FPGA Implementation of Diffusive Realization for a Distributed Control Operator

G. Goavec-Merou*, Y. Yakoubi†, R. Couturier‡, M. Lenczner*, J.M. Friedt* and F. Yang§
*FEMTO-ST, Time-Frequency Department, 26, chemin de l'Epitaphe, 25030 Besançon, France
†Laboratoire Jacques-Louis Lions, Pierre et Marie Curie University, 75252 Paris Cedex 05, France
‡University of Franche-Comte, LIFC, IUT Belfort-Montbéliard, rue Engel Gros, 90000 Belfort, France
§Le2i, Bourgogne University, Mirande University, 21000 Dijon, France

***Abstract*—We focus on the question of real-time computation for optimal distributed filtering or control applicable to MEMS Arrays. We present an algorithm for the realization of a linear operator solution to a functional equation through its application to a Lyapunov operatorial equation associated to the heat equation in one dimension. It is based on the diffusive realization, and turns to be well suited for fined grained parallel computer architecture as Field Programmable Gate Arrays (FPGA). An effective FPGA implementation has been successfully carried out. Here, we report the main implementation steps and the final measured performances.**

***Keywords*-Distributed Control; FPGA; Diffusive Realization; Semi-decentralized Control; Fine Grained Parallel Computer Architectures.**

I. Introduction

Our concern when developing the method presented in this paper relates to embedded intensive real time computation based on fine grained parallel computer architectures as Field-Programmable Gate Arrays (FPGA). Here we address the general problem of realization of a linear operator $u \mapsto z = Pu$ in infinite dimensional spaces. Our method is presented and illustrated on the example of the operator P, solution to the Lyapunov equation,

$$\frac{d^2}{dx^2}Pu + P\frac{d^2}{dx^2}u = Qu \tag{1}$$

in $\omega = (0,1)$ for all u vanishing at the boundary of ω, where Q is another linear operator. It is issued from optimal filtering or control theory of the heat equation

$$\frac{\partial T}{\partial t} - \frac{\partial^2 T}{\partial x^2} = q \text{ in } \omega.$$

The method is in the same time fast and suitable for implementation on *semi-decentralized* architectures where communications are mainly between neighbor unit cells. Its formulation relates to an interesting technique, the *diffusive representation*, applicable to causal operator realizations which is well developed in the context of time operators. One of the main recognized advantages of this approach is its low computational cost, see the papers of G. Montseny *et al*., e.g. [2], and of D. Matignon *et al*., e.g. [5], for representations of various pseudodifferential operators and for their approximation. Those of C. Lubich et al., e.g. [6], apply a similar idea to convolution operators and they develop optimized numerical methods.

In this paper, we briefly present the *diffusive representation* of the solution P to (1) together with algorithms for their approximation implementable in *semi-decentralized* architectures. We refer the interested reader to the general theory shortly introduced in [3] and detailed in [4]. Then, the core of the paper is devoted to FPGA implementation using integer computation. We start by a re-scaling of the algorithms guarantying that all intermediary and final results be in a prescribed interval. Then we report the results of our study on the effect of quantization. After these preparation steps, the FPGA mapping is analyzed for three different possible schemes, namely: Iterative, Pipelined and Parallel. The Parallel solution has been implemented in the Spartan3A by Xilinx, and we provide detailed results regarding resource and time consumptions.

II. Diffusive Representation of P

We consider the kernel formulation of the operator P,

$$Pu(x) = \int_0^1 p(x,y)u(y)\ dy,$$

and the decomposition $z = z^+ + z^-$ into causal and anti-causal parts,

$$z^+ = \int_0^x p(x,y)u(y)\ dy \quad \text{and} \quad z^- = \int_x^1 p(x,y)u(y)\ dy.$$

The kernel p is the unique solution to the boundary value problem

$$\begin{aligned} -\Delta p = q \quad &\text{in the square } (0,1)\times(0,1), \\ \text{and } p = 0 \quad &\text{on the square boundary } (0,1)\times(0,1), \end{aligned} \tag{2}$$

where q is the kernel of Q. The realization of z^+ and of z^- may be formulated thanks to the diffusive representation, see [3] and [1], in the form

$$\begin{aligned} z^+(x) &= \int_{\mathbb{R}} \mu^+(x,\xi)\psi^+(x,\xi)\ d\xi \\ \text{and } z^-(x) &= \int_{\mathbb{R}} \mu^-(x,\xi)\psi^-(x,\xi)\ d\xi, \end{aligned} \tag{3}$$

where both ψ^+ and ψ^- store a part of the history of the input data u. They are respectively solution to the forward and backward ordinary differential equation in x,

$$\begin{aligned} \partial_x\psi^+(x,\xi) + \theta^+(\xi)\psi^+(x,\xi) &= u(x) \\ \text{with } \psi^+(0,\xi) &= 0, \end{aligned} \tag{4}$$

$$\begin{aligned} \text{and } \partial_x\psi^-(x,\xi) - \theta^-(\xi)\psi^-(x,\xi) &= u(x) \\ \text{with } \psi^-(1,\xi) &= 0, \end{aligned} \tag{5}$$

where ξ is a real parameter. We emphasis that they are independently of a specific operator P. Conversely, the coefficients μ^+ and μ^-, called *diffusive symbols*, depend on P but not on u. The functions $\xi \mapsto \theta^+(\xi)$ and $\theta^-(\xi)$ parameterize two closed paths in the complex plane, satisfying the cone condition, and enlacing the singularities of the Laplace transform $\mathcal{P}^\pm$ defined hereafter. The diffusive symbol derivation requires several steps. The two functions

$$y \mapsto p(x, x-y) \quad \text{and} \quad y \mapsto p(x, x+y), \tag{6}$$

corresponding to the causal part and the anti-causal parts of the impulse response, are analytically extended to $\mathbb{R}^+$. We assume that their Laplace transforms $\mathcal{P}^+$ and $\mathcal{P}^-$ are well-defined in $\mathbb{C}^+$, and

that they admit holomorphic extensions vanishing at infinity. The Diffusive symbols are then given by

$$\mu^{\pm}(x,\xi) = -\frac{\theta^{\pm\prime}(\xi)}{2i\pi}\mathcal{P}^{\pm}\left(x,\theta^{\pm}(\xi)\right). \tag{7}$$

III. Approximations and Algorithms

We introduce the variational formulations for p^+ and p^- and we approximate their solutions by using convenient Galerkin base. The paths $-\theta^{\pm}$ are chosen so that to enlace, by the right, the poles of Laplace transforms of the basis functions, and the approximations $\mu^{N\pm}$ of $\mu^{\pm}$ are then given by the counterpart of the formula (7) applied to the Laplace transform of the impulse response approximations. Once the diffusive symbols are available, a numerical approximation of the integrals in (3) must be derived. The diffusive realization $z(x) = z^+(x) + z^-(x)$ is evaluated at some points $(x_n)_{n=0,..,\mathcal{N}} \in (0,1)$, thanks to a trapezoidal rule by

$$z_n = h_\xi \sum_{k=-M}^{M} \mu_{n,k}^{N+}\psi_{n,k}^{+} + \mu_{n,k}^{N-}\psi_{n,k}^{-}, \tag{8}$$

for a $M \in \mathbb{N}^*$, where $h_\xi > 0$ is the step size (distance between two consecutive integration point), $\mu_{n,k}^{N}$ and $\psi_{n,k}$ are some discrete values of μ^N and ψ at the points x_n and ξ_k. Such quadrature rule is similar to this encountered in the inverse Laplace transform in the paper of J. A. C. Weideman and L. N. Trefethen [7]. These authors have optimized the parameters of two different classes of contours, typically a parabola

$$-\theta^{\pm}(\xi) = \theta_p\left(i\xi+1\right)^2 \quad \text{for } \xi \in \mathbb{R}, \tag{9}$$

and a hyperbola

$$-\theta^{\pm}(\xi) = \theta_h\left(1+\sin\left(i\xi-\alpha\right)\right) \quad \text{for } \xi \in \mathbb{R}, \tag{10}$$

for some positive real numbers θ_p, θ_h and α the hyperbola asymptotic angle. Our computation are based on the same optimized paths.

In the FPGA, the data are allocated to processing units according to a mapping from the set of space locations $\{x_n \quad | \quad n = 0,..,\mathcal{N}\}$ to the set of processing units. For real time computation, the diffusive symbols are computed in a preprocessing step, and only the computation of $\psi_{n,k}^{+}$ and of $\psi_{n,k}^{-}$, and then of z_n through (8), remain to be done at each time step. The summation (8) is clearly a local (in space) operation, and the resolution of the Cauchy problems (4, 5) requires operations between neighbors nodes only. Indeed, we use the forward and backward recurrence relations associated to a discretization method with piecewise constant interpolation of u,

$$\begin{aligned}
&\psi_{n+1,k}^{+} = \psi_{n,k}^{+}e^{-\theta^{+}(\xi_k)\left(x_{n+1}-x_n\right)} + \\
&\frac{e^{-\theta^{+}(\xi_k)\left(x_{n+1}-x_n\right)}-1}{-\theta^{+}(\xi_k)}u(x_n), \text{with } \psi_{0,k}^{+} = 0, \\
&\text{and } \psi_{n,k}^{-} = \psi_{n+1,k}^{-}e^{-\theta^{-}(\xi_k)\left(x_{n+1}-x_n\right)} - \\
&\frac{e^{-\theta^{-}(\xi_k)\left(x_{n+1}-x_n\right)}-1}{-\theta^{-}(\xi_k)}u(x_n), \text{with } \psi_{1,k}^{-} = 0,
\end{aligned} \tag{11}$$

easy to establish, see [4]. Clearly, the approximation $\psi_{n+1,k}^{+}$ (resp. $\psi_{n,k}^{-}$) at a node x_{n+1} (resp. x_n) is derived from the approximation $\psi_{n,k}^{+}$ (resp. $\psi_{n+1,k}^{-}$) at the previous node x_n (resp. next node x_{n+1}) for all k. Finally, we present the algorithms in a compact form, where $h = x_{n+1} - x_n$ is the spatial discretization step, $\alpha_k^{\pm} = e^{-\theta_k^{\pm}h}$, $\beta_k^{\pm} = \frac{\alpha_k^{\pm}-1}{-\theta_k^{\pm}}$, $\gamma_k^{\pm} = \frac{\alpha_k^{\pm}}{-\theta_k^{\pm}}$, and $\theta_k^{\pm} = \theta^{\pm}(\xi_k)$ for convenient discretization points ξ_k.

Algorithm 1 Diffusive Realization of $z^+(x)$

1: **Offline Computation** of diffusive symbol $\mu^{N+}(x,\xi)$
2: **Online Computation**
3: **for** $n = 0,...,\mathcal{N}$ **do**
4: **for** $k = -M,...,M$ **do**
5: $\psi_{n+1,k}^{+} = \alpha_k^{+}\psi_{n,k}^{+} + \beta_k^{+}u_n, \quad \psi_{0,k}^{+} = 0,$
6: **end for**
7: $z_{n+1}^{+} = h_\xi \sum_{k=-M}^{M} \mu_{n+1,k}^{N+}\left(\alpha_k^{+}\psi_{n,k}^{+} + \gamma_k^{+}u_n\right)$
8: **end for**

Algorithm 2 Diffusive Realization of $z^-(x)$

1: **Offline Computation** of diffusive symbol $\mu^{N-}(x,\xi)$
2: **Online Computation**
3: **for** $n = 0,...,\mathcal{N}$ **do**
4: **for** $k = -M,...,M$ **do**
5: $\psi_{n,k}^{-} = \alpha_k^{-}\psi_{n+1,k}^{-} - \beta_k^{-}u_n, \quad \psi_{\mathcal{N},k}^{-} = 0,$
6: **end for**
7: $z_n^{-} = h_\xi \sum_{k=-M}^{M} \mu_{n,k}^{N-}\left(\alpha_k^{-}\psi_{n+1,k}^{-} - \gamma_k^{-}u_n\right)$
8: **end for**

IV. Scaling and value encoding

In this section we explain how we proceed to prepare Algorithm 1 before its implementation in a FPGA. Since Algorithm 2 is very similar to that one we only considered the former.

With the MATLAB implementation of Algorithm 1, all values are encoded with floating point type. More precisely, the MATLAB code uses complex numbers encoded with floating points. By default, the FPGA uses only a bit vector for coding integers and the use of floating points requires many logical cells on the FPGA. So, for many kinds of FPGA, the use of floating is not recommended. In this work, in order to be efficient, one of our constraints was to compute only with integers. In order to scale the values of the variables involved in the computation, we started by analyzing the algorithm. Doing this, we observed that the following variables could be scaled: β, γ, μ and u. Since β and γ depend on α, we do not scale $alpha$ which is almost naturally scaled without doing anything. So, for γ, μ and u, we compute the maximum values, respectively called γ_{max}, μ_{max} and u_{max}. Then we compute the scaled variables, respectively called β_{scaled}, γ_{sclaed}, μ_{scaled} and u_{scaled} with $\beta_{scaled} = \beta/\gamma_{max}$, $\gamma_{scaled} = \gamma/\gamma_{max}$, $\mu_{scaled} = \mu/\mu_{max}$ and $u_{scaled} = u/u_{max}$. It should be noticed that as β and γ depend on α we chose to scale those variables by γ_{max} which is larger than β_{max}.

So in Algorithm 1, we can deduce that ψ (line 5) is automatically scaled since its computation involves scaled variables. Likewise, the computation of z (line 7) is scaled and involves the products of variables scaled by μ_{max}, γ_{max} and u_{max} so in order to obtain the final result, we need to multiply the result by $u_{max} \times \beta_{max} \times \mu_{max}$. Thus, $z = z_{scaled} \times u_{max} \times \beta_{max} \times \mu_{max}$

To be efficient on FGPA architecture, one must avoid, when possible, floating point computation. So, next we tried to convert floating point values into integer. Using integers encoded on 32 bits, results of the computation are very similar to the ones using floating points. Nevertheless, the smaller the number of bits used to code integer in the FPGA architecture is, the less numerous are the logical cells used, so the more parallelism capabilities we obtain.

That is why, next we made some experiments using MATLAB to determine the influence of the number of bits to code integer. During the computation some variables take values greater than one even if they were all scaled before the computation. So we had to measure the number of bits required to be added in order to take this into account. In practice, for the experiments we performed, only two bits were sufficient. Likewise, as the FPGA is only ableto manipulate vector of bits representing positive integer, we had to add one more bit for negative numbers. In Figure 1, we compare the results for the causal diffusive realization with floating points, with integers encoded on 16 bits and with the exact solution. The three curves seem very similar even with the computation using integers.

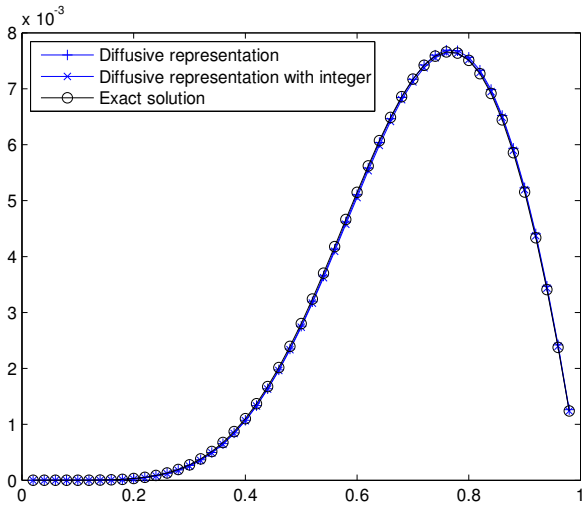

Fig. 1. Results obtained by the exact solution of the problem and by the two different approximations (one with floating points and the other with integers encoded on 16 bits).

In Table I, we can compare the maximum error and the relative error between different integer computations encoded with different numbers of bits and the exact solution. We can see that the best number of bits is 16 and that the second best is 14 instead of what we should have expected with 15.

Number of bits	Maximum error	Relative error
16	9.21e-5	1.05%
15	2.08e-4	2.51%
14	1.38e-4	1.51%
13	4.82e-4	6.79%
12	6.87e-4	9.23%
11	1.4e-3	18.69%

TABLE I
INFLUENCE OF THE NUMBER OF BITS ON THE ERROR COMPARED TO THE EXACT SOLUTION.

In Figure 2, we highlight the most significant errors due to the influence of the small number of bits used to encode integers with the diffusive representation. We can see the observation highlighted in Table I concerning the number of bits.

V. FPGA

FPGA unlike to CPU has no predefined functionalities. This allows to have a finer granularity to implement algorithms.

The FPGA used for this work is a Xilinx Spartan3A ([11]) with:

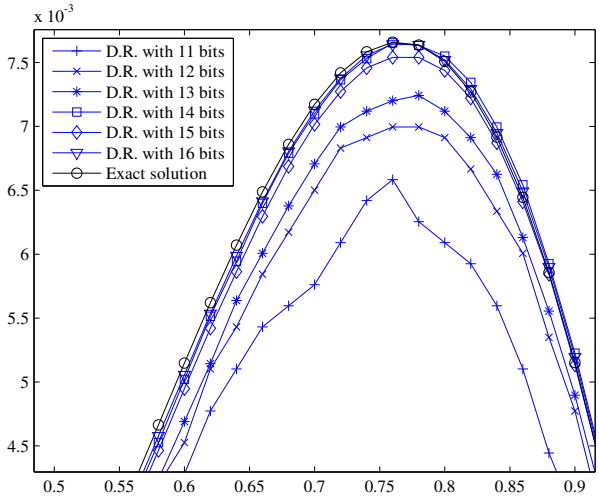

Fig. 2. Comparison of different Diffusive Representation (noted D.R.) with integers encoded with different numbers of bits with the exact solution. The figure is zoomed where the errors are the most significant.

- 200 k gates,
- 16 multipliers with 18 bits input and 36 bits output
- 16 RAM blocks of 16 kbits each,
- a 100 MHz clock.

Several implementations of a given algorithm are possible thanks to the flexibility of this type of component:

- a classical iterative implementation, as done in a general purpose computer. This solution is not efficient because for many treatments, waiting the last processing step to be completed before starting the next is needed, but the FPGA gate area use is low,
- a pipeline solution. With this strategy, the next treatment can be started on the next clock event. This type of implementation exhibits better global time performances since each new treatment only adds one clock period, but requires more gate area than the iterative technic since registers are needed to store temporary results.
- parallel processing. Many independants treatments could be made in the same time. This type of implementation has the advantage to reduce the global time and the critical time is equal to the longest treatment.

VI. POSSIBLE SOLUTIONS OF IMPLEMENTATION TO FPGA

In this section, we explain how we implemented 1 on an FPGA. The implementation choices are directly linked to constraints of the equations (variable dependencies) and FPGA capabilities.

Figure 3 illustrates the relationships between ψ, $u(x_n)$, and ξ_k.

As presented in Figure 3, for a same ξ, $\psi(x_{n+1}, \xi)$ is dependent on $\psi(x_n, \xi)$ value. It is not possible to start the next calculation step before having finished the previous one. However, the computation of ψ for different ξ are independent.

Figure 4 shows the relation between the contribution of $P(u_n)$: there are fewer constraints than ψ because it is possible to add a new value to an existing one or make every addition before to store the result.

The other aspect to be considered is the FPGA ressource availability. For example, if an equation needs to do 50 multiplications with a FPGA containing only 16 multiplier blocks, pipelining the computation to be impossible. The pipeline needs to have at least

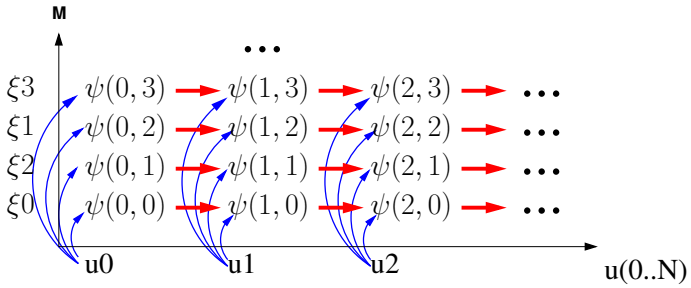

Fig. 3. Dependency for ψ computation

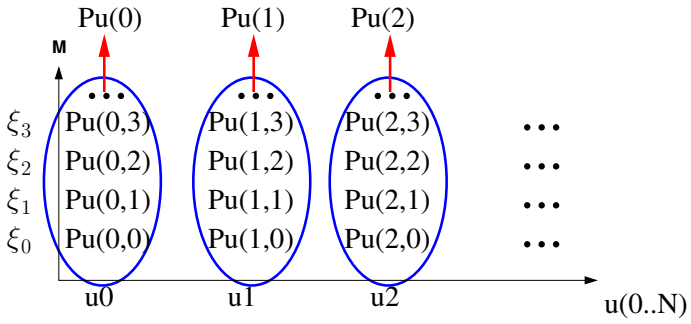

Fig. 4. Contribution for $P(u_n)$, with $P(n,k) = h_\xi \mu_{n,k} \psi_{n,k}$

as many multipliers as required multiplications. The same problem arises for memory management (RAM) and multiplier, a scarce resource on FPGA.

These constraints are important because they guide choices in terms of possible solutions, and the manner to implement them. For example, because the value of $\psi(x_n, \xi_k)$ is needed for start the computation of $\psi(x_{n+1}, \xi_k)$, it is not possible to use a pipeline solution in this direction.

In the sequel, we detail the three solutions above introduced. This is not exaust. Only possible solutions according to the constraints of equations and available ressources are presented.

A. Iterative

The first solution consists in iterating on ξ values. For each value of ξ, it consists in computing the ψ recurrence. This solution is economic in logic cells because data accesses are made sequentially, it is possible to use RAM blocks. But due to the relation between every ψ for a specific ξ_k, the total computation time is equal to the time needed to perform the computation for one value of ψ, multiplied by the number N of inputs u and the number of lines (iterations needed for the convergence M).

B. Pipeline

The second solution is to use a pipeline mechanism. To be able to use it, it is necessary to compute every $\psi(x_n, 0)$ values, and next to compute $\psi(x_n, 1)$, and so on. This solution is better than the first one because the next start of computation is made one clock period after the previous one. As in the first solution, it is possible to use block RAM because data are accessed sequentially. However, this solution has two drawbacks:

- this solution needs to have the same amount of multipliers than multiplications in equations,
- it needs Block RAM to store ψ values to make them available for the next ξ.

C. Parallel

The last solution consists in implementing the computation in parallel. Because there is no dependency between $\psi(x_n, 0)$ and another ψ for a different ξ, it is possible to achieve every computation on the same time. This solution is better for the computation time because the size of ξ has no effect on the global computation time. However because constants values are needed at the same time, it is not possible to use RAM. Hence, the resource usage is more important than for the two previous solutions, and the size for N and M will be limited by the amount of logical cells available in the FPGA.

In the following, we only consider the last solution.

VII. Implementation

On the one hand, for any couple of values for x_n and ξ_k, the calculation of ψ requires 15 multiplications. On the other hand, the Spartan3A has only 16 multipliers. Consequently, it is necessary to distribute a subset of multipliers to each computing branch. The consequence is that it is not possible to dedicate a multiplier to each multiplication. Each component needs to be used more than once.

In order to do that, the only solution consists in creating a Finite State Machine (*FSM*). Equations are split into multiple states, the number of states depending on the amount of multipliers blocks available for a branch.

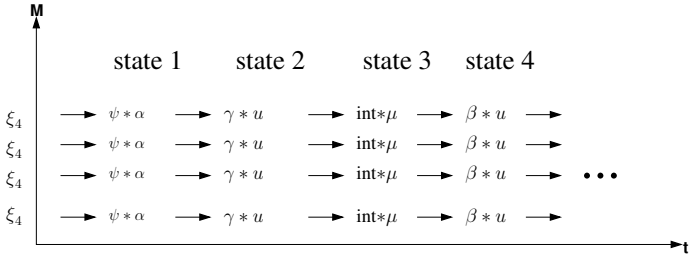

Fig. 5. Example of FSM to implement equations

Figure VII presents a possible solution with 4 multipliers per branch.

For each state, the output of a multiplier is affected through addition or substraction to registers, and the next computed values are affected to the multiplier inputs.

The complete structure is split into two components:

- the first is dedicated to handle state evolution, steering RAM.
- the second is dedicated to load values on multipliers and to store results.

This structure has been used for following reasons:

- reducing logical cell usage by having only one state machine,
- a RAM or a signal cannot have more than one source. Other solutions would have arisen problems to handle RAMs.

VIII. Area usage of FGPA

The chosen solution has the drawback of requiring many logical cells to store constants. Consequently, with a Spartan3A FPGA, the maximum size for constants is illustrated in Figure 6 where optimal values seems to be $N = 15$ and $M = 4$ or $N = 8$ and $M = 6$.

The evolution curve depends on many parameters:

- the number of significant bit for values, constants and results,
- the magnitude of N and M.

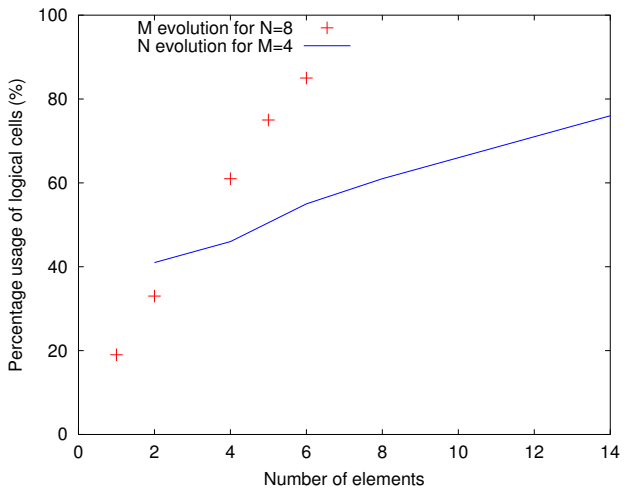

Fig. 6. Percentage of logical cells usage with respect to M and N for a 9 bits integer encoding

In consequence, for any other encoding integer number of bits, the area usage of logical cells is different.

It evolution of area usage as a function of these parameters is difficult to predict, and is here considered experimentally. For example, the increase on area usage is different for 1 bit added for different M or N magnitude. But, increasing N has a lower impact than increasing M, since N changes only the size for u (stored in RAM) and for μ.

On the other hand, increasing M increases the size of every constant stored with logical cells.

IX. Results

In order to validate our FGPA implementation and measure the execution time of each solution, we compare results synthesized on the FGPA, running on MATLAB, and the equivalent program implemented in C and running on a general purpose computer for a parameter set $M = 4$ and $N = 8$. Numerical results are exactly identical whether resulting from the execution on the FPGA, on MATLAB or using the C programs: the algorithm implementation is thus considered correct. For comparing elapsed time, we executed the C program on a laptop computer with an x86 processor 1.6GHz. It takes approximately $60,000ns$ to run the simulation.

Time results are given in Figure 7: we observe that on the FPGA, the computation time is approximately $720ns$, or an improvement of nearly two orders of magnitude with respect to the general purpose computer.

Figure 8 exhibits the extrapolation of the computation time for others solutions. These results were obtained using the following assumptions

- $t_\psi = n_{state} * t_{clock}$
- $t_{parallel} = t_\psi * M$
- $t_{iterative} = t_\psi * M * N$
- $t_{pipeline} = t_\psi + ((M * N) - 1) * t_{clock}$

with:

- t_ψ = time to compute one occurence of ψ,
- n_{state} =number of FSM states,
- t_{clock} = clock period.

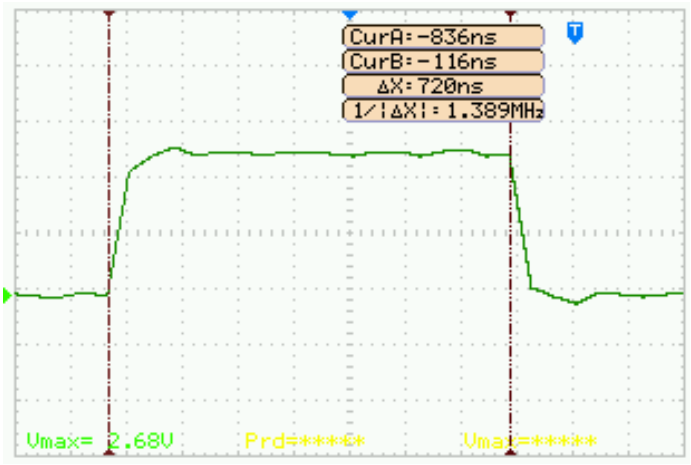

Fig. 7. Generation of a signal indicating the begining and the end of the compute, observed using a 60 MHz bandwidth oscilloscope.

The single-step computation time predicted from FPGA synthesis simulation are consistent with experimental timing measurements as shown in Fig. 7.

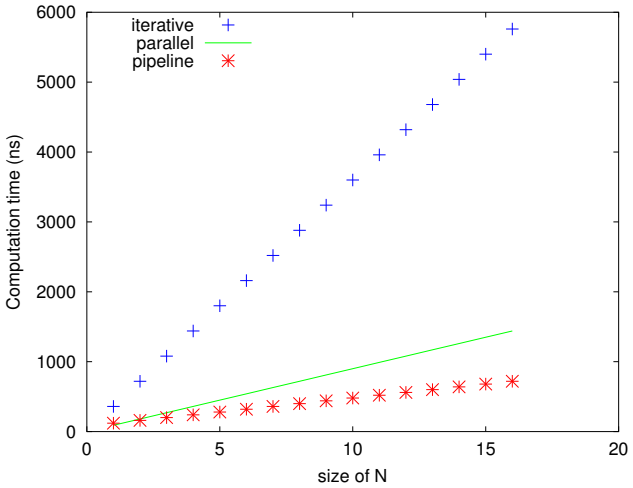

Fig. 8. Simulated computation time for $M = 4$ and $1 \leq N \leq 16$

Figure 8 exhibits the estimated computation time for others sets of parameters M and N.

For the same values $M = 4$ and $N = 8$, we estimate the following computation times:

- iterative solution: 2800 ns,
- pipeline solution: 400 ns.

for $N = 8$ and $M = 4$, the pipeline solution is better. Indeed the advantage of a parallel implementation appears when $N < \frac{n_{stat}-1}{n_{stat}-M}$ with n_{stat} the number of state for solved an occurence (Fig. 9).

X. Conclusion

We have shown that the diffusive representation is well suited for FPGA implementation of the realization of a linear operator which plays a central role in a distributed filtering or control law. We have developed every steps to analyze computation accuracy. In particular, we have tested data formats consistent with hardware implementation. Our results proved a high efficiency in terms of computation time.

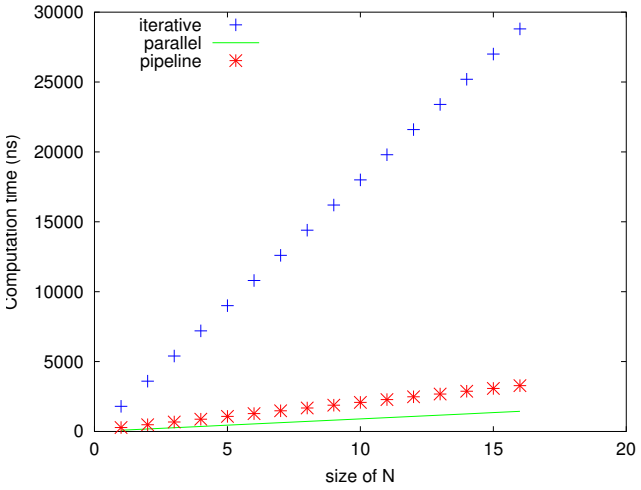

Fig. 9. Simulated computation time for $M = 20$ and $1 \leq N \leq 16$

However, using a Spartan3A FPGA, the amount of available values for constants and for inputs u is limited by the number of logical cells. In a future work, we plan to implement an evolution of this control law. In particular, we will focus on the practically useful case where M is lower than N, and implement the pipeline mechanism that would yield best performances.

REFERENCES

[1] G. Montseny, *Représentation diffusive*, Hermès-Sciences, 2005.

[2] L. Laudebat & P. Bidan & G. Montseny. *Modeling and optimal identification of pseudodifferential electrical dynamics by means of diffusive representation - Part 1: Modeling*. IEEE Transactions on Circuits and Systems I-Regular Papers, 51(9):1801–1813, 2004.

[3] M. Lenczner & G. Montseny, *Diffusive realization of operator solutions of certain operational partial differential equations*, C. R. Math. Acad. Sci. Paris, 341(12): 737–740, 2005.

[4] Y. Yakoubi, *Deux Méthodes d'Approximation pour le Contrôle Optimal Semi-Décentralisé des Systèmes Distribués*, Thèse, Université de Franche-Comté, Besançon 2010.

[5] T. Hélie & D. Matignon & R. Mignot. Criterion design for optimizing low-cost approximations of infinite-dimensional systems: towards efficient real-time simulation. Int. J. Tomogr. Stat., 7(F07):13–18, 2007.

[6] M. López-Fernández & C. Lubich, C. Palencia & A. Schädle. ast Runge-Kutta approximation of inhomogeneous parabolic equations. Numer. Math., 102(2):277–291, 2005.

[7] J. A. C. Weideman & L. N. Trefethen, *Parabolic and hyperbolic contours for computing the Bromwich integral*, Math. Comput., Volume 76, Number 259, July 2007, pp 1341–1356.

[8] R. Airiau & J. M. Bergé & V. Olive & J. Rouillard, *VHDL, langage, modélisation, synthèse. 2nd Ed.*, Presses Polytechniques et Universitaires Romandes (1998)

[9] P. P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*, Wiley (2008).

[10] S.Kilts, *ADVANCED FPGA DESIGN, Architecture, implementation, and Optimization*, Wiley Interscience/IEEE PRESS(2007).

[11] *Spartan 3A datasheet.*, http://www.xilinx.com/support/documentation/data_sheets/ds529.pdf.

[12] *Wishbone bus specification.*, http://www.opencores.org/opencores, wishbone.