Abstract

Atomic force microscopes (AFM) provide high resolution images of surfaces. In this paper, we focus our attention on an interferometry method for deflection estimation of cantilever arrays in quasi-static regime. In its original form, spline interpolation was used to determine interference fringe phase, and thus the deflections. Computations were performed on a PC. Here, we propose a new complete solution with a least square based algorithm and an optimized FPGA implementation. Simulations and real tests showed very good results and open perspective for real-time estimation and control of cantilever arrays in the dynamic regime.

Keywords

FPGA, cantilever arrays, interferometry.

A new approach based on a least square method for real-time estimation of cantilever array deflections with a FPGA

I. Introduction

Cantilevers are used in atomic force microscopes (AFM) which provide high resolution surface images. Several techniques have been reported in literature for cantilever displacement measurement. In [2], authors have shown how a piezoresistor can be integrated into a cantilever for deflection measurement. Nevertheless this approach suffers from the complexity of the microfabrication process needed to implement the sensor. In [3], authors have presented a cantilever mechanism based on capacitive sensing. These techniques require cantilever instrumentation resulting in complex fabrication processes.

In this paper our attention is focused on a method based on interferometry for cantilever displacement measurement in quasi-static regime. Cantilevers are illuminated by an optical source. Interferometry produces fringes enabling cantilever displacement computation. A high speed camera is used to analyze the fringes. In view of real time applications, images need to be processed quickly and then a fast estimation method is required to determine the displacement of each cantilever. In [4], an algorithm based on spline has been introduced for cantilever position estimation. The overall process gives accurate results but computations are performed on a standard computer using LabView ®. Consequently, the main drawback of this implementation is that the computer is a bottleneck. In this paper we pose the problem of real-time cantilever position estimation and bring a hardware/software solution. It includes a fast method based on least squares and its FPGA implementation.

The remainder of the paper is organized as follows. Section II describes the measurement process. Our solution based on the least square method and its implementation on a FPGA is presented in Section III. Numerical experimentations are described in Section IV. Finally a conclusion and some perspectives are drawn.

II. Architecture and goals

In order to build simple, cost effective and user-friendly cantilever arrays, we use a system based on interferometry. The two following sections summarize the original characteristics of its architecture and computation method.

A. Experimental setup

In opposition to other optical based systems using a laser beam deflection scheme and sensitive to the angular displacement of the cantilever, interferometry is sensitive to the optical path difference induced by the vertical displacement of the cantilever.

The system is based on a Linnick interferometer [8]. It is illustrated in Figure 1. A laser diode is first split (by the splitter) into a reference beam and a sample beam both reaching the cantilever array. The complete system including a cantilever array and the optical system can be moved thanks to a translation and rotational hexapod stage with five degrees of freedom. Thus, the cantilever array is centered in the optical system which can be adjusted accurately. The beam illuminates the array by a microscope objective and the light reflects on the cantilevers. Likewise the reference beam reflects on a movable mirror. A CMOS camera chip records the reference and sample beams which are recombined in the beam splitter and the interferogram. At the beginning of each experiment, the movable mirror is fitted manually in order to align the interferometric fringes approximately parallel to the cantilevers. Then, cantilever motion in the transverse direction produces movements in the fringes. They are detected with the CMOS camera which images are analyzed by a Labview program to recover the cantilever deflections.

B. Inteferometric based cantilever deflection estimation

As shown in Figure 2, each cantilever is covered by several interferometric fringes. The fringes distort when cantilevers are deflected. For each cantilever, the method uses three segments of pixels, parallel to its section, to determine phase shifts. The first is located just above the AFM tip (tip profile), it provides the phase shift modulo 2π. The second one is close to the base

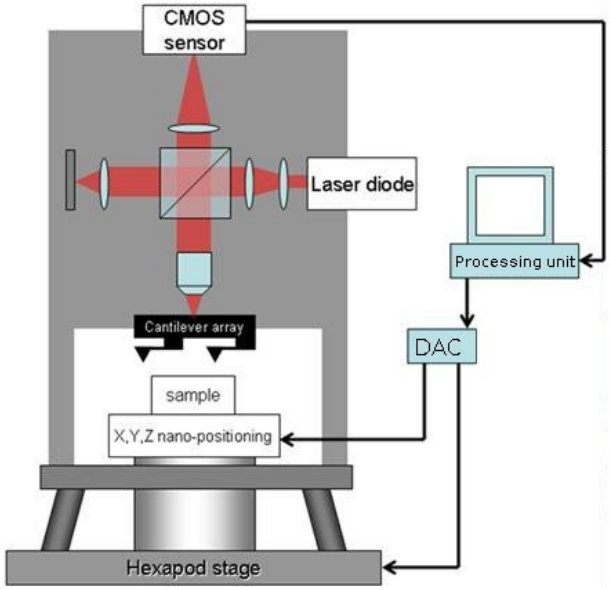

Fig. 1. AFM Setup

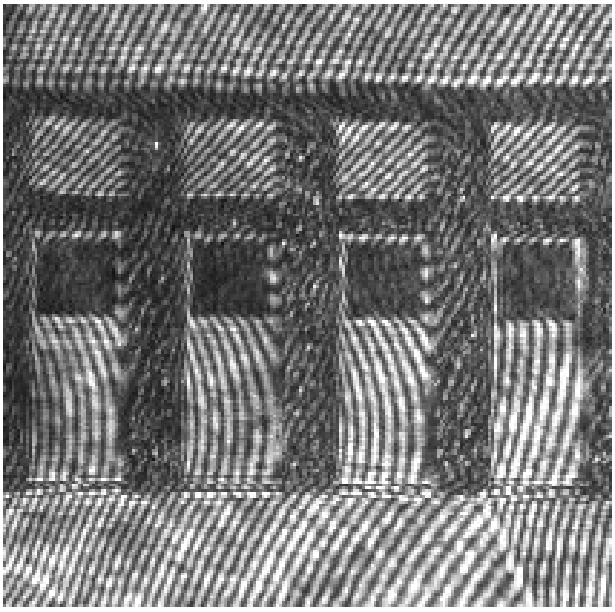

Fig. 2. Portion of a camera image showing moving interferometric fringes in cantilevers

junction (base profile) and is used to determine the exact multiple of 2π through an operation called unwrapping where it is assumed that the deflection means along the two measurement segments are linearly dependent. The third is on the base and provides a reference for noise suppression. Finally, deflections are simply derived from phase shifts.

The pixel gray-level intensity I of each profile is modelized by

$$I(x) = A\ \cos(2\pi f x + \theta) + ax + b \qquad (1)$$

where x denotes the position of a pixel in a segment, A, f and θ are the amplitude, the frequency and the phase of the light signal when the affine function $ax + b$ corresponds to the cantilever array surface tilt with respect to the light source.

The method consists in two main sequences. In the first one corresponding to precomputation, the frequency f of each profile is determined using a spline interpolation (see section III-C1) and the coefficients used for phase unwrapping are computed. The second one, that we call the *acquisition loop,* is done after images have been taken at regular time steps. For each image, the phase θ of all profiles is computed to obtain, after unwrapping, the cantilever deflection. The phase determination is achieved by a spline based algorithm, which is the most consuming part of the computation. In this article, we propose an alternate version based on the least square method which is faster and better suited for FPGA implementation. Moreover, it can be used in real-time, i.e. after each image is picked by the camera.

C. Computation design goals

To evaluate the solution performances, we choose a goal which consists in designing a computing unit able to estimate the deflections of a 10×10-cantilever array, faster than the camera image stream. In addition, the result accuracy must be close to 0.3nm, the maximum precision reached in [4]. Finally, the latency between the entrance of the first pixel of an image and the end of deflection computation must be as small as possible. All these requirement are stated in the perspective of implementing real-time active control for each cantilever, see [6], [5].

If we put aside other hardware issues like the speed of the link between the camera and the computation unit, the time to deserialize pixels and to store them in memory, the phase computation is the bottleneck of the whole process. For example, the camera in the setup of [4] provides 1024×1204 pixels with an exposition time of 2.5ms. Thus, if the pixel extraction time is neglected,

each phase calculation of a 100-cantilever array should take no more than 12.5μs.

In fact, this timing is a very hard constraint. To illustrate this point, we consider a very small program that initializes twenty million of doubles in memory and then does 1,000,000 cumulated sums on 20 contiguous values (experimental profiles have about this size). On an intel Core 2 Duo E6650 at 2.33GHz, this program reaches an average of 155Mflops. Obviously, some cache effects and optimizations on huge amount of computations can drastically increase these performances: peak efficiency is about 2.5Gflops for the considered CPU. But this is not the case for phase computation that is using only a few tenth of values.

In order to evaluate the original algorithm, we translated it in C language. As stated in section III-C3, for 20 pixels, it does about 1,550 operations, thus an estimated execution time of $1,550/155 = 10\mu$s. For a more realistic evaluation, we constructed a file of 1Mo containing 200 profiles of 20 pixels, equally scattered. This file is equivalent to an image stored in a device file representing the camera. We obtained an average of 10.5μs by profile (including I/O accesses). It is under our requirements but close to the limit. In case of an occasional load of the system, it could be largely overtaken. Solutions would be to use a real-time operating system or to search for a more efficient algorithm.

However, the main drawback is the latency of such a solution because each profile must be treated one after another and the deflection of 100 cantilevers takes about $200 \times 10.5 = 2.1$ms. This would be inadequate for real-time requirements as for individual cantilever active control. An obvious solution is to parallelize the computations, for example on a GPU. Nevertheless, the cost of transferring profile in GPU memory and of taking back results would be prohibitive compared to computation time.

It should be noticed that when possible, it is more efficient to pipeline the computation. For example, supposing that 200 profiles of 20 pixels could be pushed sequentially in a pipelined unit cadenced at a 100MHz (i.e. a pixel enters in the unit each 10ns), all profiles would be treated in $200 \times 20 \times 10.10^{-9} = 40\mu$s plus the latency of the pipeline. Such a solution would be meeting our requirements and would be 50 times faster than our C code, and even more compared to the LabView version. FPGAs are appropriate for such implementation, so they turn out to be the computation units of choice to reach our performance requirements. Nevertheless, passing from a C code to a pipelined version in VHDL is not obvious at all. It can even be impossible because of FPGA hardware constraints. All these points are discussed in the following sections.

III. An hardware/software solution

In this section we present parts of the computing solution to the above requirements. The hardware part consists in a high-speed camera linked on an embedded board hosting two FPGAs. In this way, the camera output stream can be pushed directly into the FPGA. The software part is mostly the VHDL code that deserializes the camera stream, extracts profiles and computes the deflection.

We first give some general information about FPGAs, then we describe the FPGA board we use for implementation and finally the two algorithms for phase computation are detailed. Presentation of VHDL implementations is postponned until Section IV.

A. Elements of FPGA architecture and programming

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by the customer. FGPAs are composed of programmable logic components, called configurable logic blocks (CLB). These blocks mainly contain look-up tables (LUT), flip/flops (F/F) and latches, organized in one or more slices connected together. Each CLB can be configured to perform simple (AND, XOR, ...) or complex combinational functions. They are interconnected by reconfigurable links. Modern FPGAs contain memory elements and multipliers which enable to simplify the design and to increase the performance. Nevertheless, all other complex operations like division and other functions like trigonometric functions are not available and must be built by configuring a set of CLBs. Since this is not an obvious task at all, tools like ISE [7] have been built to do this operation. Such a software can synthetize a design written in a hardware description language (HDL), maps it onto CLBs, place/route them for a specific FPGA, and finally produces a bitstream that is used to configure the FPGA. Thus, from the developer's point of view, the main difficulty is to translate an algorithm into HDL code, taking into account FPGA resources and constraints like clock signals and I/O values that drive the FPGA.

Indeed, HDL programming is very different from classic languages like C. A program can be seen as a state-machine, manipulating signals that evolve from state to state. Moreover, HDL instructions can be executed concurrently. Signals may be combined with basic logic operations to produce new states that are assigned to another signal. States are mainly expressed

as arrays of bits. Fortunately, libraries propose higher levels representations like signed integers, and arithmetic operations.

Furthermore, even if FPGAs are cadenced more slowly than classic processors, they can perform pipelines as well as parallel operations. A pipeline consists in cutting a process in a sequence of small tasks, taking the same execution time. It accepts a new data at each clock top, thus, after a known latency, it also provides a result at each clock top. The drawback is that the components of a task are not reusable by another one. Nevertheless, this is the most efficient technique on FPGAs. Because of their architecture, it is also very easy to process several data concurrently. Finally, the best performance can be reached when several pipelines are operating on multiple data streams in parallel.

B. The FPGA board

The architecture we use is designed by the Armadeus Systems company. It consists in a development board called APF27 ®, hosting a i.MX27 ARM processor (from Freescale) and a Spartan3A (from Xilinx). This board includes all classical connectors as USB and Ethernet for instance. A Flash memory contains a Linux kernel that can be launched after booting the board via u-Boot. The processor is directly connected to the Spartan3A via its special interface called WEIM. The Spartan3A is itself connected to an extension board called SP Vision ®, that hosts a Spartan6 FPGA. Thus, it is possible to develop programs that communicate between i.MX and Spartan6, using Spartan3 as a tunnel. A clock signal at 100MHz (by default) is delivered to dedicated FPGA pins. The Spartan6 of our board is an LX100 version. It has 15,822 slices, each slice containing 4 LUTs and 8 flip/flops. It is equivalent to 101,261 logic cells. There are 268 internal block RAM of 18Kbits, and 180 dedicated multiply-adders (named DSP48), which is largely enough for our project. Some I/O pins of Spartan6 are connected to two 2×17 headers that can be used for any purpose as to be connected to the interface of a camera.

C. Two algorithms for phase computation

As said in section II-B, f is computed only once but the phase needs to be computed for each image. This is why, in this paper, we focus on its computation. The next section describes the original method, based on spline interpolation, and section III-C2 presents the new one based on least squares. Finally, in section III-C3, we compare the two algorithms from their FPGA implementation point of view.

1) Spline algorithm (SPL): We denote by M the number of pixels in a segment used for phase computation. For the sake of simplicity of the notations, we consider the light intensity I a function on the interval [0,M] which itself is the range of a one-to-one mapping defined on the physical segment. The pixels are assumed to be regularly spaced and centered at the positions $x^p \in \{0, 1, \ldots, M-1\}$. We use the simplest definition of a pixel, namely the value of I at its center. The pixel intensities are considered as pre-normalized so that their minimum and maximum have been resized to -1 and 1.

The first step consists in computing the cubic spline interpolation of the intensities. This allows for interpolating I at a larger number $L = k \times M$ of points (typically $k = 4$ is sufficient) x^s in the interval $[0, M[$. During the precomputation sequence, the second step is to determine the affine part $a.x + b$ of I. It is found with an ordinary least square method, taking account the L points. Values of I in x^s are used to compute its intersections with $a.x + b$. The period of I (and thus its frequency) is deduced from the number of intersections and the distance between the first and last.

During the acquisition loop, the second step is the phase computation, with

$$\theta = atan\left[\frac{\sum_{i=0}^{N-1} \sin(2\pi f x_i^s) \times I(x_i^s)}{\sum_{i=0}^{N-1} \cos(2\pi f x_i^s) \times I(x_i^s)}\right]. \qquad (2)$$

Remarks:

- The frequency could also be obtained using the derivative of spline equations, which only implies to solve quadratic equations but certainly yields higher errors.
- Profile frequency are computed during the precomputation step, thus the values $\sin(2\pi f x_i^s)$ and $\cos(2\pi f x_i^s)$ can be determined once for all.

2) Least square algorithm (LSQ): Assuming that we compute the phase during the acquisition loop, equation 1 has only 4 parameters: a, b, A, and θ, f and x being already known. A least square method based on a Gauss-Newton algorithm can be used to determine these four parameters. This kind of iterative process ends with a convergence criterion, so it is not suited to our design goals. Fortunately, it is quite simple to reduce the number of parameters to θ only. Firstly, the affine part $ax + b$ is estimated from the M values $I(x^p)$ to determine the rectified intensities,

$$I^{corr}(x^p) \approx I(x^p) - a.x^p - b.$$

To find a and b we apply an ordinary least square method (as in SPL but on M points)

$$a = \frac{covar(x^p, I(x^p))}{\text{var}(x^p)} \text{ and } b = \overline{I(x^p)} - a.\overline{x^p}$$

where overlined symbols represent average. Then the amplitude A is approximated by

$$A \approx \frac{\max(I^{corr}) - \min(I^{corr})}{2}.$$

Finally, the problem of approximating θ is reduced to minimizing

$$\min_{\theta\in[-\pi,\pi]} \sum_{i=0}^{M-1} \left[\cos(2\pi f.i + \theta) - \frac{I^{corr}(i)}{A}\right]^2.$$

An optimal value θ^* of the minimization problem is a zero of the first derivative of the above argument,

$$2\left[\cos\theta^* \sum_{i=0}^{M-1} I^{corr}(i).\sin(2\pi f.i) + \sin\theta^* \sum_{i=0}^{M-1} I^{corr}(i).\cos(2\pi f.i)\right] -$$

$$A\left[\cos 2\theta^* \sum_{i=0}^{M-1} \sin(4\pi f.i) + \sin 2\theta^* \sum_{i=0}^{M-1} \cos(4\pi f.i)\right] = 0$$

Several points can be noticed:

- The terms $\sum_{i=0}^{M-1}\sin(4\pi f.i)$ and$\sum_{i=0}^{M-1}\cos(4\pi f.i)$ are independent of θ, they can be precomputed.
- Lookup tables (namely lut_{sfi} and lut_{cfi} in the following algorithms) can be set with the $2.M$ values $\sin(2\pi f.i)$ and $\cos(2\pi f.i)$.
- A simple method to find a zero θ^* of the optimality condition is to discretize the range $[-\pi, \pi]$ with a large number nb_s of nodes and to find which one is a minimizer in the absolute value sense. Hence, three other lookup tables (lut_s, lut_c and lut_A) can be set with the $3 \times nb_s$ values $\sin\theta$, $\cos\theta$, and

$$\left[cos2\theta \sum_{i=0}^{M-1} sin(4\pi f.i) + sin2\theta \sum_{i=0}^{M-1} cos(4\pi f.i)\right].$$

- The search algorithm can be very fast using a dichotomous process in $log_2(nb_s)$.

The overall method is synthetized in an algorithm (called LSQ in the following) divided into the precomputing part and the acquisition loop.

Algorithm 1: LSQ algorithm - before acquisition loop.

```
M ← number of pixels of the profile
I[] ← intensity of pixels
f ← frequency of the profile
s4i ← Σ_{i=0}^{M-1} sin(4πf.i)
c4i ← Σ_{i=0}^{M-1} cos(4πf.i)
nb_s ← number of discretization steps of [−π, π]
for i = 0 to nb_s do
    θ ← −π + 2π × i/nb_s
    lut_s[i] ← sinθ
    lut_c[i] ← cosθ
    lut_A[i] ← cos2θ × s4i + sin2θ × c4i
    lut_sfi[i] ← sin(2πf.i)
    lut_cfi[i] ← cos(2πf.i)
end
```

3) Algorithm comparison: We compared the two algorithms regarding three criteria:

- precision of results on a cosines profile distorted by noise,
- number of operations,
- complexity of FPGA implementation.

For the first item, we produced a Matlab version of each algorithm, running in double precision. The profile was generated for about 34,000 different quadruplets of periods ($\in [3.1, 6.1]$, step = 0.1), phases ($\in [-3.1, 3.1]$, steps = 0.062) and slopes ($\in [-2, 2]$, step = 0.4). Obviously, the discretization of $[-\pi, \pi]$ introduces an error in the phase estimation. It is at most equal to $\frac{\pi}{nb_s}$. From some experiments on a 17×4 array, we noticed an average ratio of 50 between phase variation in radians and lever end position in nanometers. Assuming such a ratio and $nb_s = 1024$, the maximum lever deflection error would be 0.15nm which is smaller than 0.3nm, the best precision achieved with the setup used.

Moreover, pixels have been paired and the paired intensities have been perturbed by addition of a random number uniformly picked in $[-N, N]$. Notice that we have observed that perturbing each pixel independently yields too weak profile distortion. We report percentages of errors between the reference and the computed phases out of 2π,

$$err = 100 \times \frac{|\theta_{ref} - \theta_{comp}|}{2\pi}.$$

Table I gives the maximum and the average errors for both algorithms and for increasing values of N the noise parameter.

Algorithm 2: LSQ algorithm - during acquisition loop.

$\bar{x} \leftarrow \frac{M-1}{2}$
$\bar{y} \leftarrow 0$, $x_{var} \leftarrow 0$, $xy_{covar} \leftarrow 0$
for $i = 0$ *to* $M-1$ **do**
  $\bar{y} \leftarrow \bar{y}+$ I$[i]$
  $x_{var} \leftarrow x_{var} + (i - \bar{x})^2$
end
$\bar{y} \leftarrow \frac{\bar{y}}{M}$
for $i = 0$ *to* $M-1$ **do**
  $xy_{covar} \leftarrow xy_{covar} + (i - \bar{x}) \times (I[i] - \bar{y})$
end
$slope \leftarrow \frac{xy_{covar}}{x_{var}}$
$start \leftarrow \bar{y} - slope \times \bar{x}$
for $i = 0$ *to* $M-1$ **do**
  $I[i] \leftarrow I[i] - start - slope \times i$
end
$I_{max} \leftarrow max_i(I[i])$, $I_{min} \leftarrow min_i(I[i])$
$amp \leftarrow \frac{I_{max}-I_{min}}{2}$
$Is \leftarrow 0$, $Ic \leftarrow 0$
for $i = 0$ *to* $M-1$ **do**
  $Is \leftarrow Is + I[i] \times$ lut$_{sfi}[i]$
  $Ic \leftarrow Ic + I[i] \times$ lut$_{cfi}[i]$
end
$\delta \leftarrow \frac{nb_s}{2}$, $b_l \leftarrow 0$, $b_r \leftarrow \delta$
$v_l \leftarrow -2.I_s - amp.lut_A[b_l]$
while $\delta >= 1$ **do**
  $v_r \leftarrow 2.[Is.$lut$_c[b_r]+Ic.$lut$_s[b_r]] - amp.$lut$_A[b_r]$
  if $!(v_l < 0$ *and* $v_r >= 0)$ **then**
    $v_l \leftarrow v_r$
    $b_l \leftarrow b_r$
  end
  $\delta \leftarrow \frac{\delta}{2}$
  $b_r \leftarrow b_l + \delta$
end
if $!(v_l < 0$ *and* $v_r >= 0)$ **then**
  $v_l \leftarrow v_r$
  $b_l \leftarrow b_r$
  $b_r \leftarrow b_l + 1$
  $v_r \leftarrow 2.[Is.$lut$_c[b_r]+Ic.$lut$_s[b_r]] - amp.$lut$_A[b_r]$
else
  $b_r \leftarrow b_l + 1$
end
if $abs(v_l) < v_r$ **then**
  $b_\theta \leftarrow b_l$
else
  $b_\theta \leftarrow b_r$
end
$\theta \leftarrow \pi \times \left[\frac{2.b_{ref}}{nb_s} - 1\right]$

noise (N)	SPL		LSQ	
	max. err.	aver. err.	max. err.	aver. err.
0	2.46	0.58	0.49	0.1
2.5	2.75	0.62	1.16	0.22
5	3.77	0.72	2.47	0.41
7.5	4.72	0.86	3.33	0.62
10	5.62	1.03	4.29	0.81
15	7.96	1.38	6.35	1.21
30	17.06	2.6	13.94	2.45

TABLE I
ERROR (IN %) FOR COSINES PROFILES, WITH NOISE.

The results show that the two algorithms yield close results, with a slight advantage for LSQ. Furthermore, both behave very well against noise. Assuming an average ratio of 50 (see above), an error of 1 percent on the phase corresponds to an error of 0.5nm on the lever deflection, which is very close to the best precision.

It is very hard to predict which level of noise will be present in real experiments and how it will distort the profiles. Results on a 17×4 array allowed us to compare experimental profiles to simulated ones. We can see on figure 3 the profile with $N = 10$ that leads to the biggest error. It is a bit distorted, with pikes and straight/rounded portions. In fact, it is very close to some of the worst experimental profiles. Figure 4 shows a sample of worst profile for $N = 30$. It is completely distorted, largely beyond any experimental ones. Obviously, these comparisons are a bit subjective and experimental profiles could also be more distorted on other experiments. Nevertheless, they give an idea about the possible error.

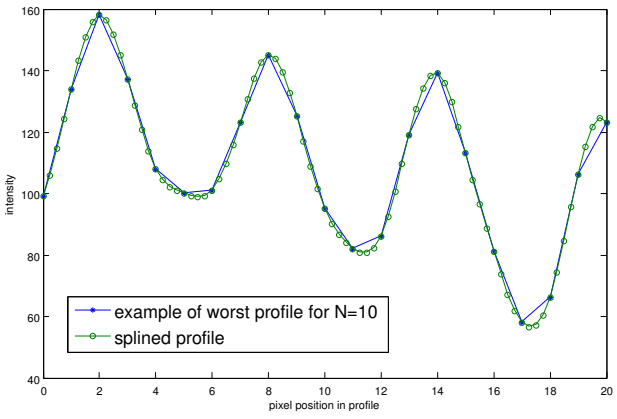

Fig. 3. Sample of worst profile for N=10

The second criterion is relatively easy to estimate for LSQ and harder for SPL because of the use of the arctangent function. In both cases, the number of

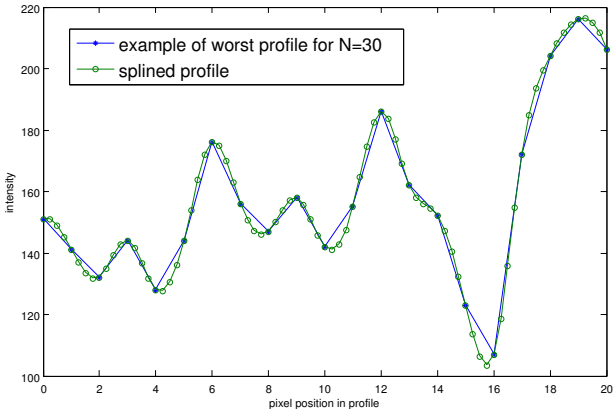

Fig. 4. Sample of worst profile for N=30

operation is proportional to M the number of pixels. For LSQ, it also depends on nb_s and for SPL on $L = k \times M$ the number of interpolated points. We assume that $M = 20$, $nb_s = 1024$ and $k = 4$, that all possible parts are already in lookup tables and that a limited set of operations (+, -, *, /, <, >) is taken into account. Translating both algorithms in C code, we obtain about 430 operations for LSQ and 1,550 (plus a few tenth for $atan$) for SPL. This result is largely in favor of LSQ. Nevertheless, considering the total number of operations is not fully relevant for FPGA implementation for which time and space consumption depends not only on the type of operations but also of their ordering. The final evaluation is thus very much driven by the third criterion.

The Spartan 6 used in our architecture has a hard constraint since it has no built-in floating point units. Obviously, it is possible to use some existing "black-boxes" for double precision operations. But they require a lot of clock cycles to complete. It is much simpler to exclusively use integers, with a quantization of all double precision values. It should be chosen in a manner that does not alterate result precision. Furthermore, it should not lead to a design with a huge latency because of operations that could not complete during a single or few clock cycles. Divisions fall into that category and, moreover, they need a varying number of clock cycles to complete. Even multiplications can be a problem since a DSP48 takes inputs of 18 bits maximum. So, for larger multiplications, several DSP must be combined which increases the overall latency.

Nevertheless, in the present algorithms, the hardest constraint does not come from the FPGA characteristics but from the algorithms themselves. Their VHDL implementation can be efficient only if they can be fully (or near) pipelined. We observe that only a small part of SPL can be pipelined, indeed, the computation of spline coefficients implies to solve a linear tridiagonal system which matrix and right-hand side are computed from incoming pixels intensity but after, the back-solve starts with the latest values, which breaks the pipeline. Moreover, SPL relies on interpolating far more points than profile size. Thus, the end of SPL works on a larger amount of data than at the beginning, which also breaks the pipeline.

LSQ has not this problem since all parts, except the dichotomic search, work on the same amount of data, i.e. the profile size. Furthermore, LSQ requires less operations than SPL, implying a smaller output latency. In total, LSQ turns out to be the best candidate for phase computation on any architecture including FPGA.

IV. VHDL IMPLEMENTATION AND EXPERIMENTAL TESTS

A. VHDL implementation

From the LSQ algorithm, we have written a C program that uses only integer values. We used a very simple quantization which consists in multiplying each double precision value by a factor power of two and by keeping the integer part. For an accurate evaluation of the division in the computation of a the slope coefficient, we also scaled the pixel intensities by another power of two. The main problem was to determine these factors. Most of the time, they are chosen to minimize the error induced by the quantization. But in our case, we also have some hardware constraints, for example the width and depth of RAMs or the input size of DSPs. Thus, having a maximum of values that fit in these sizes is a very important criterion to choose the scaling factors.

Consequently, we have determined the maximum value of each variable as a function of the scale factors and the profile size involved in the algorithm. It gave us the maximum number of bits necessary to code them. We have chosen the scale factors so that any variable (except the covariance) fits in 18 bits, which is the maximum input size of DSPs. In this way, all multiplications (except one with covariance) could be done with a single DSP, in a single clock cycle. Moreover, assuming that $nb_s = 1024$, all LUTs could fit in the 18Kbits RAMs. Finally, we compared the double and integer versions of LSQ and found a nearly perfect agreement between their results.

As mentionned above, some operations like divisions must be avoided. But when the divisor is fixed, a division can be replaced by its multiplication/shift counterpart. This is always the case in LSQ. For example, assuming

that M is fixed, x_{var} is known and fixed. Thus, $\frac{xy_{covar}}{x_{var}}$ can be replaced by

$$(xy_{covar} \times \left\lfloor \frac{2^n}{x_{var}} \right\rfloor) \gg n$$

where n depends on the desired precision (in our case $n = 24$).

Obviously, multiplications and divisions by a power of two can be replaced by left or right bit shifts. Finally, the code only contains shifts, additions, subtractions and multiplications of signed integers, which are perfectly adapted to FGPAs.

We built two versions of VHDL codes, namely one directly by hand coding and the other with Matlab using the Simulink HDL coder feature [1]. Although the approaches are completely different we obtained quite comparable VHDL codes. Each approach has advantages and drawbacks. Roughly speaking, hand coding provides beautiful and much better structured code while Simulink HDL coder allows fast code production. In terms of throughput and latency, simulations show that the two approaches yield close results with a slight advantage for hand coding.

B. Simulation

Before experimental tests on the FPGA board, we simulated our two VHDL codes with GHDL and GTKWave (two free tools with linux). We built a testbench based on experimental profiles and compared the results to values given by the SPL algorithm. Both versions lead to correct results. Our first codes were highly optimized, indeed the pipeline could compute a new phase each 33 cycles and its latency was equal to 95 cycles. Since the Spartan6 is clocked at 100MHz, estimating the deflection of 100 cantilevers would take about $(95 + 200 \times 33).10 = 66.95\mu$s, i.e. nearly 15,000 estimations by second.

C. Bitstream creation

In order to test our code on the SP Vision board, the design was extended with a component that keeps profiles in RAM, flushes them in the phase computation component and stores its output in another RAM. We also added components that implement the wishbone protocol, in order to "drive" signals to communicate between i.MX and other components. It is mainly used to start to flush profiles and to retrieve the computed phases in RAM. Unfortunately, the first designs could not be placed and routed with ISE on the Spartan6 with a 100MHz clock. The main problems were encountered with series of arithmetic operations and more especially with RAM outputs used in DSPs. So, we needed to decompose some parts of the pipeline, which added few clock cycles. Finally, we obtained a bitstream that has been successfully tested on the board.

Its latency is of 112 cycles and it computes a new phase every 40 cycles. For 100 cantilevers, it takes $(112 + 200 \times 40) \times 10ns = 81.12\mu$s to compute their deflection. It corresponds to about 12300 images per second, which is largely beyond the camera capacities and the possibility to extract a new profile from an image every 40 cycles. Nevertheless, it also largely fits our design goals.

V. Conclusion and perspectives

In this paper we have presented a full hardware/software solution for real-time cantilever deflection computation from interferometry images. Phases are computed thanks to a new algorithm based on the least square method. It has been quantized and pipelined to be mapped into a FPGA, the architecture of our solution. Performances have been analyzed through simulations and real experiments on a Spartan6 FPGA. The results meet our initial requirements. In future work, the algorithm quantization will be better analyzed and an high speed camera will be introduced in the processing chain so that to process real images. Finally, we will address real-time filtering and control problems for AFM arrays in dynamic regime.

References

[1] Simulink HDL coder 2.1. Matworks datasheet, 2011.

[2] N. Abedinov, P. Grabiec, T. Gotszalk, T. Ivanov, J. Voigt, and I. W. Rangelow. Micromachined piezoresistive cantilever array with integrated resistive microheater for calorimetry and mass detection. *Journal of Vacuum Science and Technology A*, 19(6):2884–2888, Nov 2001.

[3] D. R. Baselt, B. Fruhberger, E. Klaassen, S. Cemalovic, C. L. Britton, S. V. Patel, T. E. Mlsna, D. McCorkle, and B. Warmack. Design and performance of a microcantilever-based hydrogen sensor. *Sensors and Actuators B: Chemical*, 88(2):120–131, Jan 2003.

[4] M. Favre, J. Polesel-Maris, T. Overstolz, P. Niedermann, S. Dasen, G. Gruener, R. Ischer, P. Vettiger, M. Liley, H. Heinzelmann, and A. Meister. Parallel afm imaging and force spectroscopy using two-dimensional probe arrays for applications in cell biology. *Journal of Molecular Recognition*, 24(3):446–452, 2011.

[5] H. Hui, Y. Yakoubi, M. Lenczner, and N. Ratier. Semi-decentralized approximation of a lqr-based controller for a one-dimensional cantilever array. In *18th World Congress of the International Federation of Automatic Control (IFAC)*, 2011.

[6] M. Lenczner, N. Ratier N, E. Pillet, S. Cogan S, H. Hui, and Y. Yakoubi. *NanoSystems & Systems on Chips, Modeling, Control and Estimation*, chapter Modelling, Identification and Control of a Micro-cantilever Array. John Wiley & Sons, 2010.

[7] Hitesh Patel. Unlock new levels of productivity for your design using ISE design suite 12. Xilinx White paper, May 2010.

[8] M. B. Sinclair, M. P. de Boer, and A. D. Corwin. Long-working-distance incoherent-light interference microscope. *Applied Optics*, 44(36):7714–7721, Dec 2005.