# Fast fixed-point bicubic interpolation algorithm on FPGA

# Fast Fixed-point Bicubic Interpolation Algorithm on FPGA

1st Janne Koljonen
*School of Technology and Innovations*
*University of Vaasa*
Vaasa, Finland
https://orcid.org/0000-0001-5834-4437

2nd Vladimir A. Bochko
*School of Technology and Innovations*
*University of Vaasa*
Vaasa, Finland
https://orcid.org/0000-0002-3505-3677

3rd Sami J. Lauronen
*School of Technology and Innovations*
*University of Vaasa*
Vaasa, Finland
https://orcid.org/0000-0002-3767-045X

4th Jarmo T. Alander
*School of Technology and Innovations*
*University of Vaasa*
Vaasa, Finland
https://orcid.org/0000-0002-7161-8081

***Abstract*—We propose a fast fixed-point algorithm for bicubic interpolation on FPGA. Bicubic interpolation algorithms on FPGA are mainly used in image processing systems and based on floating-point calculation. In these systems, calculations are synchronized with the frame rate and reduction of computation time is achieved designing a particular hardware architecture. Our system is intended to work with images or other similar applications like industrial control systems. The fast and energy efficient calculation is achieved using a fixed-point implementation. We obtained a maximum frequency of 27.26 MHz, a relative quantization error of 0.36% with the fractional number of bits being 7, logic utilization of 8%, and about 30% of energy saving in comparison with a C-program on the embedded HPS for the popular Matlab test function Peaks(25,25) data on SoCkit development kit (Terasic), chip: Cyclone V, 5CSXFC6D6F31C8. The experiments confirm the feasibility of the proposed method.**

***Index Terms*—control, fixed-point algorithm, bicubic interpolation, FPGA, energy efficiency**

## I. Introduction

Interpolation is widely used in different areas of engineering and science particularly for image generation and analysis in remote sensing, computer graphics, medicine, and digital terrain modelling [1–4]. The most popular methods in digital image scaling are nearest neighbor and bilinear interpolation. However, nearest neighbor interpolation has stairstepping on the edges of the objects while bilinear interpolation produces blurring [3]. Bicubic interpolation is in turn slightly more computationally complicated but has a better image quality.

FPGA based real-time super-resolution is introduced in [5] where the FPGA based system reduces motion blur in images. The fisheye lens distortion correction system based on FPGA with a pipeline architecture is proposed in [6]. The FPGA-based fuzzy logic system is utilized in image scaling [7]. The architecture is based on pipelining and parallel processing to optimize computation time. A bilinear interpolation method for FPGA implementation has been used to improve the quality of image scaling [8]. For preprocessing purposes sharpening and smoothing filters are adopted followed by a bilinear interpolator. The adaptive image resizing algorithm is verified in FPGA [9]. The architecture consists of several stage parallel pipelines.

Implementations of bicubic interpolation using FPGA for image scaling [10, 11] usually use floating-point arithmetic. In [11], the floating-point multiplication is replaced by a look-up table method and convolution designed using a library of parameterized modules. These methods deal with a batch of data, i.e. all image frame pixels are available concurrently, and the purpose is to provide real-time video-processing at image frame rate.

Our task is different, as the goal includes also a high-speed industrial control applications, where fast-rate data sequentially arrive from sensors and the interpolated control data has to be sent to the actuators within low latency delay that can only be achieved using FPGA or ASIC. Our control system is similar to the look-up table implementations of fuzzy controllers, e.g. [12]. In real-time applications, it is computationally efficient to implement the nonlinear control surface as a (possibly multi-dimensional) look-up table, which is obtained by spatial sampling from the continuous control surface. The control output samples can use either floating or fixed-point representation. Subsequently, the interpolated control outputs between the sample grid points can be computed in runtime.

In contrast to the studies presented in [10, 11] we implement the interpolation algorithm using fixed-point arithmetic. The objective is to obtain accurate data quantization working with the same rate as the data arrives. Obviously, the use of fixed-point numbers introduces round-off errors at several phases: quantization of measurements, sampling, and in internal calculations. The benefit of fixed-point algorithms include reduced complexity of the logic and, subsequently, a higher operating frequency.

This study was supported by the Academy of Finland (project SA/SICSURFIS). 978-1-7281-2769-9/19/$31.00 ©2019 IEEE

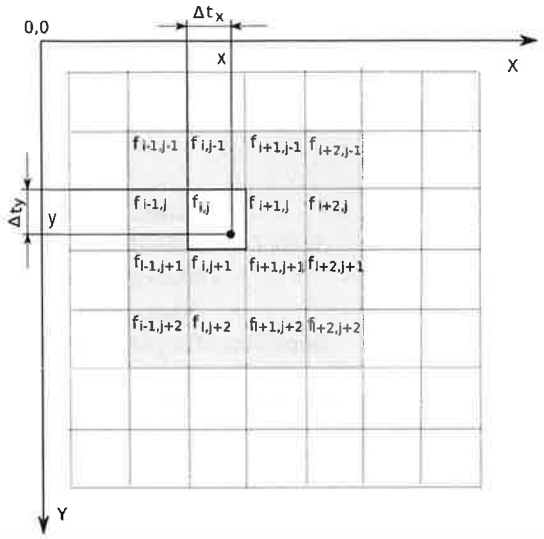


Fig. 1. Notations used in bicubic interpolation. Note the convention of image processing for y-axis towards line.

As for fixed-point implementations there are several competitive optimization objectives. On one hand, the quantization error should be minimized. On the other hand, the resource use and latency time should be minimized and the throughput maximized. One solution is to find a suitable wordlength to serve all the objectives reasonably well. Additionally, the internal arithmetic can be implemented smartly: avoiding complex arithmetic and using, e.g., additions and shift operations instead, and using the potential of VHDL language to define custom data types with only the required number of bits can result in significant savings in resources. This makes the fixed-point calculation a demanding problem when implementing in FPGA.

Reference [13] defines two main methods to optimize the wordlength as for fixed-point computations. First, the fixed-point implementation can be compared to the equivalent floating-point system by simulation. Second, several analytical approaches can be used. We use the simulation approach.

## II. Bicubic Interpolation

The objective is to interpolate a two-dimensional function $F(x, y)$ defined on a regular rectangular grid (Fig. 1). The function values are known in the intersection points $(f_{i,j})$.

The point of interpolation $(x, y)$ is a function value down and to the right of a grid point $(f_{i,j})$ with a deviation $(\Delta t_x, \Delta t_y)$ from the previous grid points. For interpolating one point, $4 \times 4 = 16$ grid points plus the deviations $(\Delta t_x, \Delta t_y)$ are needed. This is a good example how we can trade between speed and resources with FPGAs: we can either compute the $i, \Delta t_x$ and $j, \Delta t_y$ in parallel to gain speed or sequentially in series to minimize hardware. In any case we can define a hardware module that does it for one dimension (using a fixed-point approach). Bicubic spline interpolation requires the solution of a linear system, described in [14], for each grid cell. An interpolator with similar properties can be obtained by applying a convolution with the following kernel in both dimensions:

$$W(x)=\begin{cases}(a+2)|x|^3-(a+3)|x|^2+1 & \text{for } |x|\leq 1,\\ a|x|^3-5a|x|^2+8a|x|-4a & \text{for } |x|<2,\\ 0 & \text{otherwise,}\end{cases} \quad (1)$$

where $a$ is usually set to $-0.5$ or $-0.75$. Note that $W(0) = 1$ and $W(n) = 0$ for all nonzero integers $n$. Keys, who showed third-order convergence with respect to the sampling interval of the original function, proposed this method [14].

If we use the matrix notation for the common case $a = -0.5$, we can express the equation as follows:

$$p(t) = \frac{1}{2}\begin{bmatrix}1 & \Delta t & \Delta t^2 & \Delta t^3\end{bmatrix}\begin{bmatrix}0 & 2 & 0 & 0\\ -1 & 0 & 1 & 0\\ 2 & -5 & 4 & -1\\ -1 & 3 & -3 & 1\end{bmatrix}\begin{bmatrix}f_{-1}\\ f_0\\ f_1\\ f_2\end{bmatrix}, \quad (2)$$

for $\Delta t \in [0, 1)$ for one dimension. Note that for 1-dimensional cubic convolution interpolation requires four sample points. For each inquiry two samples are located to the left and two to the right from the point of interest. These points are indexed from $-1$ to 2 in this paper. The distance from the point indexed with 0 to the inquiry point is denoted by $\Delta t$ here.

For a point of interest in a 2D grid, interpolation is first applied four times in $x$ and then once in $y$ direction:

$$\begin{aligned}
b_{-1}&=p(\Delta t_x, f_{(i-1,j-1)}, f_{(i,j-1)}, f_{(i+1,j-1)}, f_{(i+2,j-1)}),\\
b_0&=p(\Delta t_x, f_{(i-1,j)}, f_{(i,j)}, f_{(i+1,j)}, f_{(i+2,j)}),\\
b_1&=p(\Delta t_x, f_{(i-1,j+1)}, f_{(i,j+1)}, f_{(i+1,j+1)}, f_{(i+2,j+1)}),\\
b_2&=p(\Delta t_x, f_{(i-1,j+2)}, f_{(i,j+2)}, f_{(i+1,j+2)}, f_{(i+2,j+2)}),\\
p(y,x)&=p(\Delta t_y, b_{-1}, b_0, b_1, b_2),
\end{aligned} \quad (3)$$

The size of the data matrix $f$ is denoted by $s_x \times s_y$. To enable interpolation also at the edge points we extend the data to the top and left margins by repeating data from the top row and the left column, respectively, and to right and bottom margins by repeating twice the right column and the bottom row, respectively. Thus, the size of the extended matrix is $s_{xe} \times s_{ye} = (s_x + 3)(s_y + 3)$.

## III. Fixed-point Numbers and Arithmetic

We use $Q_{m,n}$ numbers to define the $m$ integer and $n$ fractional bits for the fixed-point approach. The fractional part determines the interpolation and quantization resolutions, i.e. the interval between two consecutive numbers or interpolated points. This is defined as $|\Delta t_k - \Delta t_{k-1}| = 2^{-n}$. In general, the data range determines the number of integer bits needed. In particular purposes $m$ is as follows: The value $m$ is determined using the absolute maximum value of the given data set $f$.

In addition, from (2) we note that $\Delta t < 1$, and the absolute values of the matrix entries are integers in the range $[0, 5]$. Multiplication by 2 and 4 can be replaced by left shifts. Due to the fact that entries 3 and 5 can be decomposed to $(2 + 1)$ and $(4 + 1)$, respectively, multiplication by 3 and 5 can be replaced by left shifts and one summation.

Finally, we assume that value $m$ is defined by a number of bits representing the absolute maximum value of $f$ shifted


Fig. 2. The HPS-FPGA interaction scheme. The HPS does data preprocessing, testing and reporting. The fixed-point algorithm is implemented in FPGA.

left twice. The given data is positive and negative. Therefore, signed decimal numbers are used and, thus, a sign bit is also needed. The wordlength for $f$ is $m+n+1$.

The corresponding wordlengths for $x$ and $y$ are $m_x+n+1$ and $m_y+n+1$, where $m_x$ and $m_y$ are the least number of bits needed to represent the data matrix $f$ size $s_{xe}$ and $s_{ye}$, respectively.

### A. Fixed-point Implementation in VHDL

We could use a fixed-point package for modeling [15]. However, this package may not be available for electronic design automation tools needed for programming design functionality in FPGA. In addition, bicubic interpolation includes arithmetic operations avoiding multiplication, division and other time and resource consuming operations, which simplify the design for fixed-point calculations. Therefore, we model the fixed-point numbers and arithmetic directly in VHDL.

We use both simulation and a Hard Processor System (HPS-FPGA) scheme in the implementation and testing (Fig. 2). The software of the HPS performs preprocessing of input data needed for the fixed-point algorithm. We use Python program for preprocessing the data. The original data have floating-point coordinates in the range $[-a,a]$ for $x$ and $[-b,b]$ for $y$. The HPS translate these values by adding $a+1$ and $b+1$ to $x$ and $y$, respectively, to make them positive values in the range $[1,s_x]$ and $[1,s_y]$ that are, subsequently, suitable for separating into integer and fractional parts. In addition, we multiply their values by $2^n$ to convert them to fixed-point numbers. After preprocessing, input data $(x,y)$ are sent to the FPGA. The output of the FPGA is an interpolated value read back to the HPS. The HPS divides the interpolated values by $2^n$ to convert them back the floating-point values. We do not delegate preprocessing to FPGA since the focus of the study is on interpolation and the original data are not necessary floating-point values.

We implemented the fixed-point algorithm in VHDL for the FPGA. The dataflow for the bicubic interpolation includes: extractor of integer and fractional part, convolution, dot product and output register (in Fig. 3).

For VHDL the input is $(x,y)$ (Fig. 3). First, component Bicubic interpolation calculates the integer and fractional parts of the input. The integer part gives indexes $(i,j)$ of matrix $f$. The matrix $f$ is implemented as a VHDL 2D array in a package (fixed control surface). The fractional part defines $(\Delta t_x, \Delta t_y)$. This information is used to calculate convolution according to (3). We have 4 ($b_{-1}$–$b_2$) of 5 convolution operations implementing in parallel. Component Convolution calculates the product between the matrix and vector containing $f$ values of (2) to obtain a weighted composition of values $f$ and, then, passes the result to component Dot product to

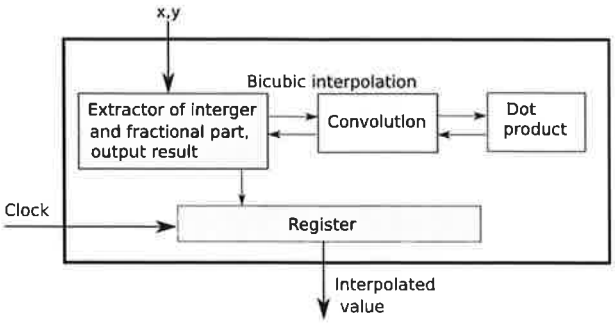


Fig. 3. The dataflow for bicubic interpolation.

calculate the dot product of the weighted composition and the vector containing $\Delta t$ and its powered values.

When the weighted composition is determined, all multiplications are replaced by summations and shifting to accelerate the calculation. The other arithmetical operations are as follows:

- VHDL package *numeric_std* provides summation/subtraction of signed integer numbers [16].
- Multiplication/division by a factor $2^k$, where $k=1,2,,$ is replaced by a bit shift.
- The left shift for the negative and positive numbers was implemented keeping the sign bit, shifting all bits to the left, removing the MSB and adding 0 to the LSB.
- The right shift for the positive numbers was implemented keeping the signed bit, shifting all bits to the right, inserting 0 to the MSB and removing the LSB. The right shift for the negative numbers was implemented keeping the signed bit, shifting all bits to the right, inserting 1 to the MSB and removing the LSB. The difference in shifting is because the negative numbers have a complement form.
- VHDL package *numeric_std* provides multiplication of signed decimal numbers in component Dot product. The result of multiplication if both operands have the same format is: two (repeated) sign bits, $2m$ integer bits, $2n$ fractional bits. We denote the length of the word without the sign bits with four parts: $m'+m''+n'+n''$ ($m'=m''$ and $n'=n''$). To convert the result to the format of the operand, one has to keep one (any) sign bit, and $m''+n'$ bits.

We do not use hardware multipliers, because we use variable wordlength. This gives more flexibility to scale up the design for any number of bits. Shifting is simply by array indices, therefore DSP logic is not needed.

### B. Fixed-point Implementation in Matlab

For verification, we implemented floating-point and fixed-point algorithm variants in Matlab. For fixed-point we use the same $Q_{m,n}$ numbers and the Matlab integer data type with 32 bits (int32). The arithmetic operations for the fixed-point algorithm are as follows:

- Matlab supports summation and subtraction of the integer numbers.

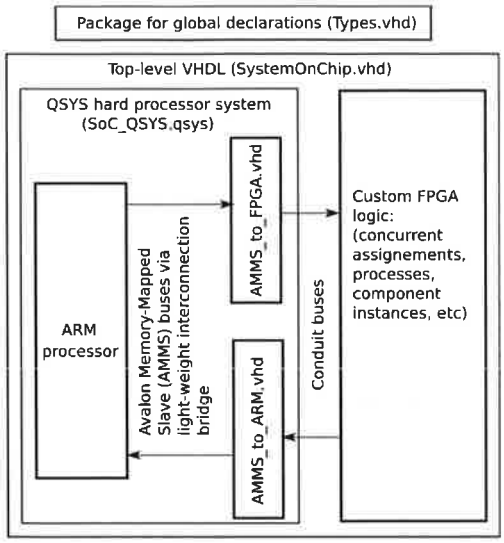


Fig. 4. Data flow between ARM and FPGA. Notations: Avalon Memory Mapped Slave (AMMS), System on Chip (SoC), and System Integration Tool (QSYS).

- Multiplication of variables by factors or variables was made by converting the decimal numbers to the integer 64-bit format and then the result was multiplied by $2^n$, respectively, and converted back to the 32-bit format.
- Matlab provides division by a factor of 2.

## IV. Synthesis Using HPS and FPGA

For synthesis we use the TerasicAltera SoCKit development board combining HPS (800 MHz, A Dual-Core ARM Cortex™ - A9 MPCoreTM Processor) and FPGA (Cyclone V, 5CSXFC6D6F31C6). This Section includes the description of the interface between HPS and FPGA, method to establish a communication between HPS and FPGA, and the C language program to access FPGA.

### A. Interface between HPS and FPGA

The interface establishes a communication between ARM and FPGA. The dataflow diagram of the interface is given in Fig. 4. The interface consists of: the ARM processor (HPS), where software code is written, compiled, and run, Avalon Memory Mapped Slave (AMMS) interfaces from HPS to FPGA and FPGA to HPS. Avalon buses are Intel's denitions for a few general purpose buses. In this study, they are used to synchronously transfer data from HPS to FPGA and from FPGA to HPS. As both buses are slave buses, it implies that HPS is the master, i.e., data is transferred only when the software-side requests so.

The ARM processor and the AMMS buses are instantiated and integrated in QSYS (Intel). Inside QSYS systems, Avalon buses are usually used in communication. Intel also provides the possibility to use arbitrary buses. These are called conduits, which may be useful in communication between a QSYS system and custom FPGA logic that does not support Avalon buses. As the custom FPGA logic, our fixed-point bicubic interpolation parallel arithmetic operations with signed integers are implemented. The top-level entity includes: ports to the outside of the SoC (System on Chip) chip, an instance of the QSYS system, and possible instances of the custom FPGA logic components. To make the code more readable and the integration and parametrization of different parts simpler, a VHDL package to define custom global signal types and constants is also declared.

### B. Access to FPGA

From HPS, the Avalon buses are seen as memory-mapped IOs. For this low-level memory access a program written in C is used. Its purpose is to write the $x$ and $y$ coordinates to two memory addresses of the lightweight bridge, and then read the result from another address. The read function can be called immediately after calling the write function, because the FPGA calculates the result with a time, which is less than the delay between the two function calls. Before using the write and read functions of the program, the initialization function maps the memory addresses of the lightweight bridge into the process memory, so that these addresses can be used later.

## V. Experiments

We conducted experiments to study the quantization error, complexity, speed and power/energy consumption of the proposed algorithm. We implemented the floating-point and fixed-point algorithms in Matlab and fixed-point algorithm in VHDL. The floating point algorithm (Matlab) was used for the analysis of fixed-point finite wordlength errors in Matlab and FPGA. For simplicity, we will call finite wordlength errors caused by quantization of signals, roundoff errors occurring at arithmetic operations and quantization of constants as a quantization error.

### A. Input data and wordlength

For testing we choose a well-known Matlab data generated by the function Peaks(25,25) [17]. The function generates a mixture of 2-D Gaussians. The data matrix size is $25 \times 25$. Thus the range of $x$ and $y$ is $[1, 25]$ and translation is not needed. The original Peaks(25,25) values are multiplied by 30. This gives a data range $[-189.79, 239.89]$.

According to our generalized wordlength representation (Section 3) we suppose to work with *signed* $Q_{10,7}$ numbers for $f_{(i,j)}$ and *unsigned* $Q_{5,7}$ for $x$ and $y$. Given the $Q_{m,n}$ numbers Matlab automatically generates a VHDL package containing the constants determining the several wordlengths used in the fixed-point calculations. The HPS-FPGA scheme is used for calculation (Fig. 2). The input data represents coordinates $x$ and $y$. The HPS multiplies these values by $2^7$ for the fixed-point calculation. Finally, the HPS divides the interpolated value by $2^7$.

### B. Matlab Test

First, we implemented a floating-point algorithm in Matlab. To test it we generated a 3D surface using the given matrix $f$ (function Peaks(25,25) data) for interpolating and, then,

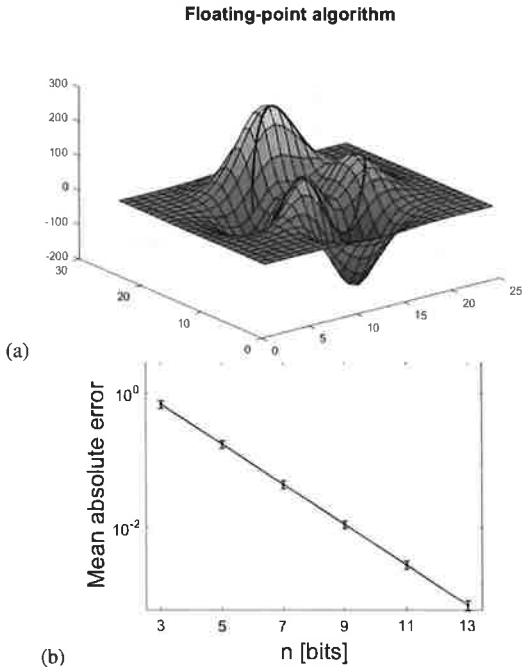


Fig. 5. a) Floating-point interpolation using Matlab. The circle with a radius 5 and center at (14,14) is projected onto the surface interpolating the input data (black curve). b) The mean absolute error (logarithmic scale) vs. the number of fractional bits n. The vertical error bars scaled by a factor of 4 for visualization show the confidence interval at level 0.95.

synthesized the projected circle with a radius 5, center located at $(14, 14)$. One can see the interpolation results in Fig. 5a.

Before FPGA implementation we tested the quantization error depending on the number of fractional bits $n$ at a confidence interval (CI) of 0.95 (Fig. 5b). Figure 5b shows that a reasonable choice for the number of bits is 7 that gives a relatively small quantitative error (mean absolute error of 0.044 at 95% CI[0.0014 0.073]).

### C. FPGA Test

The quantization error was calculated for 10,000 uniformly distributed random points. One set of interpolated points was determined using the floating-point Matlab algorithm. The other set of interpolated points was determined using the fixed-point algorithm on FPGA. Four quantization error metrics were used in comparisons: maximum absolute error (MAXAE), mean absolute error (MEANAE), median absolute error (MEDIANAE), and standard deviation (STD) at $n = 7$ (Tab. I). The relative error defined as the ratio of the maximum absolute error and the maximum absolute value of signal is 0.36% at $n = 7$.

TABLE I
FOUR QUANTIZATION ERROR METRICS

| *MAXAE* | *MEANAE* | *MEDIANAE* | *STD* |
|---|---|---|---|
| 0.87 | 0.08 | 0.03 | 0.13 |

The quantization error surface is shown in Fig. 6a. One can see that the quantization error is nonuniformly distributed upon the interpolated surface. To understand the error behavior we calculated the numerical gradient over the interpolated surface (Fig. 6b). Two plots (Fig. 6b, 6c) indicate that the quantization error increases with the increasing gradient.

Then, we calculated the gradient magnitude and mean absolute error over the interpolated surface (Fig. 6c). The mean absolute error for the data in each cell of the grid was calculated. The gradient magnitude is as follows:

$$G = \sqrt{(f'_x)^2 + (f'_y)^2}, \tag{4}$$

where $f'_x$ and $f'_y$ are numerical derivatives for $x$ and $y$ coordinates. It is clear that there is a reasonable linear dependence between the mean absolute error and gradient magnitude. The Pearson correlation coefficient is 0.42 that indicates a moderate positive relationship between mean absolute error and gradient magnitude. In addition, we measured the correlation coefficient for the slowly varying industrial application data set. The value measured was 0.8, i.e. a strong correlation. This is in accordance with the nature of bicubic interpolation, which well suits for smoothed data.

Timing analysis was implemented using TimeQuest Timing Analyzer (Intel). The solution was analyzed for delays in the digital circuit. To find the maximum clock frequency, the multi corner mode was utilized. The obtained result for bicubic interpolation is $F_{max} = 27.26$ MHz.

To estimate the complexity and logic utilization of the solution compilations with several system parameters were made (Tab. II). In this experiment, we varied $n$ the number of bits in the fractional part of $Q_{m,n}$ and monitored logic utilization, number of registers and DSP blocks. The results show the increase number of logic initialization and total registers with the increase of fractional bits while the number of DSP blocks are not changed.

TABLE II
COMPARISON WITH VARIED SYSTEM PARAMETERS. THE NUMBER OF DSP BLOCKS IS 25 (22%) FOR ALL CASES.

| *n bits of $Q_{m,n}$* | *n=3* | *n=5* | *n=7* | *n=9* | *n=11* | *n=13* |
|---|---|---|---|---|---|---|
| *Logic initialization* | 2,528 6% | 2,952 7% | 3,356 8% | 3,799 9% | 4,144 10% | 4,545 11% |
| *Total registers* | 14 | 16 | 18 | 20 | 22 | 24 |

Finally, we measured power and energy consumption with and without FPGA accelerator using the same SoC board (Fig. 7). For calculation, we utilized the same 10,000 uniformly distributed random points used in the quantization test. The measurements were made using the oscilloscope Agilent DSO-X 4024A (Tab. III).

Tests with the C-program running in HPS and the accelerated program using HPS-FPGA were run eight times each. We measured the static and dynamic parameters. Table III shows that the static power of HPS is higher than HPS-FPGA even though that depends on a number of active logical elements. The average dynamic power with the HPS only configuration is lower than with FPGA accelerator (0.28 W against 0.34 W). However, the computational time with HPS-FPGA is shorter

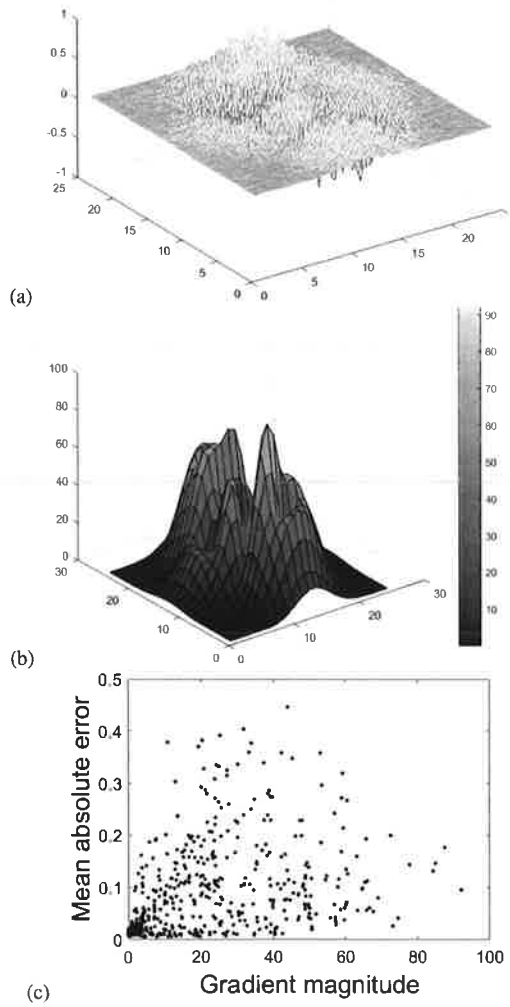


Fig. 6. a) Quantization error surface. b) The gradient over the interpolated surface. The highest values of gradient are shown by white color. c) Mean absolute error vs. gradient magnitude showing a moderate strength of relationship.

TABLE III
POWER (P) AND ENERGY (E) FOR HPS (C-PROGRAM) AND HPS-FPGA USING THE SAME SOC BOARD FOR EIGHT MEASUREMENTS. THE INDEX H STANDS FOR HPS AND F STANDS FOR HPS-FPGA.

| | *Parameter, rms* | *Average value and confidence interval* |
|---|---|---|
| Static | $P_H, W$ | 5.7, 95% CI[5.7, 5.7] |
| | $P_F, W$ | 5.46, 95% CI[5.46, 5.46] |
| Dynamic | $P_H, W$ | 0.28, 95% CI[0.259, 0.301] |
| | $P_F, W$ | 0.34, 95% CI[0.32, 0.36] |
| | $E_H, J$ | 0.19, 95% CI[0.169, 0.211] |
| | $E_F, J$ | 0.13, 95% CI[0.123, 0.137] |
| | $E_{saved}, \%$ | 31.57 |

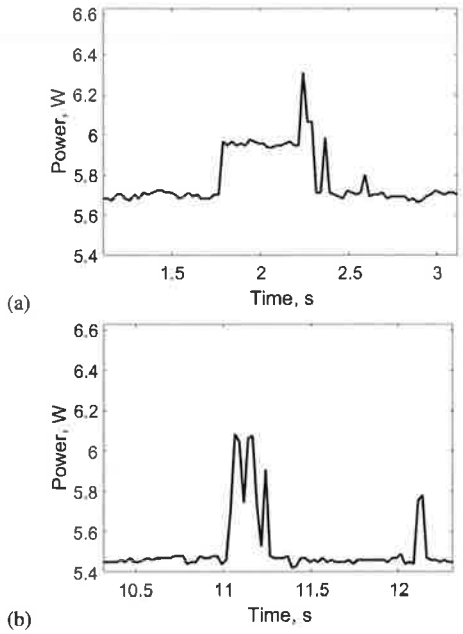


Fig. 7. Power oscillogram for HPS (a) and HPS-FPGA (b) (one measurement). The static power for HPS-FPGA is lower while the dynamic power is higher than for HPS. The HPS-FPGA computational time is shorter than HPS and as a result, the energy consumption is lower (31.57% less). The time discrete is 25 ms and the measurement time interval is 2 s.

(in average 59% of C-program time) and as a result, the total energy consumption is lower (31.57% less). We note that fixed costs due to reading and writing files and preprocessing the data reduce the total percentage saving of execution time and energy consumption.

## VI. CONCLUSIONS

In this paper, we proposed a hardware implementation of an accurate fixed-point bicubic interpolation intended for an industrial control system. The general recommendation for the wordlength selection depending on the input data format were given. In the experiments, we used *signed* $Q_{10,7}$ numbers for the interpolated values and *unsigned* $Q_{5,7}$ numbers for the input values. These values can be changed because the constants depending on these wordlength values are automatically calculated in Matlab for the VHDL package. The chosen $Q_{m,n}$ numbers for the input and output gave the relative quantization error of 0.36% and achieved 27.26 MHz frequency for function Peaks(25,25). The HPS-FPGA energy consumption was about 31% lower than when using a C-program only running in the same chip. The HPS-FPGA static power was 4.2% lower than when using the C-program.

In the future, we plan to implement fixed-point bicubic interpolation for images.

## ACKNOWLEDGMENT

We thank Markku Suistala from the Vaasa University of Applied Sciences, Finland, for the help in the FPGA energy measurements.

## References


[1] J. F.Hughes, A. Van Dam, J. D. Foley , M. McGuire, S. K. Feiner, and D. F. Sklar, Computer Graphics: Principles and Practice, Pearson Education, 2014.

[2] J. Garnero and D. Godone, "Comparisons between different interpolation techniques," The Role of Geomatics in Hydrogeological Risk, Padua, Italy, The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, vol. XL-5/W3, Feb. 2013, pp. 139–144.

[3] C. C. Lin, M. H. Sheu, H. K. Chiang, Z. C. Wu, J. Y. Tu, and C. H. Chen, "A low-cost VLSI design of extended linear interpolation for real time digital image processing," In 2008 International Conference on Embedded Software and Systems, July 2008, pp. 196–202.

[4] T. M. Lehmann, C. Gonner, and K. Spitzer, "Survey: Interpolation methods in medical image processing," IEEE Transactions on Medical Imaging, vol. 18, November 1999, pp. 1049–75.

[5] M.E. Angelopoulou, C. S. Bouganis, P.Y. Cheung, and G. A. Constantinides, "FPGA-based real-time super-resolution on an adaptive image sensor," In International Workshop on Applied Reconfigurable Computing, Springer, Berlin, Heidelberg, March 2008, pp. 125–136.

[6] N. Bellas, S. M. Chai, M. Dwyer, and D. Linzmeier, "Real-time fisheye lens distortion correction using automatically generated streaming accelerators," In 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines, April 2009, pp. 149–156.

[7] A. Amanatiadis, I. Andreadis, and K. Konstantinidis, "Design and implementation of a fuzzy area-based image-scaling technique," IEEE Transactions on Instrumentation and Measurement, August 2008, vol. 57, pp.1504–1513.

[8] N. Vidyashree and S. Usharani, "Implementation of image scalar based on bilinear interpolation using FPGA," IJARECE, June 2015, vol. 4, pp. 1620–1624.

[9] J. Xiao, X. Zou, Z. Liu, and X. Guo, "Adaptive interpolation algorithm for real-time image resizing," In First International Conference on Innovative Computing, Information and Control, Aug. 2006, vol. 2, pp. 221–224.

[10] M. A. Nuno-Maganda and M. O. Arias-Estrada. "Real-time FPGA-based architecture for bicubic interpolation: an application for digital image scaling," In 2005 International Conference on Reconfigurable Computing and FPGAs, Sep. 2005, pp. 8–pp.

[11] Y. Zhang, Y. Li, J. Zhen, J. Li, and R. Xie, "The hardware realization of the bicubic interpolation enlargement algorithm based on FPGA," In 2010 Third International Symposium on Information Processing, Oct. 2010, pp. 277–281.

[12] J. Jantzen, "Tuning of fuzzy PID controllers," Technical University of Denmark, report. 1998.

[13] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens, "A methodology and design environment for DSP ASIC fixed point refinement," In Design, Automation and Test in Europe Conference and Exhibition, Proceedings (Cat. No. PR00078), 1999, pp. 271–276.

[14] R. Keys, "Cubic convolution interpolation for digital image processing," IEEE Transactions on Acoustics, Speech, and Signal Processing, 1981, Vol. 29(6), pp.1153–1160.

[15] D. Bishop, "Fixed point package users guide," Packages and bodies for the IEEE, 2010, pp. 1076–2008.

[16] Doulos: https://www.doulos.com/knowhow/vhdl_designers_guide/numeric_std/, Last access: 14.05.2019.

[17] MathWorks: https://se.mathworks.com/help/matlab/ref/peaks.html, Last access: 22.05.2019.