired precision. For $DSP3$, however, the useful bits are equivalent to a 10x17 bit product. Hence, using a DSP block instead of a reconfigurable logic is the best option, despite many bits are computed and discarded.

B. Single precision in Intel FPGAs

When using Intel FPGAs, the circuit is shown in Figure 3. In this case, a^2 is obtained using one 18x18 multiplier, which is half of the capabilities of $DSP1$, and the other half may be used for other purposes. Then, only one additional 27x27 multiplier is needed, at $DSP2$. At the pre-adder, $a+3b$ is computed as a 25-bit number, and then multiplied by a^2 truncated to 27 bits. The result must then be normalized.

As in previous section, the only source of error is skipping rounding. All the bits entering $DSP1$ and $DSP2$ are needed to obtain the desired precision. The only truncation happens to a^2 before $DSP2$, but the impact is lower than 2^{-26}.

C. Double precision in Xilinx FPGAs

In double precision, the 53 bits of the mantissa must be split in three parts, as the hardwired multipliers in Xilinx DSP blocks can deal with up to 26-bit unsigned operands. In Table II, the size and bit-ranges for the terms in equation 2 are shown. Other terms, such as ac^2 have such a low weight that there is no need to compute them. As it can be seen, many of the computed bits will be discarded due to their low weight, but they must be obtained in many cases.

Let's take a^2b as an example, for which at least 13 bits must be discarded. First, a^2 can be computed, using one DSP

TABLE II
TERMS FOR COMPUTING THE CUBE OF A DOUBLE PRECISION MANTISSA ON XILINX FPGAS. SIZE IS GIVEN IN BITS. THE POSITIONS OF THE LEAST AND MOST SIGNIFICANT BITS ARE ALSO GIVEN.

	$\mathbf{a, b, c}$	$\mathbf{a^3}$	$\mathbf{a^2b}$	$\mathbf{a^2b}$	$\mathbf{ab^2}$	$\mathbf{abc}$	$\mathbf{b^3}$
size	17, 17, 19	51	51	53	51	53	51
LSB	-16, -33, -52	-48	-65	-84	-82	-101	-99
MSB	0, -17, -34	2	-15	-32	-32	-49	-49

and producing a result in the [1:-32] range. Then, it can be multiplied by b. If a^2 is truncated to [1:-24], the product by b can be computed with a single multiplier. However, a maximum error $< 2^{-40}$ would be introduced, which is unacceptable. Computing $ab \cdot a$ after truncating ab to [-16:-41] would produce the same error. If a^2 is computed first, obtaining a^3, a^2b and a^2c will require 7 DSPs, but a large number of bits will be discarded.

Therefore, a better strategy must be devised. The proposed architecture is shown in Figure 4, where a^2 is computed in $DSP1$. Then, a^2 multiplies $M+2b+2c$, being M the whole mantissa. The product is split into $DSP2$, $DSP3$, $DSP4$ and $DSP5$. The pre-adders of those DSPs are used to add $2b+2c$, so that $a^3+3a^2b+3a^2c$ is computed using only 5 DSPs.

Computing b^3 is simpler than it seems as, for n bits of b, there are only 2^n possible results. If $n \leq 6$, a single LUT (look-up table) may store/compute 1 bit of the result, and n LUTs will produce b^3 concurrently. A quick analysis reveals that using the 6 MSBs of b is insufficient to obtain the desired precision. However, above 6 bits, the number of LUTs required to store the result grows quadratically. For a desired precision of 9 bits, 57 LUTs are needed, which may be considered too high. However, it is possible to reduce that number by splitting and approximating the calculation. This technique will be exploited several times in this paper in order to reduce the size of small multipliers implemented with LUTs.

Thus, b^3 is computed as follows: $x = b$[-17 : -22], $y = b$[-17 : -19], $z = b$[-23 : -25]. A good approximation would be x^3+3y^2z, where the weight of each term has been obviated. Both x^3 and $3y^2z$ can be computed as 6-bit functions and added together with a cost of only 25 LUTs and a maximum error of $\approx 2^{-54}$.

The remaining terms are $3ab^2$ and $6abc$. As a first option, ab^2 can be computed as b^2 truncated to [-33:-58] multiplied by a. Multiplying by 3 could be achieved using the pre-adders. This requires only 2 DSPs and the error is lower than 2^{-57}. However, computing $6abc$ with low error would either require 2 additional DSPs or a large number of LUTs.

Therefore, the following implementation is preferred: ab is computed first using $DSP6$. Then $b+2c$ is obtained using LUTs. Next, $3b+6c$ is calculated in the pre-adders of $DSP7$ and $DSP8$ by adding $b+2c$, and the same amount shifted 1 bit. Finally, the 2 parts of ab are multiplied. The post-adder of $DSP7$ is used to add a^2c, which is also approximated using 8 LUTs. The error of the result is lower than 2^{-55}.

Finally, all the terms are added up as shown in Figure 5. Up to 3 $n-$bit terms can be added using just n LUTs. Thus, the terms are assembled in 6 groups and added in 2

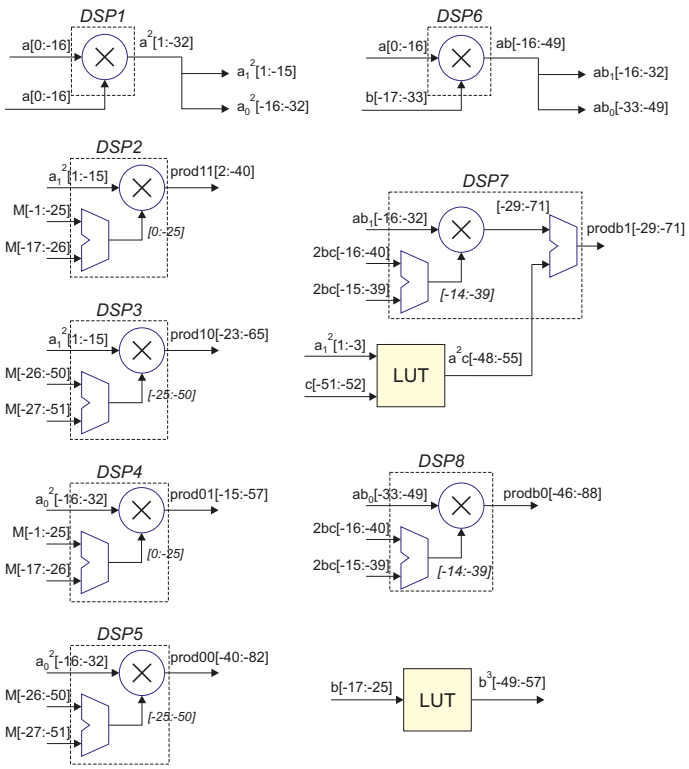


Fig. 4. Architecture for double precision cube on Xilinx FPGAs.

levels, grouped by similar lengths. Alternatively, the 6 groups could be compressed and then added as proposed in [15] [16]. This technique is sometimes referred to as *bit-heap*, as in [2]. However, this possibility has been tried and it does not reduce the hardware cost significantly. There are, however, other drawbacks. The compressor was described in Verilog making use of recent Xilinx LUTs at low level, so the implementation cannot be ported to other FPGAs. Moreover, the high-level synthesis tool manages the compressor as a black box, and it is unable to pipeline it. Therefore, more than one implementation must be provided and hand selected depending on the targeted clock frequency. Hence, letting Vivado to implement the sums is both simple and reasonably efficient.

The accuracy of the calculation is bounded by terms a^2c[-48 : -55] and b^3[-49 : -57]. This leaves 3 guard bits to let carry propagate without introducing any error. All the DSP blocks make use of all their input bits, with the exception of $DSP8$, for which a 9×9 bit product would introduce an error lower than 2^{-55}. Therefore, an alternative implementation could spare $DSP8$ and use LUTs instead. In this work, it is preferred using a DSP, which also provides a pre-adder to compute $6bc$.

D. Double precision in Intel FPGAs

The larger multipliers in Intel's DSP blocks allow to split the mantissa in only 2 parts and avoid computing a large number or terms. In Table III, the size and bit range of the terms in Equation 1 are shown.

From those data, a^2 can be computed with full precision using $DSP1$ in Figure 6. Then, as most of the bits of a^3 are

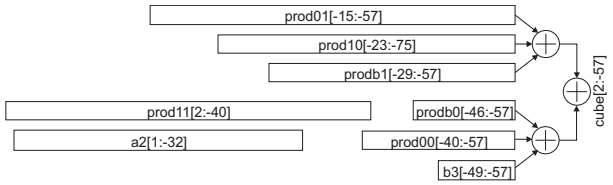


Fig. 5. Architecture for sum on Xilinx FPGAs.

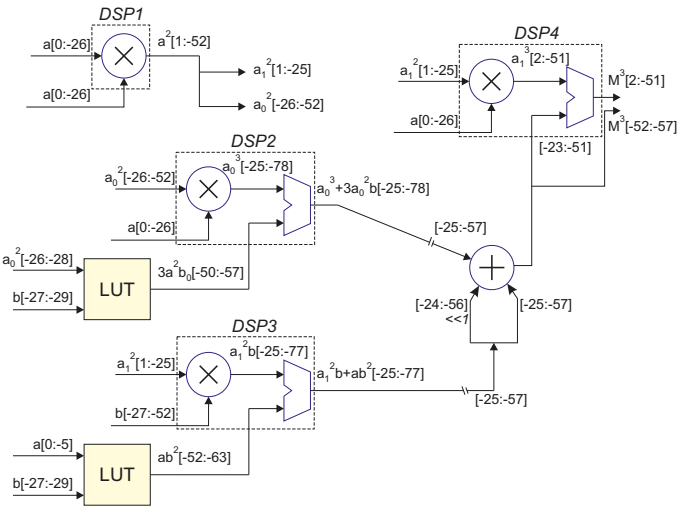


Fig. 6. Architecture for double precision cube on Intel FPGAs.

needed, it must be computed with full precision as well, using $DSP2$ and $DSP4$. At is can be seen in the figure, those DSP blocks are also used to add other terms as it will explained in the following paragraphs.

Next, $3a^2b$ is obtained multiplying a^2 by $3b$. If a^2 is truncated to 27 bits, only 1 DSP is needed. However the error would be $n \leq 2^{-50}$, which should be improved. Thus, there are two choices. For the first one, a^2b is computed using 1 DSP and a half. The full DSP computes $a^2[1:-23] \times b$, while the half one computes $a^2[-24:-39] \times b[-27:-44]$. The truncation error is $< 2^{-66}$, and it happens in the half-size product. The second option uses LUTs instead of half a DSP to multiply heavily truncated operands: $a^2[-26:-28] \times b[-27:-32]$. As only 16 LUTs units are required, this option is preferred, as it saves valuable DSP blocks. Also, the LUTs will actually compute $3a^2b$ with little additional cost. Thus, the LUT attached to the post-adder in $DSP2$ will produce the least significant bits in $3a^2b$, while $DSP3$ will deliver the most significant ones.

The term ab^2 can be computed using LUTs as well. First, $b[-27:-29]$ is used to calculate b^2. Next, $ab^2[-52:-63]$ is obtained multiplying by $a[0:-5]$, using LUTs as well. This term can be added up to the result using the post-adder at $DSP3$.

An additional adder is used to sum the result from $DSP2$ plus 3 times the result from $DSP3$. The latter is achieved by adding the output from $DSP3$ twice, one of them left-shifted one position. Adding those 3 terms has the same cost as adding just 2. The final result is obtained in the post-adder of $DSP4$, which only requires being normalized. Actually, the post-adder cannot handle the bits in the [-52:-57] range, which are just bypassed from the lower entry of the adder.

The only truncated operands are $3a^2b_0[-50:-57]$ and $ab^2[-52:-63]$, so precision is guaranteed. All the DSPs make use of all their incoming bits, so none of them can be substituted by a smaller multiplier.

TABLE III
TERMS FOR COMPUTING THE CUBE OF A DOUBLE PRECISION MANTISSA ON INTEL FPGAS. SIZE IS GIVEN IN BITS. THE POSITIONS OF THE LEAST AND MOST SIGNIFICANT BITS ARE ALSO GIVEN.

	a	**b**	$\mathbf{a^2}$	$\mathbf{a^3}$	$\mathbf{b^2}$	$\mathbf{a^2b}$	$\mathbf{ab^2}$	$\mathbf{b^3}$
size	27	26	54	81	52	80	79	78
lsb	-26	-52	-52	-78	-104	-104	-130	-156
msb	0	-27	1	2	-53	-25	-52	-79

E. Review of other works

There are only a few papers in the literature that address the computation of the cube. The most interesting ones, by Piñeiro et al. [3] [17] use minimax quadratic approximation [18] for implementing several powers, including the square root and reciprocals of single precision floating point numbers. Basically, those functions are locally approximated by a quadratic polynomial. Such approximation only works in a narrow subinterval so, in order to compute the function in the whole [1.0, 2.0) interval, the latter is divided and several segments (from 18 to 1024 depending on the power) and, for each segment, a different set of polynomial coefficients is used. Hence, given a value X, the 7 to 10 most significant bits are used to address tables that will produce 3 coefficients. Then, the 16 to 13 least significant bits of X will be used to compute the polynomial using those coefficients. The main advantages of this method are its simplicity and the possibility of using the same hardware for computing different functions by selecting different tables. The main disadvantage is to be limited to single precision, as extending the idea to double precision would require enormous tables. Therefore, no paper has been published applying the same technique for double precision. Actually, and in order to compute the double precision reciprocal, a minimax approximation is used in [19] as the starting point for the Goldschmidt algorithm, but not on its own.

In the case of the single precision cube, 3 tables totaling 24.5 Kb are needed (42×2^9). In [17] they are implemented using LUTs , but actually it makes more sense to consolidate the 3 tables and implement then using 2 BRAM (18 Kb each), or 1 BRAM plus 117 LUTs. Next a 9×9 squarer is needed, plus a 12×7 and a 14×15 multipliers. The squarer is optimized as also is, to some extent, the 12×7 multiplier. In [17], all the computations, are carried out using LUTs, whereas it would make more sense to implement both 12×7 and 14×15 products using DSPs, as Xilinx Vivado does.

Another work that includes the calculation of the cube can be found in [2] as part of the computation of the sine function using Taylor series. In that paper, the angle is reduced so that the argument of the cube is lower than 2^{-6} for single precision and 2^{-10} for double precision, although the later is

not implemented. Provided that the argument is significantly lower than 1, the cube can be computed using a combination of calculations and table look-up. Overall, the solution is convenient only for small precisions, as a significant amount of logic is required even when z^3 is calculated only with 6 to 14 bits of precision.

F. Fourth power computation, and beyond

Provided that squaring requires so little resources, double squaring is the most efficient solution to compute the fourth power in most cases. In this way, and for single precision, only 2 DSP blocks are needed for Xilinx's and Intel's FPGAs. In any case, the method proposed in [3] should be applicable and probably more convenient.

For double precision, 8 and 6 DSP blocks would be required for Xilinx and Intel, respectively. A close examination shows that it is not possible to reduce the number of DSP blocks for Xilinx FPGAs. This is due to the fact that the mantissa must be divided in 3 pieces, producing 4 significant partial products. Therefore, adding them up, and computing the square again, is the most efficient way to compute the fourth power.

For double precision in Intel FPGAs, however, the number of DSP blocks may be reduced to 4, saving 2 DSPs. The architecture in presented in Figure 7. The square of the mantissa M is $a^2+2ab+b^2$. The dominant terms are computed at $DSP1$ and $DSP2$. By adding the less significant half of a^2 and the most significant part of ab, the most significant bits of $a^2 \times (a^2+2ab)$ can be obtained at $DSP3$ and $DSP4$. All the remaining terms are either too small (such as b^4) or can be obtained using LUTs. As an example, 6 bits from a_2^2 and 4 bits from ab_0 are multiplied in the top-left LUT and, by carefully arranging the partial products, only 26 6-input LUTs are needed. The bottom-right LUT squares the 7-bit input using 18 6-input LUTs. The latter case makes use of the pre-adder at $DSP4$ which, actually, is not available as an output of the DSP block. Therefore, that operation must be computed again using LUTs for its most significant bits. Finally, the 6 partial products are added up and normalized to obtain the new mantissa.

1) Beyond the fourth power: For higher powers, a combination of squaring and cubing should be the best option. Table IV shows a comparison of the minimum number of DSPs needed to compute several powers for Xilinx and Intel FPGAs using single and double precision. For each product or power 3 figures are given. The first one makes use, when possible, of optimized square, cube and fourth power implementation as described in this paper. The second figure makes use of optimized squaring as described in Section II-D. The third one uses normal products, exclusively. For each power, the best option has been chosen for each case. Thus, the sixth power is obtained using the cube followed by squaring, and the ninth one by cubing twice. The double precision fourth, eight and twelfth powers use the implementation from Figure 7 in Intel FPGAs.

Whereas it is difficult that the higher powers are ever used in real applications, Table IV shows an interesting fact: computing those integer powers require less DSP blocks than

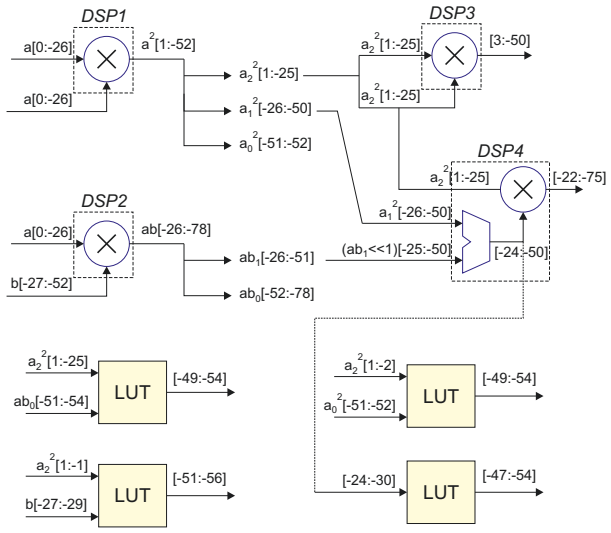


Fig. 7. Architecture for double precision fourth power on Intel FPGAs.

the straightforward solution $m^n = exp(n \cdot ln(m))$, as exp and ln require 26 and 61 DSP blocks, respectively, according to Xilinx HLS. However, better implementations are likely to exist.

Despite the work in [3] does not mention any power above the cube, we should assume that the same technique should be applicable to at least some higher powers, although restricted to single precision. Therefore, using the polynomial approximation proposed in [3] could be the best option for single precision 4th, 5th, or even higher powers. However, this is difficult to state without applying the minimax technique also for those powers and test their accuracy.

2) Assessment: Overall, the benefit of implementing the cube and fourth power vary depending on the precision and the size of DSPs. Intel´s DSP size clearly targets floating point applications, so products can be implemented using fewer blocks. In the particular case of single precision, the proposed architectures do not bring any advantage, apart from the fact that normalizing the cube is implemented just once, instead of twice if two products are used. For double precision, savings are significant compared to using only optimized squaring, reducing more than 30% of the DSPs in most cases.

For Xilinx FPGAs, the smaller and rectangular multipliers make the design process more challenging. However, there are more opportunities for optimizing the operations. Hence, nearly 50% of DSP block may be saved for single precision, and more than 25% for double precision.

The technique of optimizing calculations by skipping unnecessary partial products and removing redundant operations can be exploited for other functions, such as trigonometric ones, natural logarithms and exponentiation. The obvious choice is the optimization of power operations if Taylor series are used, but some functions allow for other implementations similar to the one proposed here. An example splitting the argument in exponentiation may be found in [9].

TABLE IV
NUMBER OF DPS BLOCKS FOR THE IMPLEMENTATION OF DIFFERENT POWERS. THE FIRST FIGURE IN THE TRIPLETS IS ACHIEVED USING THIS WORK; THE SECOND ONE MAKING USE OF OPTIMIZED SQUARES, WHEN POSSIBLE; AND THE THIRD ONE PLAIN MULTIPLICATIONS.

	Xilinx		Intel	
	SP	**DP**	**SP**	**DP**
product	3/3/3	8/8/8	1/1/1	4/4/4
square	1/1/3	4/4/8	1/1/1	3/3/4
cube	2/4/6	8/12/16	1.5/2/2	4/7/8
fourth	2/2/6	8/8/16	2/2/2	4/6/8
fifth	5/5/9	16/16/24	3/3/3	8/10/12
sixth	4/5/9	12/16/24	2.5/3/3	7/10/12
seventh	7/8/12	20/24/32	3.5/4/4	11/14/16
eighth	3/3/9	12/12/24	3/3/3	7/9/12
ninth	4/5/12	16/20/24	4/4/4	8/13/16
tenth	6/6/12	20/20/24	4/4/4	13/13/16
eleventh	9/9/15	28/28/40	5/5/5	15/17/20
twelfth	6/6/12	16/20/32	3.5/4/4	8/13/16

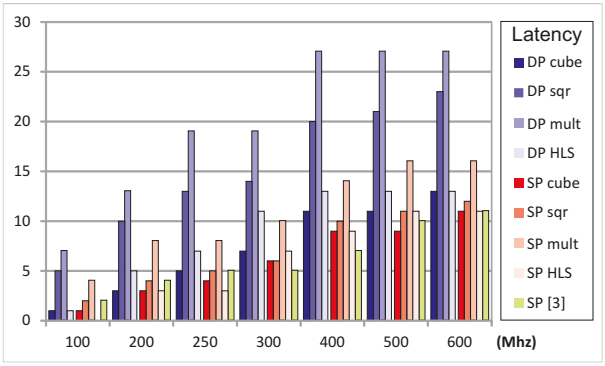


Fig. 8. Latency in cycles for single and double precision implementations of the cube function on Xilinx Virtex Ultrascale. *Mult* uses two normal products, *sqr* an optimized squarer plus one product, *HLS* an integer cube synthesized by Vivado, and *cube* the proposed architecture. Results for [3] SP using DSPs are also shown.

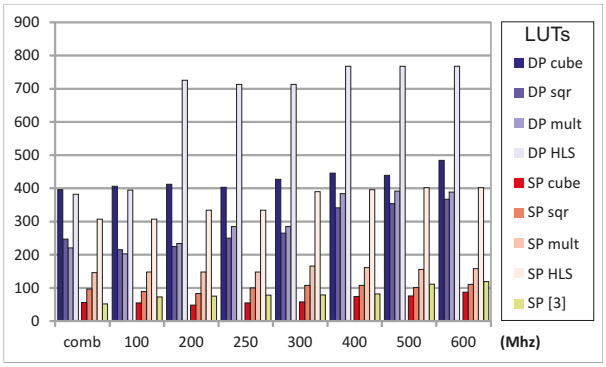


Fig. 9. Number of LUTs for single and double precision implementations of the cube function on Xilinx Virtex Ultrascale using 4 different strategies and [3] (only SP).

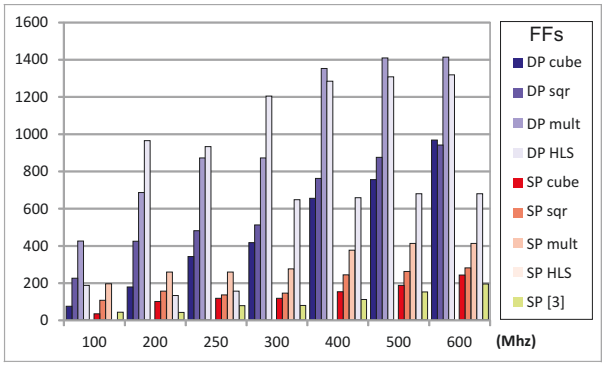


Fig. 10. Number of FFs for single and double precision implementations of the cube function on Xilinx Virtex Ultrascale using 4 different strategies and [3] (only SP).

IV. IMPLEMENTATION COST AND PERFORMANCE

In Section III, the design process focused on reducing the number of DSP blocks. In this section, the total cost of implementation will be discussed, including a comparison with more basic alternatives, with other works found in the literature, and the impact of the cube in a larger application.

A. Cube function assessment

The performance and cost of the proposed architectures is compared to 4 different implementations. First, the cube is computed as 2 sequential floating point multiplications. Then, as an optimized squaring followed by an additional floating point multiplication. Next, Xilinx HLS code has been written that computes the exponent, but lets HLS to implement the cube of the integer mantissa. Finally, the single precision implementation from [3], using one DSP block and pure logic.

Two hardware description languages haven been used. For the non-pipelined architectures in Section IV-A1, VHDL allowed a low level specification of the circuits without sequential elements. For the pipelined architectures in Section IV-A2, Xilinx Vivado HLS has been used. This consists of C++ code that can be both high or low level. At high level, it has been used to automatically implement floating point products and additions. At low level, C++ code that mimics VHDL has been written, specifying integer arithmetic operations, alignments and truncations. The HLS compiler is able to interpret those operations and translate them into hardware with only a small overhead compared to VHDL. The main advantage is that HLS is able of retiming the circuits provided a target frequency.

In both cases, the code specifies how the mantissa and exponent are obtained. The former, according to the procedures described in Section III. The latter is obtained in 2 steps. First, the original value is multiplied by 3, and the bias of the exponent is compensated. Then, the exponent can be incremented in 1 or 2 units if the computed mantissa needs to be normalized 1 or 2 positions.

1) Non-pipelined architectures: In Table V the number of DSPs and LUTs/ALMs is presented for all the non-pipelined architectures. As it has been shown before, computing the cube reduces the number of DSP blocks with respect to implement straight multiplications, but also compared to squaring the operand followed by one multiplication. For single precision, the number of LUTs/ALMs is also reduced. However, the double precision architectures make use of look-up tables, increasing the usage of reconfigurable logic. This is also true, to a lower extent, for the optimized square. This question will be discussed in more detail later in this section. The selected

TABLE V
IMPLEMENTATION COST OF NON-PIPELINED CUBE ARCHITECTURES COMPARED TO USING ONE SQUARER AND MULTIPLIER, AND USING TWO MULTIPLIERS.

	Single precision				Double precision			
	Xilinx		Intel		Xilinx		Intel	
	DSP	LUT	DSP	ALM	DSP	LUT	DSP	ALM
cube	2	56	1.5	57	8	316	4	226
sqr	4	95	2	65	10	247	7	211
mult	6	144	2	70	12	219	8	209
HLS	4	307			18	380		
[3] **DSP**	2	52	1	86				
[3] **logic**	0	403	0	230				

devices have been Xilinx Virtex Ultrascale VU440 and Intel Arria 10 SX 160.

The results labeled as HLS show the importance of designing architectures that skip useless computations. In the proposed implementation, the mantissa is squared first and then truncated to 24/53 bits before being multiplied again. As HLS itself is unable to discover which bits in the intermediate computations can propagate to the final cube value, it devotes a large number of DSP blocks to compute unnecessary bits. This is more notable for double precision.

Also in Table V, the implementation results for the quadratic approximation method proposed in [3] are presented. Those results are only available for single precision, as explained in Section III-E. As it can be seen, the cost in DSPs and logic is very similar to our work but, on top of that, 2 BRAM or 2 M20K memory blocks are also required to store the coefficients for the approximation. These figure have been obtained using one optimized squarer that only requires 46 LUTs. Additionally, 2 full DSPs are needed in Xilinx FPGAs, or 2 half ones in Intel FPGAs. When all the products are implemented using logic, the consumption of LUTs or ALMs increases notably, as shown in the last row in Table V. Implementation details for the work in [2] are not listed here because, as stated in *table 1* of the same paper, only between 6 and 14 bits of precision are obtained, and the results in *table 3* are not sufficiently detailed to tell the cost of implementing the cube from the cost of implementing other parts of the architecture.

2) Pipelined architectures: Additionally, Xilinx Vivado HLS has been used to obtain pipelined implementations at different clock speeds by specifying the desired cycle duration in ns. Equivalent results could be obtained for Intel's devices using Quartus Prime with the required expertise.

In Figures 8, 9, and 10, four different implementations are shown for each precision and target frequency. These are: implementing the cube using two normal products (*mult*), one optimized squarer followed by one normal multiplication (*sqr*), the HLS implementation of the cube (*hls*), and the architectures proposed in this paper (*cube*). Also, and only for single precision, the implementation using DSPs of [3] is shown.

Latency is plotted in Figure 8. Increasing clock speed in the horizontal axis implies reducing the length of pipeline stages and, therefore, increasing the number of those and the latency. This is true with some exceptions, as Vivado manages to clock the designs at 500 MHz without increasing the latency with respect to 400 MHz, and the same happens between 250 and 300 MHz. As it can be seen, *cube* implementations requires always fewer cycles to complete for each target frequency and, moreover, the double precision implementation even outperforms the single precision version of *mult*. Compared to using an optimized squarer, *cube* also exhibits lower latency, especially for double precision and high frequencies. This is important because *sqr* already improves *mult* significantly. Regarding *hls*, it achieves the second lower latency for double precision, and is comparable to *cube* for single precision. The method in [3] manages to keep latency low in most cases, which can be explained because never 2 DSPs are chained.

The number of LUTs used by each architecture is shown in Figure 9. This is the only graph in which *cube* does not outperform the other options. It can be noticed again that double precision *cube* requires a large number of LUTs, as it implements several small products using reconfigurable logic. On the contrary, *mult* and *sqr* rely almost exclusively on DSP blocks. Also, *hls* requires by far the largest number of LUTs. Regarding [3], it averages 30% more LUTs than our proposal. This is a small difference, but it must be considered that it also requires 2 BRAM. It has been tested that when Vivado tries to use LUTs to implement the coefficient tables or the products in [3], the number of consumed LUTs grows to several hundreds, even for low frequencies.

For single precision, *cube* is the architecture with lower cost, followed by *sqr* and then *mult* and *hls* as the most expensive ones. With the exception of *hls*, none of them implement products using LUTs, and the numbers are nearly constant across the horizontal axis, growing slightly at high speeds. For single precision *cube*, the number of LUTs in the 200 MHz version is lower than for the non-pipelined version. Apparently, some synergies in the pipelined version allow to save 8 LUTs. The difference is not high, but it has not been possible to identify the reasons for that unexpected behavior. The same happens at 100 MHz for double precision *cube*.

Next, the number of flip-flops is displayed in Figure 10. The usage increases with the number of stages in the pipeline, so there is a clear similarity with Figure 8. However, there are some differences when Vivado finds a way to use the internal flip-flops in DSP block in some cases. This is evident for the double precision *hls* at 250 MHz, which requires less FFs than at 200 MHz. In any case, *hls* is one the most expensive implementations in terms of FFS. Also, for the single precision *cube* unit at 300 MHz, latency increases, but the number of flip-flops does not. The architecture proposed in [3] requires fewer flip-flops, partly due to the lower latency, and partly because the internal registers in the BRAMs are not accounted. High FF counts are always a drawback of highly clocked designs, which could raise some concerns about increasing frequency. Hence, the balance between the consumption of different types of resources on FPGAs is addressed in the following paragraph.

Overall, the architecture in [3] consumes a similar amount of LUTs and flip-flops, with reduced latency, and the same amount of DSP blocks. However, it requires significant amount of internal SRAM to store the coefficient tables, and it is

limited to single precision. It must also be considered that the main aim of the work in [3] is not to compute the cube, but mostly the square root and its reciprocal and, in those cases, it is among the state of the art.

One of the main aims of our work is reducing the number of DSP blocks required to implement the cube. With the results from Figures 8, 9, and 10, is now possible to justify that decision. An analysis has been carried out comparing the implementation cost with the resources available in all the Virtex Ultrascale line of devices. For single precision, our architecture takes between 0.10 and 0.50% of the available DSPs in the device whereas, even for the most pipelined version of the architecture, only 0.01-0.02% of LUTs and 0.01-0.03% of FFs are used. For double precision, between 0.28-1.33% of DSPs, but 0.02-0.14% of LUTs and FFs. In both cases, DSP resources are consumed in greater amounts than LUTs and FFs. The fact is also true for normal products and optimized squarers, only that even more DSPs are requiered to implement the same operations. Therefore, this results seem to justify the choice of focusing on reducing the number of DSPs in the architectures. At the same time, it shows that the consumption of FFs is affordable even for high frequencies, as they are not the limiting factor.

B. Case example: Painlevé's second equation

In the previous section, it has been demonstrated that custom units to compute the cube function allow to reduce both latency and the consumption of valuable DSP blocks. However, this means little if those advantages are diluted in the context of a larger application. Therefore, a simple example is presented here, Painlevé's II equation [6] [8] reduced to real numbers. This is a second order differential equation, and the part relevant for our example is shown in Eq. 3.

$$y^3 + ay + b \tag{3}$$

The equation has been implemented in Xilinx Vivado HLS in four different ways using double precision arithmetic. The most significant results are shown in Figure 11. It can be seen that using the custom cube unit reduces the latency (Figure 11(a)) of the whole circuit for most operating frequencies, but the higher one. Also, the number of DSP blocks (Figure 11(b)) is significantly lower. Averaging in the range of considered frequencies, the custom cube saves 10% of computing cycles when compared to using 2 multiplications, and 8% compared to using the optimized squarer. Average savings in DSP blocks are 37% and 22% respectively. It can also be seen that the number of DSPs lowers starting at 400 MHz. At low frequencies, Vivado HLS prefers to implement floating point adders using the integrated integer adders in DSP blocks. At higher frequencies, however, it favors using LUTS. To a lesser extent, this also happens with floating point multipliers. Whereas it would be interesting to keep the same number of DSPs for all the implementations, there is no way to force Vivado HLS to use DSPs in all the frequency range. But savings in DSPs do not come without a cost. In Figures 11(c) and 11(d) it can be seen that the number of flip-flops and LUTs grows accordingly as frequency increases, and the sudden increase of the number of LUTs at 400 MHz is certainly notable.

The same results have been obtained for the non-forced Duffing's oscillator [5] [7], so the are not shown here. In both Painleveé's and Duffing's equations, the cubic term is present, but not the quadratic one. Therefore, there is no gain in computing the square first and share computations with the cube. Despite focusing on Painlevé's could look like cherry picking, this type of cubic equations, which are known as depressed cubics [20], are quite common.

Finally, when these or similar equations are computed in the context of numerical integration, a stream of data is introduced and computed without dependencies. The longer the stream, the less relevant latency becomes. For example, in a stream of 100 data points, two architectures with latency of 20 and 25 cycles will actually end in 120 and 125 cycles respectively. Therefore, it seems reasonable that the impact of reduced latency could be diluted in some cases. However, the savings in DSP consumption are always preserved.

C. Case example: Hodgkin-Huxley's potential equation

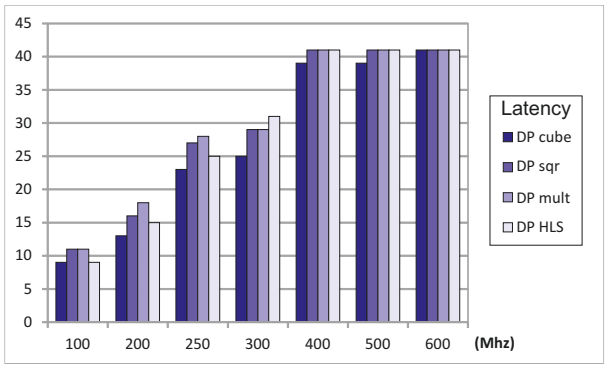
As a second example, the integration of the electric potential (Equation 4) in the Hodgkin-Huxley's model [4] [9] is presented. It includes the calculation of m^3 and n^4.

$$\begin{aligned}\frac{dv}{dt} = iC_m \cdot [I - g_{Na} m^3 h(v - E_{Na}) \\ - g_K n^4 (v - E_K) - g_L (v - E_L)]\end{aligned} \tag{4}$$

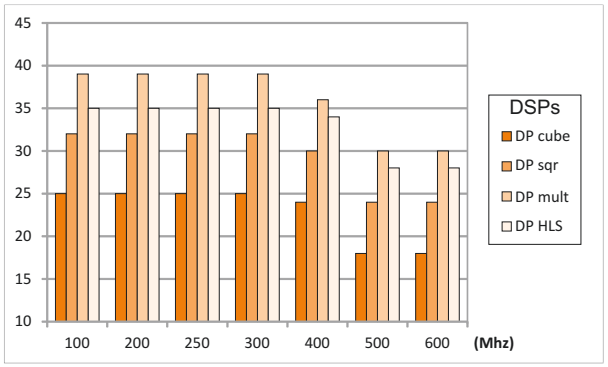
Both can be computed as normal products, or combined with an optimized square. Furthermore, m^3 can be calculated directly in 2 ways as already shown. All this possibilities have been tested, and results have been obtained using Xilinx Vivado HLS and graphed in Figure 12. In this example, the critical path does not include the computation of the cube. Hence, latency is basically the same for all the architectures and it only changes with the operational frequency. The number of DSPS, however is still dependent on the use of optimized square and cube functions. As it can be seen in Figure 12(b), the smaller savings are with respect to using the optimized squarer ($\sim$6%); followed by the HLS cube ($\sim$9%); and simple multipliers ($\sim$20%). Despite there is not improvement in latency, this new example makes clear that resource savings are still significant even when for more complex equations. As already noticed for Painlevé's II equation, Vivado HLS decides to use less DSPs for adders and more LUTs (Figure 12(d)) starting at 400 MHz, but this does not change the trend in the number of DSPs consumed by each architecture.

V. Additional testing of accuracy

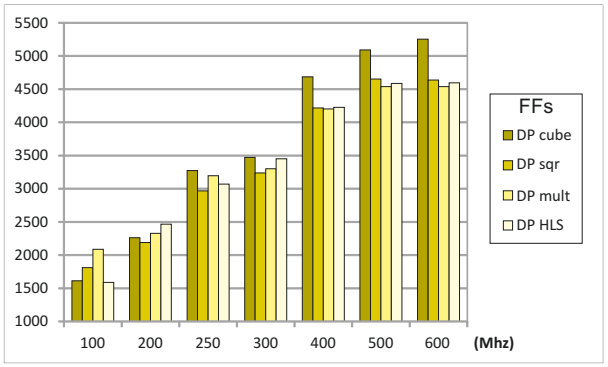
An analysis of the terms used for computing the cube has been carried out when describing each of the 4 architectures. Whereas that analysis shows that truncation errors should never sum up to more than 1 ulp, this is further tested in this



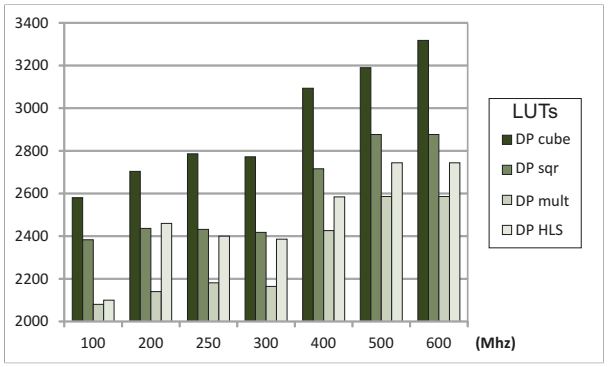
(a) Latency in cycles.



(b) Number of DSP blocks.

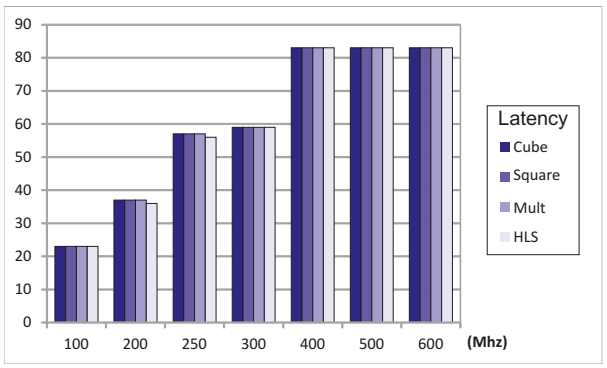


(c) Number of flip-flops.

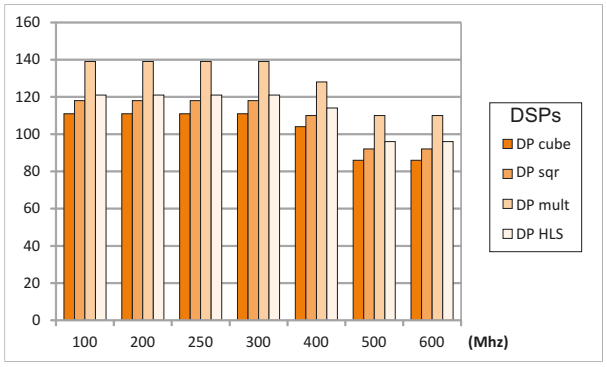


(d) Number of LUTs.

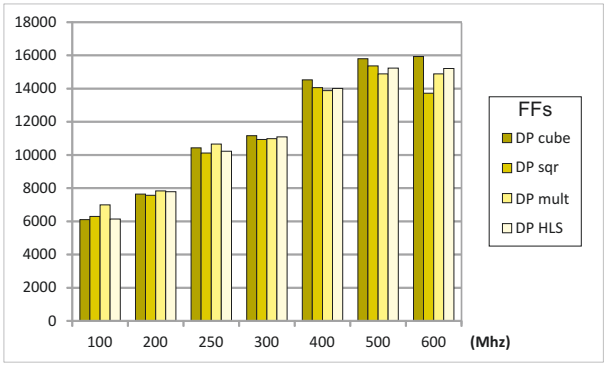
Fig. 11. Implementation results at different clock frequencies for double precision Painlevé's second equation computing the cube in four different ways: Optimized cube; optimized square plus normal multiplication; only normal multiplications; and HLS synthesis.



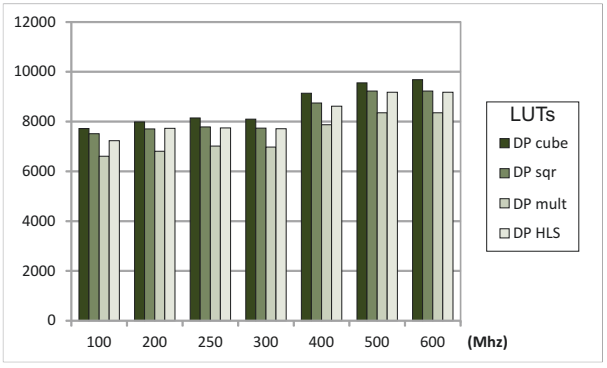
(a) Latency in cycles. Differences are minimal.



(b) Number of DSP blocks.



(c) Number of flip-flops.



(d) Number of LUTs.

Fig. 12. Implementation results at different clock frequencies for double precision Hodgkin-Huxley's potential equation computing the cube in four different ways: Optimized cube; optimized square plus normal multiplication; only normal multiplications; and HLS synthesis.

TABLE VI
MEAN SQUARED ERROR WITH RESPECT TO *exact* CUBE COMPUTATION OF THE MANTISSA, MEASURED IN *ulps*. RESULTS ARE GIVEN FOR THIS WORK (*cube*) AND MICROPROCESSOR *proc*. THE *exact* RESULTS USED AS A REFERENCE HAVE BEEN OBTAINED BOTH TRUNCATING (*Tr*) AND ROUNDING (*Rd*).

	Single precision				**Double precision**			
	Xilinx		**Intel**		**Xilinx**		**Intel**	
	Tr	**Rd**	**Tr**	**Rd**	**Tr**	**Rd**	**Tr**	**Rd**
cube	0.01	0.50	0.07	0.57	0.22	0.66	0.24	0.74
proc	0.50	0.26	0.51	0.25	0.51	0.26	0.51	0.26

section by means of simulation. Thus, the proposed architectures have been simulated in C and VHDL. The simulations in C are significantly faster than those using VHDL and allowed a greater coverage of inputs. The results (*cube* from now on) have been compared at binary level with those produced by an Intel Core i5 microprocessor (*proc* from now on) and, moreover, the cube of the mantissa has also been computed using extended 72 bit precision (*exact*).

For single precision, all possible mantissa values have been tested for Xilinx and Intel, and the maximum difference between *cube* and *exact* is 1 *ulp*. The mean squared error (MSE) with respect to *exact* is shown in Table VI. Actually, 2 different errors are given, depending on whether the value of *exact* is truncated to 24 bits (Tr), or rounded (Rd). A quick analysis shows that *proc* overestimates the result because rounding is applied twice to compute the cube, but *cube* underestimates the results by never rounding up. Overall, the error is small and implementation is simplified by avoiding rounding and re-normalization.

For double precision, testing 2^{52} mantissas would require thousands of computer hours. Instead, 2^{36} values are tested for Xilinx and Intel. For Xilinx, the mantissa is divided in 3 parts, a, b and c. For each of them, 2^{12} values are tried, covering all the combinations of the 6 *msb* and the 6 *lsb*. A random value is then assigned to the bits in between (different for each a, b and c). In this way, a wide sample of values are tested, covering all the combinations of large and small values of the 3 terms. For Intel, the mantissa is divided in 2 terms, a and b, so the same strategy is followed, only that 2^{18} values are tried for each one, covering all the combinations of the 9 *msb* and the 9 *lsb*. The results are also shown in Table VI and, again, the error is lower than 1 *ulp* in all cases, with *proc* tending to overestimate and *cube* tending to underestimate the result. The source code for assessing the error, plus the VHDL and HLS code to implement the cube function can be found at [21].

VI. CONCLUSION

Specialized arithmetic units allow for faster and more efficient calculations in custom computing machines. For computing the cube, architectures for single and double precision for the two main FPGA manufacturers have been proposed. Compared to using floating point multiplications, even if squaring is optimized, implementation cost is reduced, particularly the number of DSPs, which is critical. Latency is also reduced, as fewer operations are implemented to obtain the same result. The proposed architectures can also be pipelined and reach high clock speeds. All those benefits are achieved at the cost of a minimal loss in precision. Error is never greater than 1 *ulp*, and average error is comparable with that achieved by a standard microprocessor. Compared with a previous architecture that uses quadratic approximation to implement several powers only for single precision, this proposal has very similar cost and performance, but it avoids using RAM to store the coefficient tables. Finally, combining optimized cube and squaring architectures allows to reduce the cost of implementing higher powers, if needed.

ACKNOWLEDGMENT

This work was supported by the Ministry of Science and Innovation of Spain (PID2019-104184RB-I00 / AEI / 10.13039/501100011033), and by Xunta de Galicia and FEDER funds of the EU (Centro de Investigación de Galicia accreditation 2019–2022, ref. ED431G 2019/01; Consolidation Program of Competitive Reference Groups, ref. ED431C 2021/30).

REFERENCES

[1] M. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.
[2] F. de Dinechin, M. Istoan, and G. Sergent, "Fixed-point trigonometric functions on FPGAs," *SIGARCH Comput. Archit. News*, vol. 41, no. 5, p. 83–88, jun 2014. [Online]. Available: https://doi.org/10.1145/2641361.2641375
[3] J. Piñeiro, J. Bruguera, and J. Muller, "Faithful powering computation using table look-up and a fused accumulation tree," in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, 2001, pp. 40–47.
[4] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952.
[5] M. Brennan, I. Kovacic, A. Carrella, and T. Waters, "On the jump-up and jump-down frequencies of the Duffing oscillator," *Journal of Sound and Vibration*, vol. 318, pp. 1250–1261, 12 2008.
[6] *Solitons, nonlinear evolution equations and inverse scattering*. Cambridge University Press, 1991.
[7] G. Quaranta, G. Monti, and G. Marano, "Parameters identification of Van der Pol–Duffing oscillators via particle swarm optimization and differential evolution," *Mechanical Systems and Signal Processing*, vol. 24, pp. 2076–2095, 10 2010.
[8] N. Khan, S. Ahmad, O. Razzaq, and M. Ayaz, "Rational approximation with cuckoo search algorithm for multifarious Painlevé type differential equations," *Ain Shams Engineering Journal*, vol. 20, pp. 179–190, 03 2020.
[9] R. R. Osorio, "Pipelined FPGA implementation of numerical integration of the Hodgkin-Huxley model," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 07 2016, pp. 202–206.
[10] D. A. Buell and K. L. Pocek, "Custom computing machines: An introduction," *The Journal of Supercomputing*, 1995.
[11] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.
[12] "FloPoCo project website," http://www.flopoco.org, 2022.
[13] "Xilinx Vivado," http://www.xilinx.com, 2022.
[14] S. Xu, S. Fahmy, and I. McLoughlin, "Efficient large integer squarers on FPGA," in *Proceedings - 21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2013*, 04 2013, pp. 198–201.
[15] M. Kumm and P. Zipf, "Pipelined compressor tree optimization using integer linear programming," in *Conference Digest - 24th International Conference on Field Programmable Logic and Applications, FPL 2014*, 09 2014.
[16] Y. Yuan, L. Tu, K. Huang, X. Zhang, T. Zhang, D. Chen, and Z. Wang, "Area optimized synthesis of compressor trees on Xilinx FPGAs using generalized parallel counters," *IEEE Access*, vol. PP, pp. 1–1, 09 2019.

[17] J. Piñeiro, J. Bruguera, and J. Muller, "FPGA implementation of a faithful polynomial approximation for powering function computation," in *Proceedings Euromicro Symposium on Digital Systems Design*, 2001, pp. 262–269.
[18] J.-A. Piñeiro, S. Oberman, J.-M. Muller, and J. Bruguera, "High-speed function approximation using a minimax quadratic interpolator," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 304–318, 2005.
[19] J.-A. Piñeiro and J. D. Bruguera, "High-speed double-precision computation of reciprocal, division, square root and inverse square root," *IEEE Trans. Computers*, vol. 51, pp. 1377–1388, 2002.
[20] M. Durán-Camejo, "The cubic and quartic equations," 11 2019. [Online]. Available: https://www.researchgate.net/publication/337482063_The_cubic_and_quartic_equations
[21] R. Osorio, "Source code to implement and assess the cube function." https://github.com/RobertoUDC/CubeFPGA, 2022.

Roberto R. Osorio got his Ph.D. in Physics in 1999 at the University of Santiago de Compostela, Spain. In 2000, he joined IMEC v.z.w., where he contributed to MPEG-21. From 2003 he was a researcher within the Ramón y Cajal program, and associate professor from 2008. In 2010 he joined the University of A Coruña. His main research interests are in the areas of application specific circuits, digital arithmetic, and image and video coding. His homepage is http://gac.udc.es/~roberto.osorio.