# A cost effective AFM setup, combining interferometry and FPGA

Raphaël Couturier*, Stéphane Domas*, Gwenhaël Goavec-Merou†,
Mélanie Favre‡, Michel Lenczner† and André Meister‡
*FEMTO-ST, DISC
University of Franche-Comté, Belfort, France
{raphael.couturier,stephane.domas}@univ-fcomte.fr
†FEMTO-ST, Time-Frequency
University of Franche-Comté, Besançon, France
michel.lenczner@utbm.fr,gwenhael.goavec@trabucayre.com
‡CSEM
Neuchatel, Switzerland
{melanie.favre,andre.meister}@csem.ch

**Abstract**

Atomic force microscopes (AFM) provide high resolution images of surfaces. In this paper, we focus our attention on an interferometry method for the deflection estimation of cantilever arrays in quasi-static regime. Here, we propose a novel complete solution with a least square based algorithm to determine interference fringe phase and its optimized FPGA implementation. Simulations and real tests show very good results and open perspectives for real-time estimation and control of cantilever arrays in the dynamic regime.

## 1. 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 [1], 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 [2], 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 [3], 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 decomposes into two parts: a calibration phase that uses the algorithm based on splines presented in [3], and an acquisition loop that relies on a fast method based on least squares and its FPGA implementation.

The remainder of the paper is organized as follows. Section 2 describes the goals we chose for our setup. Our solution based on the FPGA implementation of a least square method is presented in Section 3. Numerical experimentations are described in Section 4. Finally a conclusion and some perspectives are drawn.

## 2. Design goals and choices

In order to build simple, cost effective and user-friendly cantilever arrays, we use a system based on interferometry. The experimental setup is described in [4]. It is based on a Linnick interferometer [5] to produce interferometric fringes on the sample surface. The sample is itself placed on a nano-positioning table. A CMOS camera takes images of the surface, that are analyzed to recover the cantilever deflections.

Current experiments operate on arrays with up to $17 \times 4 = 68$ levers but it should increase in a near future. Thus, we chose a goal which consists in designing a computing unit able to estimate the deflections of at least 100 cantilevers, for an image rate of 1KHz. The geometry of the array of cantilevers is not fixed but our solution should manage a grid, for example $10 \times 10$, $7 \times 15$, $4 \times 25$, $\ldots$

In addition, the result accuracy must be close to 1nm, the maximum precision reached in [3]. 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 requirements are stated in the perspective of implementing real-time active control for each cantilever (see [6], [7]).

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 100 cantilevers, our method implies to compute the phase of interferometric fringes located on 201 segments of pixels (i.e. *profiles*, see Section 3). Supposing the camera is able to pick images every milliseconds, each phase calculation (including the time to extract pixels) should take no more than $5\mu s$.

In fact, this timing is a very hard constraint. A very simple test-bench that does cumulated sums on 20 values (the average profile size in our experiments) reaches an average of 155Mflops on an Intel Core 2 Duo E6650 at 2.33GHz. 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. This test-bench implies that the phase computation algorithm should do less than 775 floating point instructions, which is very small.

However, the most important point is the latency of the whole computation. Supposing that each profile must be treated one after the other in $5\mu s$, the deflection of 100 cantilevers would take $1ms$. It is totally 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 profiles in GPU memory and of taking back results would be prohibitive compared to computation time. It is far more efficient to pipeline the computation. For example, supposing that 200 profiles of 20 pixels could be pushed sequentially in a pipelined unit clocked 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 accurate for real-time control. FPGAs are appropriate for such an implementation, so they turn out to be the computation units of choice to reach our goals.

## 3. Interferometric based cantilever deflection estimation

As shown in Figure 1, 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 one (reference profile) is on the base of the array. It is common to all levers and provides a reference for noise suppression. The second is located just above the AFM tip (tip profile), it provides the phase shift modulo $2\pi$. The third one is close to the base junction (base profile). It is used to determine the "real" tip phase through an operation called unwrapping where it is assumed that the average deflections along the base and tip profiles are linearly dependent. Finally, deflections are simply derived from this unwrapped phase.

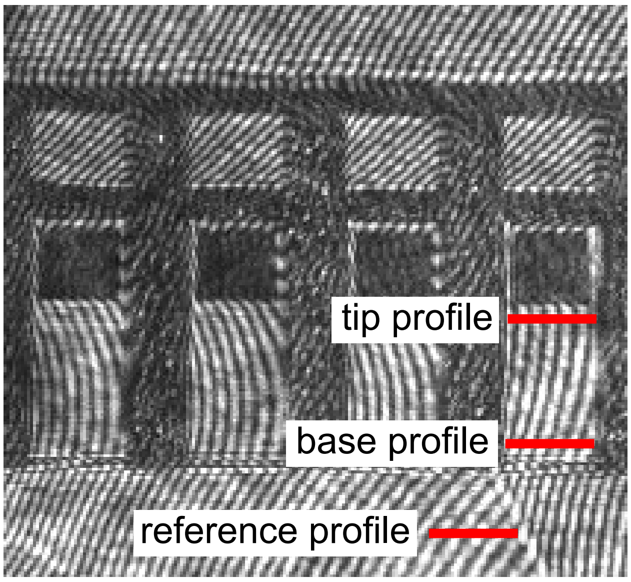

Figure 1. Portion of a camera image showing moving interferometric fringes on a $17 \times 4$ array of cantilevers

The pixel gray-level intensity $I$ of each profile is modeled 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 $\theta$ 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.

Our method consists in two main sequences: *calibration* and *acquisition loop*. The calibration does not require real-time computations. First, an image of the whole array is taken, without any forces applied to levers. From profile locations, we determine the frequency $f$ of each profile using a spline interpolation, described in Section 3-A. Then, we take several images for different locations of the nano-positioning table in order to compute, for each lever, the coefficients of the linear relation between base and tip phases. And finally, we find the slope of the array compared to the horizontal plane.

The *acquisition loop* consists in moving the nano-positioning table and taking images at regular time steps. For each image, the phase $\theta$ of each profile is computed to obtain, after unwrapping, the cantilever deflection. The phase determination is achieved on the FGPA, with the least square method described in Section 3-B.

### A. Frequency determination

As mentioned above, the phase needs to be computed for each image but $f$ is computed only once during calibration. This is why the frequency determination can be done on a CPU with a time consuming method based on spline interpolation.

We denote by $M$ the number of pixels in a segment (i.e. a profile) used for phase computation. For the sake of the simplicity of the notations, we consider the light intensity $I$ a function on the interval $[0, M-1]$ 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 coordinates $x^p \in \{0, 1, \ldots, M-1\}$. 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 interpolating $I$ at a larger number $L = 1 + k \times (M-1)$ of points (typically $k = 4$ is sufficient) in the interval $[0, M-1]$. We denote these points $x_i^s$ with $i = 0, 1, \ldots, L-1$.

The second step is to determine the affine part $ax + b$ of $I$. It is found with an ordinary least square method, taking into account the $L$ points. Values of $I$ in $x_i^s$ are used to compute its intersections with $ax + b$. The period of $I$ (and thus its frequency) is deduced from the number of intersections and the distance between the first and the last.

The frequency could also be obtained using the derivative of spline functions, which only requires to solve quadratic equations but yields higher errors.

## *B. Phase determination*

Since $f$ and $x$ are already known, Equation (1) has only 4 parameters: $a, b, A$, and $\theta$. 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 $\theta$ only. Firstly, the affine part $ax + b$ is estimated from the $M$ values $I(x^p)$ to determine the corrected intensities,

$$I^{corr}(x^p) \approx I(x^p) - a.x^p - b. \tag{2}$$

To find $a$ and $b$ we apply the least square method (as in SPL but on $M$ points). Let $X^p$ be a vector containing the values of $x^p$, then

$$a = \frac{\mathrm{covar}(X^p, I(X^p))}{\mathrm{var}(X^p)} \text{ and } b = \overline{I(X^p)} - a.\overline{X^p}. \tag{3}$$

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

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

Finally, the problem of approximating $\theta$ 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 $\theta^*$ of the minimization problem is a zero of the first derivative of the above argument,

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

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

Several points can be noticed:

- If all profiles have the same fixed size, then $\overline{X^p}$ and $\mathrm{var}(X^p)$ are constants.
- 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 $\theta$, they can be precomputed.
- Lookup tables 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 $\theta^*$ of the optimality condition is to discretize the range $[-\pi, \pi]$ with a large number $nb_s$ of points and to find which one is a minimizer in the absolute value sense. Hence, three other lookup tables can be set with the $3 \times nb_s$ values $\sin\theta$, $\cos\theta$, and
$$\left[ \cos 2\theta \sum_{i=0}^{M-1} \sin(4\pi f i) + \sin 2\theta \sum_{i=0}^{M-1} \cos(4\pi f i) \right].$$
- The search algorithm can be very fast using a dichotomous process namely in $log_2(nb_s)$.

Taking into account all these remarks, we obtain an algorithm (called LSQ in the following) that uses no complex operations (division, trigonometric functions, ...) which is mandatory to be ported efficiently on an FPGA. Nevertheless, it should also be proved that it fits with our goals.

## *C. LSQ evaluation*

We evaluated the algorithm over 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, 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 $\frac{\pi}{nb_s}$. In our last experiments on a $17 \times 4$ 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 1nm, the precision to reach.

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 distortions. We report percentages of errors between the reference and the computed phases out of $2\pi$,

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

Table 1 gives the maximum and the average errors for both algorithms and for increasing values of $N$ the noise parameter.

| | LSQ | |
|---|---|---|
| noise (N) | max. err. | aver. err. |
| 0 | 0.49 | 0.1 |
| 2.5 | 1.16 | 0.22 |
| 5 | 2.47 | 0.41 |
| 7.5 | 3.33 | 0.62 |
| 10 | 4.29 | 0.81 |
| 15 | 6.35 | 1.21 |
| 30 | 13.94 | 2.45 |

Table 1. Error (in %) for cosines profiles, with noise.

The results show that the algorithm 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 still under the precision to reach.

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 the $17 \times 4$ array mentioned above allowed us to compare experimental profiles to simulated ones. We can see on figure 2 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 3 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 of the possible error.

The second criterion is relatively easy to estimate. The number of operation is proportional to $M$ the number of pixels. It also depends on $nb_s$. 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 LSQ algorithm in C code, we obtain about 430 operations, which is largely under the limit of 775 (see section2). 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

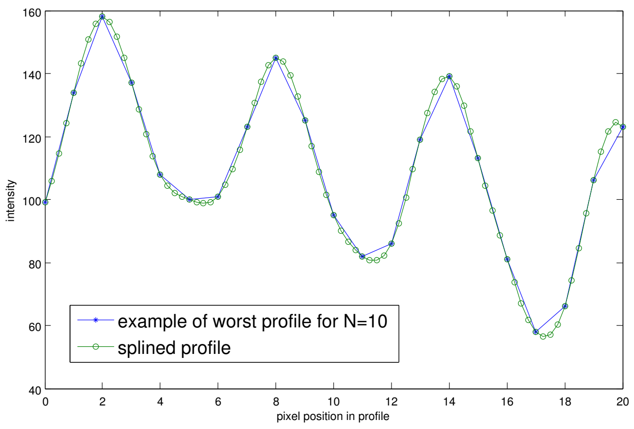

Figure 2. Sample of worst profile for N=10

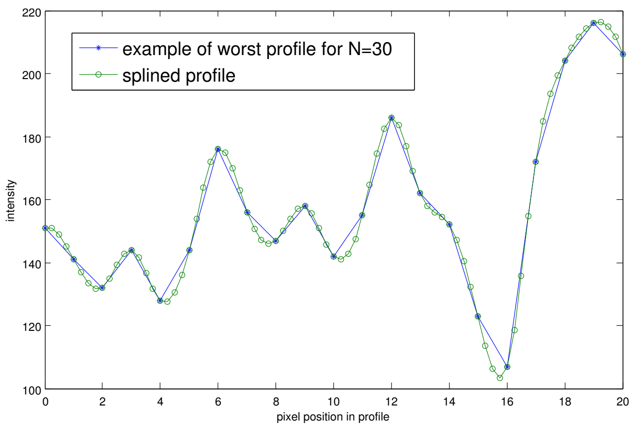

Figure 3. Sample of worst profile for N=30

type of operations but also on 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 alter 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, the hardest constraint does not come from the FPGA characteristics but from the algorithm itself. Its VHDL implementation can be efficient only if it can be fully (or near) pipelined. We observe that it is not a problem with LSQ since all parts, except the dichotomic search, work on the same data with a constant size: the profile.

## 4. VHDL implementation and experimental tests

### A. 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 $2 \times 17$ headers that can be used for any purpose. In our setup, they are connected to an interface board that is bound to the camera and the nano-positioning table controller.

### B. 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 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 chose 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 mentioned above, some operations like divisions must be avoided. 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, $\text{var}(X^p)$ is known and fixed. Thus, $\frac{\text{covar}(X^p, I(X^p))}{\text{var}(X^p)}$ can be replaced by

$$(\text{covar}(X^p, I(X^p)) \times \left\lfloor \frac{2^n}{\text{var}(X^p)} \right\rfloor) \gg n$$

where $\gg$ is a right bit-shift and $n$ 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 FPGAs.

We built two versions of VHDL codes, namely one directly by hand coding and the other with Matlab using the Simulink HDL coder feature [8]. 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.

### C. 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 test-bench based on experimental profiles and compared the results to values given by the C implementation. 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.

### D. Complete design

Figure 4 shows the complete processing chain, including FPGA components, the i.MX processor, and external devices. The main interface between the i.MX and the FPGA is a VHDL component called *interconnector*, that implements the wishbone protocol. It is in charge of relaying data from/to i.MX to/from other components. It is mainly used before the acquisition loop to fill the precomputed values LUTs, to send profile positions and identifiers to the *profile extractor* component, and to setup the sweep parameters to *Table move*. It also relays orders like to begin profile extraction.

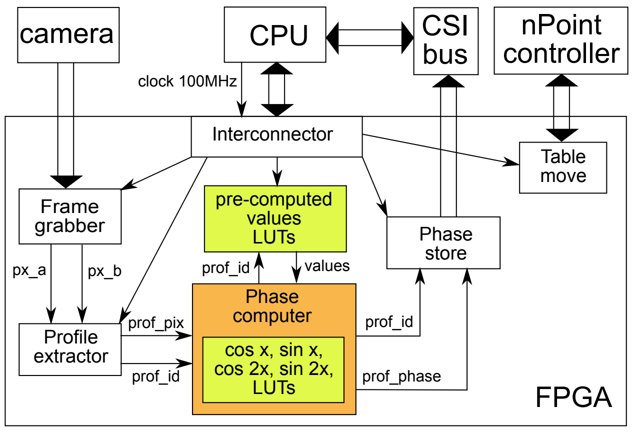

Figure 4. Overall design

The *table move* component is directly connected to the nPoint controller that drives the nano-positioning table. It can be programmed to achieve two types of scan: a sequence of round trips in the x-y dimension or a simple round trip in the z dimension. The first one is mainly used to obtain topographic informations and the second one a force curve. For each type, the user can specify the size of the steps in space and time and their number.

The *phase store* component is connected to a specific bus on the board, called CSI bus. Usually, this bus is used to grab video frames from little CMOS camera at high rate (up to 60 MHz). We use it to flush the phases of all profiles, each time an incoming image has been totally processed. It is the most efficient way to send the phases to the CPU.

It should be noticed that unwrapping the tip phase is done on the CPU since it is not time consuming and quite complex to implement on the FPGA.

The whole source code represents 9,300 lines of VHDL (without comments) but nearly 3,800 were generated by ISE CoreGen® to implement wrappers for RAMs and DSPs. The *phase computer* which is the heart of the design consists in 2,200 lines of code, highly optimized. It perfectly illustrates the complexity to transpose the 150 lines of LSQ written in C into VHDL.

### *E. Bitstream creation and tests*

Unfortunately, the version of *phase computer* used during simulations led to a design that 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. The distance between some RAMs and DSPs, combined to the time needed to achieve the multiplication led to a signal propagation that lasts longer than the clock cycle. We solve the problem by inserting a delay after the RAMs that were in this case.

Finally, we obtained a *phase computer* with a latency of 128 cycles and that can compute a new phase every 60 cycles. For 100 cantilevers, it would take $(128 + 201 \times 60) \times 10ns = 121.88\mu s$ to compute their deflection. It corresponds to about 8,200 estimations per second, which is largely beyond our camera capacities and the possibility to extract a new profile from an image every 60 cycles. Nevertheless, it also largely fits our design goals.

## 5. Conclusion and perspectives

In this paper we have presented a full hardware and 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 heart of our solution. Performances have been analyzed through simulations and real experiments on a Spartan6 FPGA. The results meet our initial requirements in terms of throughput (at least 100 deflection estimations every millisecond) and precision (1nm). The next step is to finalize the setup, integrating the camera and the nano-positioning table for the hardware part, and implementing components to drive them for the software part.

Future works will also address more general problems such as algorithm quantization and real-time filtering or control for AFM arrays in dynamic regime.

## 6. Acknowledgements

This work is partially supported by the European Territorial Cooperation Programme INTERREG IV A France-Switzerland 2007-2013 and by the French National Programme LABEX ACTION.

## References

[1] 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.

[2] 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.

[3] 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.

[4] A. Meister, Gruner G., and J. Polesel-Maris. Brevet ep2336789 (a1) : Parallel cantilever deflection measurment. 2012.

[5] 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.

[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] 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.

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