Pipelined FPGA Implementation of Numerical Integration of the Hodgkin-Huxley Model

Roberto R. Osorio
University of A Coruña
Dept. Electronics and Systems
e-mail: roberto.osorio@udc.es

***Abstract*—The Hodgkin-Huxley model describes the initiation and propagation of action potential in neurons' axons. The model consists of a set of nonlinear differential equations that can be solved using numerical methods for a given choice of parameters. As the equations reflect physiological processes, the value of those parameters are subject to great variability. Therefore, numerical integration is often combined with differential evolution methods in order to find which set of parameters minimizes some fitness function. As modern FPGAs are large enough to implement complex functions using double-precision floating-point arithmetic, intensive scientific computations may be carried out showing competitive performance and cost. In this work, we present a pipelined architecture for performing the 4th order Runge-Kutta integration of the equations of the Hodgkin-Huxley model, introducing convenient implementations of complex mathematical functions.**

I. Introduction

FPGAs have emerged as an interesting platform for high-performance computing (HPC) thanks to the potential of using thousands of computational resources and memories in a concurrent manner. This potential is often spoiled, however, due to the overhead of communicating with a host computer or external memory.

The most promising applications are those that work on a reduced set of data by repeatedly making concurrent computations over them. Optimization using evolutionary algorithms is a good example of a HPC application that performs a huge amount of operations over a reduced data set with little control and data dependencies. In this paper we consider the implementation of one those problems: the Hodgkin-Huxley model in the context of differential evolution optimization.

Dating form 1952, the breakthrough work [1] by Alan L. Hodgkin and Andrew Huxley proposes a mathematical model that, using a set of nonlinear differential equations, describes the behavior of electrical potentials in neurons as a function of time. The model has been subject to some improvements since them, but it is the seminal work on the field and earned their authors the Nobel Prize in Physiology in 1963.

The differential equations in the model must be solved using numerical integration. The Hodgkin-Huxley model may be integrated using the 4^{th} order Runge-Kutta integration method, which makes it an approachable problem for an FPGA implementation. The model includes a set of empirical parameters that may change according to the specific case (the original paper focused on a giant squid axon). The aim of the optimization is to find the best choice of those empirical parameters. Therefore, the integration produces a series of results over time that should fit to a set of expected ones.

Differential evolution [2] is a well known optimization technique that proposes solutions to a given problem by hybridizing a set of existing ones. Those new solutions that outperform the initial ones are kept for the next generation. The ones that not, are discarded. Differential evolution may require thousands or millions of iterations/generations, implementing complex models such as the Hodgkin-Huxley one.

This paper is structured as follows. Section II describes the equations of the Hodgkin-Huxley model. Section III discusses the architecture and components. In Section IV the performance and implementation cost is discussed. Finally, the conclusions are presented.

II. The Hodgkin-Huxley model

The model describes the electrical potentials in a membrane as a function of the capacitance of the membrane and the currents that pass across it. Those currents are due to the interchange of Potassium and Sodium ions and also leakages:

$$\frac{dv}{dt} = \frac{1}{C_m} \cdot [I - g_{Na} m^3 h (v - E_{Na}) - g_K n^4 (v - E_K) - g_L (v - E_L)] \quad (1)$$

The values of n, m and h must be integrated from the differential equations:

$$\frac{dn}{dt} = \alpha_n(v)(1-n) - \beta_n(v) n \quad (2)$$

$$\frac{dm}{dt} = \alpha_m(v)(1-m) - \beta_m(v) m \quad (3)$$

$$\frac{dh}{dt} = \alpha_h(v)(1-h) - \beta_h(v) h \quad (4)$$

The values of $\alpha(v)$ and $\beta(v)$ are obtained as shown in Equations 5 to 10. The original equations are shown on the left. On the right, the same equation are simplified assuming that v has been previously divided by 10. We will make use of the simplified form, which only requires scaling the values

This work has been supported by the Ministry of Economy and Competitiviness of Spain (project TIN2013-42148-P) and by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Research Units (Ref. GRC2013/055), both co-funded by FEDER funds of the EU.

of g_{Na}, E_{Na}, g_K, E_K, g_L and E_L in Equation 1 to produce the same results.

$$\alpha_n(v) = \frac{0.01(v+10)}{\exp\frac{v+10}{10}-1} \quad \rightarrow \quad \frac{0.1(v+1)}{\exp(v+1)-1} \tag{5}$$

$$\beta_n(v) = 0.125\exp(\frac{v}{80}) \quad \rightarrow \quad 0.125\exp(\frac{v}{8}) \tag{6}$$

$$\alpha_m(v) = \frac{0.1(v+25)}{\exp\frac{v+25}{10}-1} \quad \rightarrow \quad \frac{v+2.5}{\exp(v+2.5)-1} \tag{7}$$

$$\beta_m(v) = 4\exp(\frac{v}{18}) \quad \rightarrow \quad 4\exp(\frac{v}{1.8}) \tag{8}$$

$$\alpha_h(v) = 0.07\exp(\frac{v}{20}) \quad \rightarrow \quad 0.07\exp(\frac{v}{2}) \tag{9}$$

$$\beta_h(v) = \frac{1}{\exp\frac{v+30}{10}+1} \quad \rightarrow \quad \frac{1}{\exp(v+3)+1} \tag{10}$$

A. Numerical integration

The Hodgkin-Huxley model can be integrated with accuracy using the 4^{th} order Runge-Kutta method, which will be implemented as follows:

- An initial value is assigned to the voltage, v
- Equations 5 to 10 are computed. Then, initial values are given to n, m and h as:

$$\begin{aligned} n &= \alpha_n(v)/(\alpha_n(v)+\beta_n(v)) \\ m &= \alpha_m(v)/(\alpha_m(v)+\beta_m(v)) \\ h &= \alpha_h(v)/(\alpha_h(v)+\beta_h(v)) \end{aligned} \tag{11}$$

- A loop starts that goes through the integration time interval in a number of iterations
- In each step, the 4^{th} order Runge-Kutta is implemented following these four steps:
  - Use Equations 1 to 4 to obtain temporal values $v1$, $n1$, $m1$ and $h1$
  - Apply Equations 1 to 4 using $v + v1/2$, $n + n1/2$, $m + m1/2$ and $h + h1/2$, obtaining $v2$, $n2$, $m2$ and $h2$
  - Apply Equations 1 to 4 using $v + v2/2$, $n + n2/2$, $m + m2/2$ and $h + h2/2$, obtaining $v3$, $n3$, $m3$ and $h3$
  - Apply Equations 1 to 4 using $v+v3$, $n+n3$, $m+m3$ and $h + h3$, obtaining $v4$, $n4$, $m4$ and $h4$
- Then, new values of v, n, m and h are obtained for the next iteration of the loop as (substitute x by v, n, m and h):

$$x' = x + (x1 + 2 \cdot x2 + 2 \cdot x3 + x4)/6 \tag{12}$$

- The new values are stored and used later to evaluate the fitness function

III. Architecture

The exploration of the architecture starts with an assessment of the cost of implementing the core of the model, which corresponds to Equations 1 to 10. Double-precision floating-point arithmetic will be used, which consumes a significant amount of FPGA resources. A preliminary analysis of the problem suggests that the availability of multipliers would be the main constraint.

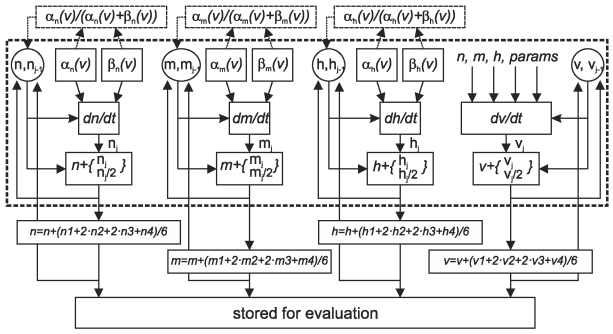

Fig. 1. Folded architecture for 4th order Runge-Kutta integration

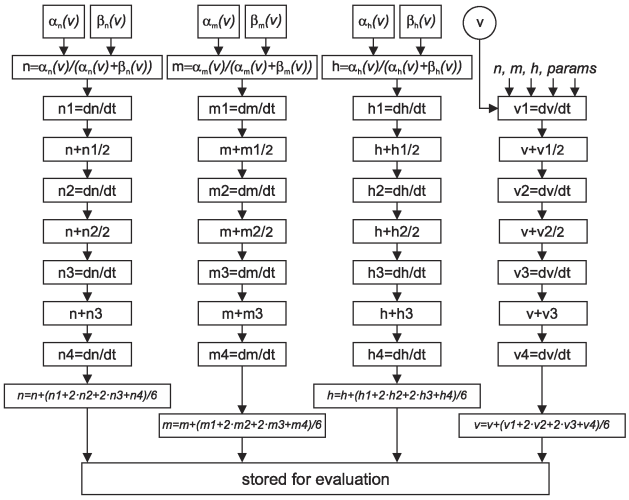

Fig. 2. Unfolded architecture

The kernel, inside the dotted box, is repeated 4 times to implement the numerical integration. Previously, $\alpha_x(v)$ and $\beta_x(v)$ are evaluated once to compute the initial values of n, m and h as shown in Equation 11 ($\alpha_x(v)/(\alpha_x(v)+\beta_x(v))$. Then, dn, dm, dh and dv are computed as in Equations 1 to 4; and the estimations of n, m, h and v are updated. After the 4 steps, new values of n, m, h and v are obtained applying Equation 12. Those values will start a new iteration and will also be stored to evaluate the fitness of the set of parameters.

In Figure 2, a fully unfolded architecture is depicted. The depth of the pipeline is 4 times the folded one, potentially allowing higher performance at a higher cost. In Section IV, both options will be discussed.

A. Implementation of floating-point arithmetic

Implementing basic arithmetic functions on an FPGA using double-precision floating-point takes a considerable amount of

resources. This work has been carried out using Xilinx FPGAs and software, therefore, some of the design decisions have been taken based on the DSP and memory blocks available on those devices.

The FloPoCo project [3] allows generating customized cores for adders, multipliers, and some elementary functions. Some interesting ideas in FloPoCo include using flags for signaling zero, infinites and NaN. Also, the exponent field is extended 1 bit, making denormal numbers just normal ones, and greatly simplifying the implementation. In this work, FloPoCo is not used, but we deal with denormal numbers in the same way.

Additions and multiplications have been designed to have a 3 cycle latency, with a target speed of 200 MHz in Virtex-7 devices. The remaining functional units were clocked at the same speed. Whereas higher clock frequencies can be achieved through deeper pipelining, that would complicate the design of the exponential unit, so it will be considered as future work.

B. Additions

The first stage of addition pre-computes the exponent and pre-aligns the operands. The second stage implements the integer addition. The third one normalizes the result and updates the exponent and the sign of the result.

Adding or subtracting constants may be optimized. In Equations 5 to 10, the maximum value of the voltage is lower than 2. Therefore, adding 1, 2.5 or 3 involves a maximum of 3 bits. The remaining ones are just appended. The post alignment of the result is also restricted to a few bits. Other optimizations are also possible but, at the current state of the research, these must be applied one by one. Automatic tools for synthesizing optimized adders would be necessary to fully exploit those opportunities.

C. Multiplications

For multiplication, the first stage implements partial products and pre-computes the exponent. In latest Xilinx devices, DSP blocks allow up to 24 $\times$ 17 bit unsigned products. Thus, 9 DSP block may be necessary to implement a 53 $\times$ 53 bit product. However, as shown in [4], this can be shorten down to 8 DSPs and additional LUTs. The second stage adds up the partial products, and the third one aligns the mantissa and corrects the exponent.

In Figure 3.a, a 58 $\times$ 58 bit product is implemented as eight 24 $\times$ 17 bit products plus a 10 $\times$ 10 bit one. The useful 53 $\times$ 53 are highlighted with a bold line. The partial products implemented using LUTs are colored. In Figure 3.b the same product is shown in the more familiar staggered way.

Squaring is a particular case of multiplication in which some of the partial products appear twice, so they can be computed just once. As explained in [4], squaring in double precision takes 6 DSPs instead of 8 or 9, a significant saving.

Finally, cost may be reduced if a small error is accepted, as a 53 $\times$ 53 bit product renders a 106 bit result that must be rounded down to 53 bits. The maximum error is less than 1 ulp (unit in the last place). If a slightly larger error can be assumed, some of the less significant partial products can be skipped. Figure 3.c shows a partitioning using only 6 DSPs and additional LUTs. In Figure 3.d, the missing partial products become obvious, but the error is small. The gray bar at the bottom of the figure marks the desired 53 bits, and the point on its right, the maximum error introduced by the truncation. Therefore, this possibility will be included in our analysis. Similarly, squaring can be performed with only 5 DSPs with negligible error. With respect to multiplying by constants, powers of 2 are optimized, as well as repeating fractions.

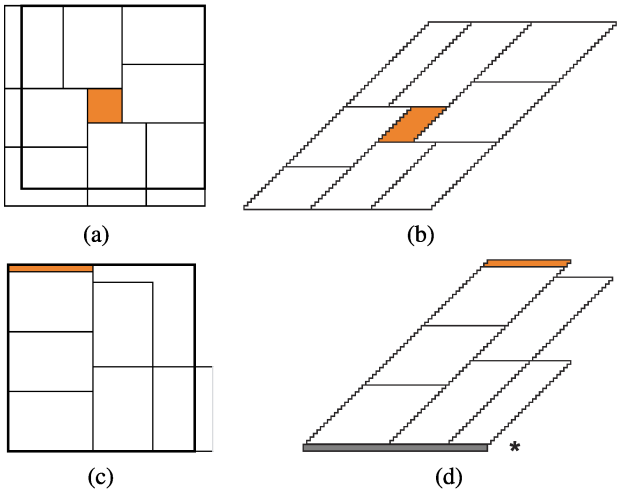

Fig. 3. Partitioning of 53 $\times$ 53 bit multipliers using 25 $\times$ 18 DSPs and logic. Exact implementations in (a) and (b). Approximate versions in (c) and (d).

D. Exponential function

Exponentiation is a costly function which must be implemented several times in the model. An implementation is proposed based on the range of the arguments and the availability of RAM blocks for implementing look-up tables.

Given the range of values of v in Equations 5 to 10, the range of arguments for the exponential function is approximately [4, -11]. Therefore, the argument is translated to a 5-bit signed integer plus a fractional part. The argument is broken down into three 9-bit long pieces (a, b and c) plus a residue (d). Then, exponential is computed as shown in Equation 13.

$$\begin{aligned} x &= a_4 \cdots a_0.a_{-1} \cdots a_{-4} b_{-5} \cdots b_{-13} c_{-14} \cdots \\ &\quad c_{-22} d_{-23} \cdots d_{-52} \\ A &= a_4 \cdots a_0.a_{-1} \cdots a_{-4} \\ B &= 0.00 \cdots b_{-5} \cdots b_{-13} \\ C &= 0.00 \cdots c_{-14} \cdots c_{-22} \\ D &= 0.00 \cdots d_{-23} \cdots d_{-52} \\ e^x &= e^A \times e^B \times e^C \times D \end{aligned} \tag{13}$$

The value of e^A is obtained from a look-up table using $a_4 \cdots a_{-4}$. We proceed similarly for B and C. For D, a truncated second order Taylor approximation may be taken, rendering $1+D+D^2/2$, in which D^2 is computed using LUTs

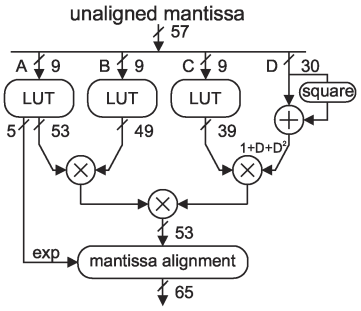

Fig. 4. Exponential calculation

as a 7×7 product. Figure 4 depicts how the exponential is computed.

This implementation requires three 53×53 bits multiplications, requiring $3 \times 9 = 27$ DSP blocks. However, that number can be reduced by carefully adjusting the precision of each product. For $e^A \times e^B$, 8 instead of 9 DSPs could be enough assuming a small error. More important, it can be seen that only 39 and 30 bits are not zero in the fractional parts of e^C and D, respectively. Therefore, if $e^C = 1 + \zeta \cdot 2^{-52}$ and $D = 1 + \delta \cdot 2^{-52}$, the latter product can be implemented as $1 + (\zeta + \delta) \cdot 2^{-52} + \zeta \cdot \delta \cdot 2^{-104}$. Calculating $\zeta \cdot \delta$ takes just 1 DSP. Similarly, the third product between $e^A \times e^B$ and $e^C \times D$ can be implemented as a 53×40 bit product, which requires 4 DSPs and some logic, followed by an addition. Eventually, only 15 DSPs are needed, and the total latency is 7 cycles.

The 3 look-up tables will take 1 BRAM36K each. However, RAM blocks are double-ported, so they can be shared between 2 modules. Moreover, the RAM blocks can be clocked at double speed, doubling also the sharing. Eventually, the cost of the look-up tables is diluted.

E. Division

Division is here implemented using the Goldschmidt algorithm, based on architecture proposed in [5], that implements a second degree polynomial to approximate the reciprocal with great accuracy, and converging in just one iteration. An FPGA implementation of that architecture is explained in detail in [6], which we have adopted. The cost of the architecture is 11 DSP blocks and 1 BRAM36K. Again, the RAM block can be shared between 2 or 4 dividers. The latency is 8 cycles.

IV. Results

A. Cost assessment

Table I shows the resources required to implement each of the principal arithmetic functions in the architecture.

Table II shows the aggregated cost of implementing the functional units in Equations 1 to 12. The unfolded version implements 4 copies of Equations 1 to 10, plus a single instance of Equations 11 and 12, plus an additional instance of Equations 5 to 10 for the initial values of n, m and h. The rows labeled as x_{i+1} correspond to implementing v_i, n_i, m_i and h_i as explained in Section II-A. Block RAM are shared, so the cost is expressed in fourths.

TABLE I
Resources for the principal functions

	LUT	FF	DSP	BRAM
addition	681	334		
multiplication	524	322	8 (6)	
exponential	1449	998	15	3
division	1853	1094	11	1
square	313	200	6 (5)	

TABLE II
Resources used by the folded and unfolded architectures

Equation	LUT	FF	DSP	BRAM
Folded				
1	11K	6K	85	1/4
2-4	7.5K	4K	48	
5-10	20K	12K	123	3/4 + 6/4
11	7.6K	4.2K	33	3/4
12	12K	6K	16	
$\mathbf{x_{i+1}}$	2.7K	1.4K		
Total	61K	34K	305	4
Unfolded				
1	44K	24K	340	4
2-4	30K	16K	192	0
5-10	81K	48K	492	3+6
$\mathbf{x_{i+1}}$	11K	5.6K		
Total	204K	115K	1196	17

The proposed architecture must be integrated into a differential evolution (DE) engine [2], which requires additional tasks such as hybridization and range checking of each parameter. Dealing with up to 8 parameters, additional 39K LUTs, 20K FFs and 112 DSPs are required. These amounts are largely inferior to the ones listed in Table II. Hence, the whole system can be implemented in a modern FPGA, not only the numerical integration described in this work. We have confirmed that nearly any device in the Virtex 7 series can implement these architectures.

B. Latency and throughput

Several iterations of the Runge-Kutta integration must be run to obtain results in a time interval. As each iteration needs the results from the previous one, there is a strong data dependency. The depth of the pipeline, and the number of different sets of data making use of it, have a great influence in the performance.

For the folded architecture in Figure 1, 32 cycles are needed for obtaining the initial values of n, m and h; 41 for the core of the iteration; and 12 cycles for equation 12. Hence, the core of the iteration can support up to 41 computations at the same time. For the unfolded architecture in Figure 2, the main difference is that the core of the iteration takes 141 cycles (the fourth step in the integration is slightly shorter), allowing a larger number of concurrent computations.

Using Hodgkin-Huxley model with DE will require dealing with 30 to 80 solutions. In Figure 5, the number for cycles needed to compute the Runge-Kutta integration is shown for the 2 architectures, obtaining 10 or 20 points for each solution. The number of solutions goes from 30 to 160 in the X-axis. The folded architecture is almost as fast as the unfolded one

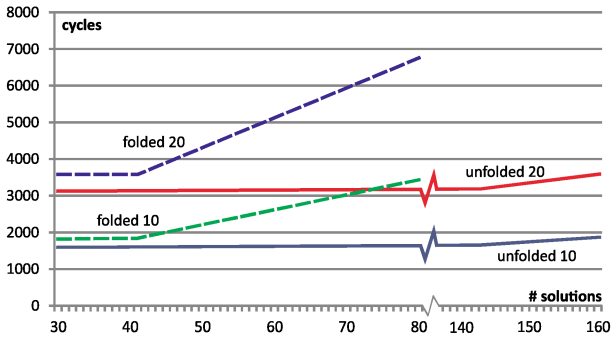

Fig. 5. Computing cycles vs the number of solutions

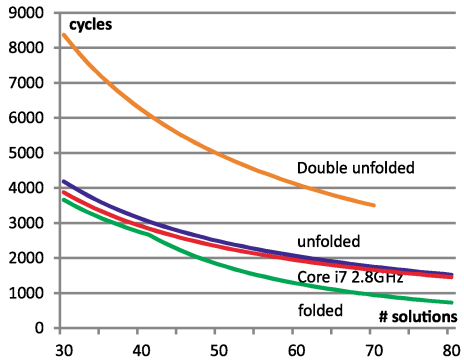

Fig. 6. Number of generations per second vs the number of solutions

when the number of solutions is lower than the latency of the pipeline, 41 cycles. Then, the number of cycles increases and the unfolded version is faster. The latency of the unfolded architecture is nearly flat until the number of solutions is larger than 141, the number of stages of the pipeline.

C. Performance

Integrating this work within a DE system will increase the latency in less than 30 cycles. Figure 6 shows the number of generations (complete sets of solutions) that the architectures are able to process per second. For clarity sake, we just show results for 10 samples. Together with the folded and unfolded architectures, 2 new sets of results are shown. First, results obtained running a software version on an Intel Core i7 processor clocked at 2.8 GHz (single thread). Secondly, the unfolded version running 2 sets of solutions in a time-multiplexed manner.

The unfolded architecture is faster than the folded one, up to 200% when the pipeline is full. To fully exploit the unfolded pipeline, this must be shared by 2 DE engines, as shown in the graph, as long as that the aggregated number of solutions does not exceed the length of the pipeline. Finally, the software version shows excellent performance in this application, considering the large number of exponentials and quotients that are computed.

For the Runge-Kutta integration of the Hodgkin-Huxley model, the performance of the proposed architectures is competitive with respect to software implementations. However, a larger performance difference would be desirable in order to justify the design effort. We expect to achieve those results focusing in more complex problems.

Some recent works on hardware acceleration of differential evolution are [7] using FPGAs, and [8] with GPUs. Those works are, however, not focused on real applications, but on well-known benchmarks such as Rastrign's and Schwefel's.

We extract 3 lessons from this work. First, current FPGAs are large enough to map complex architectures on them. Secondly, knowing the application allows us to develop specific implementations of complex functions, such as the exponential, with lower cost and latency than a general purpose implementation. In this specific case, we refer to the restriction to a given range of input values and the sharing of look-up tables. Thirdly, complex computational model imply large pipelines, which can be shared between several instances of the optimization algorithm running concurrently. In this sense, we will consider, as a future work, to segment adders and multipliers to a higher degree, obtaining deeper pipelines and higher clock frequencies.

V. Conclusions

In this work, two architectures for the 4^{th} order Runge-Kutta integration of the Hodgkin-Huxley model have been proposed for FPGA implementation, meant to be integrated with a DE optimization engine. The folded and unfolded architectures can be mapped on modern mid and high-end devices. Some optimizations have been proposed, such as adapting the precision of the multipliers and sharing the memories that implement the look-up tables in exponentials and divisions. As future work, we consider to segment the functional unit to a higher degree in order to increase the clock frequency, and sharing the pipeline between several optimization engines. Finally, we consider that other, more complex, optimization problems may be implemented in larger FPGAs with a considerable advantage over CPUs.

References

[1] 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.

[2] R. Storn and K. Price, "Differential evolution. A simple and efficient heuristic for global optimization over continuous spaces," *J. of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[3] 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.

[4] ——, "Large multipliers with fewer DSP blocks," in *19th International Conference on Field Programmable Logic and Applications, FPL 2009*, 2009, pp. 250–255.

[5] 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.

[6] B. Liebig and A. Koch, "Low-latency double-precision floating-point division for FPGAs," in *FPT*, 2014.

[7] Y. Jewajinda, "Parallel hardware architecture and fpga implementation of a differential evolution algorithm," in *TENCON*, 2014.

[8] W. Zhu, "Massively parallel differential evolution—pattern search optimization with graphics hardware acceleration: an investigation on bound constrained optimization problems," *Journal of Global Optimization*, vol. 50, no. 3, pp. 417–437, 2010.