FPGA Implementation of Diffusive Realization for a Distributed Control Operator

G. Goavec-Merou*, Y. Yakoubi†, R. Couturier‡, M. Lenczner*, J.M. Fried† 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-Comté, 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 fine 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 computation based on fine grained parallel computer architectures as Field-Programmable Gate Arrays (FPGA). Here we address the 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,

\[
d^2\!u \over dx^2 + P \frac{d^2}{dx^2} u = Qu
\]

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

\[
\partial T / \partial t - \partial^2 T / \partial x^2 = q \quad \text{in} \quad \omega.
\]

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 guaranteeing 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

\[
-\Delta p = q \quad \text{in} \quad \left(0, 1\right) \times \left(0, 1\right),
\]

and \( p = 0 \) on the square boundary \( \left(0, 1\right) \times \left(0, 1\right), \)

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

\[
z^+(x) = \int_R \mu^+(x, \xi)\psi^+(x, \xi) \, d\xi
\]

and \( z^-(x) = \int_R \mu^-(x, \xi)\psi^-(x, \xi) \, d\xi, \)

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

\[
\partial_x \psi^+(x, \xi) + \theta^+(\xi)\psi^+(x, \xi) = u(x)
\]

with \( \psi^+(0, \xi) = 0 \),

and \( \partial_x \psi^-(x, \xi) - \theta^-(\xi)\psi^-(x, \xi) = u(x) \)

with \( \psi^-(1, \xi) = 0 \),

where \( \xi \) is a real parameter. We emphasis that they are independently of a specific operator \( P \). Conversely, the coefficients \( \mu^+ \) and \( \mu^- \), 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 encasing the singularities of the Laplace transform \( P\hat{\cdot} \) defined hereafter. The diffusive symbol derivation requires several steps. The two functions

\[
y \mapsto p(x, x - y) \quad \text{and} \quad y \mapsto p(x, x + y),
\]

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 \( P^+ \) and \( P^- \) are well-defined in \( \mathbb{C}^+ \), and
that they admit holomorphic extensions vanishing at infinity. The Diffusive symbols are then given by
\[
\mu^\pm (x, \xi) = -\frac{\theta^\pm (\xi)}{2\pi} \mathcal{P}^\pm (x, \theta^\pm (\xi)).
\] (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 enclose, by the right, the poles of Laplace transforms of the basis functions, and the approximations \( \mu^\pm \) 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 \( (x_n)_{n=0,...,N} \in (0,1) \), thanks to a trapezoidal rule by
\[
z_n = h \xi \sum_{k=-M}^{M} \mu_{-k,n}^+ \psi_{k,n}^+ + \mu_{n,k}^+ \psi_{n,k}^-,
\] (8)
for a \( M \in \mathbb{N}^* \), where \( h_\xi > 0 \) is the step size (distance between two consecutive integration point), \( \mu_{n,k}^+ \) and \( \psi_{n,k}^- \) are some discrete values of \( \mu^N \) and \( \psi \) at the points \( x_n \) and \( \xi_k \). 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
\[
-\theta^\pm (\xi) = \theta_p (i\xi + 1)^2 \quad \text{for} \quad \xi \in \mathbb{R},
\] (9)
and a hyperbola
\[
-\theta^\pm (\xi) = \theta_h (1 + \sin (i\xi - \alpha)) \quad \text{for} \quad \xi \in \mathbb{R},
\] (10)
for some positive real numbers \( \theta_p, \theta_h \) and \( \alpha \) 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 \( \{x_n \mid n = 0,...,N\} \) 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 \( \psi_{n,k}^- \) is done in the computation. Such quadrature rules are identical to that one we only considered the former.

To be efficient on FPGA architecture, one must avoid, when possible, floating point computation. 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. Nevertheless, the smaller the number of bits used to encode integer in the FPGA architecture is, the less numerous are the logical cells used, so the more parallelism capabilities we obtain.

In this section we explain how we proceed to prepare Algorithm 1 before implementing it 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: \( \beta, \gamma, \mu \) and \( u \). Since \( \beta \) and \( \gamma \) depend on \( \alpha \), we do not scale \( \alpha \) which is almost naturally scaled without doing anything. So, for \( \gamma, \mu \) and \( u \), we compute the maximum values, respectively called \( \gamma_{\text{max}} \), \( \mu_{\text{max}} \) and \( u_{\text{max}} \). Then we compute the scaled variables, respectively called \( \beta_{\text{scaled}}, \gamma_{\text{scaled}}, \mu_{\text{scaled}}, u_{\text{scaled}} \) with \( \beta_{\text{scaled}} = \beta / \gamma_{\text{max}}, \gamma_{\text{scaled}} = \gamma / \gamma_{\text{max}}, \mu_{\text{scaled}} = \mu / \mu_{\text{max}} \) and \( u_{\text{scaled}} = u / u_{\text{max}} \). It should be noticed that \( \beta \) and \( \gamma \) depend on \( \alpha \), and \( \alpha \) is chosen to scale those variables by \( \gamma_{\text{max}} \) which is larger than \( \beta_{\text{max}} \).

In Algorithm 1, we can deduce that \( \psi \) (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 \( \mu_{\text{max}} \), \( \gamma_{\text{max}} \) and \( u_{\text{max}} \) so in order to obtain the final result, we need to multiply the result by \( u_{\text{max}} \mu_{\text{max}} \). Thus, \( z = z_{\text{scaled}} \times u_{\text{max}} 	imes \beta_{\text{max}} \times \mu_{\text{max}} \).

To be efficient on FPGA 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.

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

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

IV. Scaling and Value Encoding
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 able to 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.

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.

<table>
<thead>
<tr>
<th>Number of bits</th>
<th>Maximum error</th>
<th>Relative error</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>9.21e-5</td>
<td>1.05%</td>
</tr>
<tr>
<td>15</td>
<td>2.08e-4</td>
<td>2.51%</td>
</tr>
<tr>
<td>14</td>
<td>1.38e-4</td>
<td>1.51%</td>
</tr>
<tr>
<td>13</td>
<td>8.82e-4</td>
<td>6.79%</td>
</tr>
<tr>
<td>12</td>
<td>6.87e-4</td>
<td>9.23%</td>
</tr>
<tr>
<td>11</td>
<td>1.4e-3</td>
<td>18.69%</td>
</tr>
</tbody>
</table>

**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:

- 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 $\psi$, $u(x_n)$, and $\xi_k$. As presented in Figure 3, for a same $\xi$, $\psi(x_{n+1}, \xi)$ 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 $\psi$ for different $\xi$ are independent.

Figure 4 shows the relation between the contribution of $P(u_n)$: there are fewer constraints than $\psi$ 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 resource 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
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 resources are presented.

A. Iterative

The first solution consists in iterating on $\xi$ values. For each value of $\xi$, it consists in computing the $\psi$ 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 $\psi$ for a specific $\xi_k$, the total computation time is equal to the time needed to perform the computation for one value of $\psi_k$ 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 $\psi$ values to make them available for the next $\xi$.

C. Parallel

The last solution consists in implementing the computation in parallel. Because there is no dependency between $\psi(x_n, 0)$ and another $\psi$ for a different $\xi$, it is possible to achieve every computation on the same time. This solution is better for the computation time because the size of $\xi$ 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 $\xi_k$, the calculation of $\psi$ 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.
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 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

\[ t_\psi = n_{state} \times t_{clock} \]
\[ t_{parallel} = t_\psi \times M \]
\[ t_{iterative} = t_\psi \times M \times N \]
\[ t_{pipeline} = t_\psi + ((M \times N) - 1) \times t_{clock} \]

with:

- \( t_\psi \) = time to compute one occurrence of \( \psi \),
- \( n_{state} \) = number of FSM states,
- \( t_{clock} \) = clock period.

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 $\alpha$ 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