ISBN: 978-3-95450-201-1 doi:10.18429/JACoW-IBIC2018-THOA01

LOW VS HIGH LEVEL PROGRAMMING FOR FPGA

Jan Marjanovic*, Deutsches Elektronen-Synchrotron DESY, Hamburg, Germany

Abstract

From their introduction in the eighties, Field-Programmable Gate Arrays (FPGAs) have grown in size and performance for several orders of magnitude. As the FPGA capabilities have grown, so have the designs. It seems that current tools and languages (VHDL and (System)Verilog) do not match the complexity required for advanced digital signal processing (DSP) systems usually found in experimental physics applications. In the last couple of years several commercial High-Level Synthesis (HLS) tools have emerged, providing a new method to implement FPGA designs, or at least some parts of it. By providing a higher level of abstraction, new tools offer a possibility to express algorithms in a way which is closer to the mathematical description. Such implementation is understood by a broader range of people, and thus minimizes the documentation and communication issues. Several examples of DSP algorithms relevant for beam instrumentation will be presented. Implementations of these algorithms with different HLS tools and traditional implementation in VHDL will be compared.

INTRODUCTION

According to [1], FPGAs have reached its fourth age. After the "Age of Invention", "Age of Expansion" and "Age of Accumulation" there are devices with enough capacity to include the entire system on a single chip [2]. Although the need to efficiently use the resources provided by the FPGA is still present, the main challenges are managing the complexity of the design and integration of 3rd party IP cores (e.g. DDR3/4 memory controllers, PCIe blocks, DMAs, 1 and 10 Gigabit Ethernet MAC, processors, DSP blocks...).

Numerous approaches to provide a higher level of abstraction were and are presented, mostly by academia, but also by the commercial vendors. One tool which successfully made a transition from an academic tool [3] to a tool in FPGA engineer's toolbox is Xilinx Vivado HLS.

Several studies of HLS vs RTL can be found in the literature. Vendors of the tools are compelled to present their tools in the nicest way [4] or with usually simplified examples. Some of the studies are also quite general [5] or the examples are simplified versions of the problems from experimental physics [6]. In some cases comparison is valuable but only partially relevant for experimental physics applications [7–9].

The framework described in [10] provides a method to develop applications in Matlab for certain MicroTCA boards. Vivado HLS is also used in 10G and 40G Ethernet accelerators, described in [11], and for two real-time data acquisition applications (crystal identification and timestamp sorting), described in [12]. Recently, Vivado HLS was used for the development of `hls4ml` [13], a machine learning framework for particle physics.

In the rest of the paper, three examples of modules relevant for experimental physics will be presented. Implementations in VHDL and C++ targeting Vivado HLS are presented side-by-side. Interesting snippets of the code are presented to highlight the differences in the two languages, and to give the reader a possibility to compare the readability of the code for himself or herself. Number of lines of code (reported by cloc[14]), resource usage and minimal clock period (= $1/f_{max}$) for both implementations are also reported.

There are several reasons to base the evaluation on Vivado HLS:

- Author has considerable experience both with Xilinx FPGAs in general and with related tools (Vivado, Vivado HLS)
- Xilinx FPGAs are heavily used at DESY and on MicroTCA AMC boards
- The created IP integrates nicely with the rest of the IPs in Xilinx ecosystem
- Vivado HLS is significantly cheaper than other HLS software suites; therefore it is very likely that industrial partners will have access to it

The latest version available at the time of the writing, Vivado HLS 2018.2, was used for the examples in this paper.

Several other High-Level Synthesis software suites exists, such as Intel HLS Compiler, LabVIEW FPGA, Mathworks HDL Coder, Cadence Status, Mentor Graphics Catapult, and Synopsys Synphony C Compiler. Some open-source tools, such as Panda Bamboo and LegUp are also available. These tools were not considered for this paper.

EXAMPLES

To better illustrate the differences, the advantages and the shortcomings of both methods, three examples will be compared. The examples are presented in the order of complexity, simplest first.

The first example, linearization function, requires high throughput, but it is in its core quite simple - each sample is processed on its own; there are no dependencies between the samples. The scheduler has an easy task pipelining the operations.

The second example, two-dimensional mean and standard deviation is slightly more complex; because the samples need to be accumulated together, scheduler needs some help from the designer to be able to pipeline the operations.

The third example, IIR filter is a well studied topic in digital signal processing [15]. In the case presented here,

* jan.marjanovic@desy.de

ISBN: 978-3-95450-201-1 doi:10.18429/JACoW-IBIC2018-THOA01

we require that the filter can process one sample per clock cycle, which would demonstrate whether the HLS compiler is able to schedule the operations to satisfy this requirement.

Example 1: Linearization Module

In the following section, we look at the linearization transfer function, providing harmonic spur suppression in data acquisition system for DFMC-DSx00.

The ADC on DFMC-DSx00 is capable of producing 12-bit samples at maximum rate of 800 MSPS. To simplify the timing closure for the DSP modules in the system, the data from ADC is parallelized into 4 lanes, running at 200 MHz. The output from the ADC interface core is a 128-bit wide AXI4-Stream interface. The interface for DSx00 is shown in Figure 1.

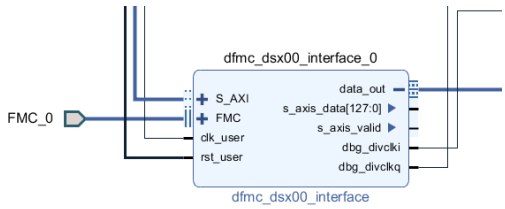

Figure 1: DFMC-DSx00 interface module on KCU105 evaluation board.

The linearization module needs to connect to the ADC interface module on the AXI4-Stream interface; it shall provide an AXI4-Stream slave port. The linearization module shall also provide the output values on an AXI4-Stream master port, which can be attached either to AXI DMA or some other DSP module (e.g. decimation module) in the final application.

The core of the linearization module is a 2nd order polynomial to be evaluated for each of the acquired samples from the ADC.

The transfer function can be described as:

$$y[n] = \sum_{i=0}^{N} a_i x^i[n], N = 2$$

where a_i are statically determined coefficients.

One can immediately notice the simplicity of this approach. Only the current sample is considered at each point, which means that problem is trivially pipelineable.

The top function for this module accepts an array of 8 12-bit numbers as input and it outputs an array of 8 12-bit numbers. By providing the output of the same size as the input, the use of this module is transparent for the downstream DSP modules and the software. The module also provides a bypass for the linearization polynomial, which is useful for evaluating the effects of the linearization process in the real system.

The main body of the core consist of just one simple `for` loop to apply the `linearize()` function on each lane (8 lanes in total). Depending on the value of `bypass` variable, the output is either direct assignment from the input or the return value of `linearize()` function. A compiler directive is used to replicate (or `unroll`) each instance of the loop to match the output data format from the upstream module.

At the core of the `linearize()` function is the implementation of Horner's Rule [16].

Presented in Listing 1 is the core of the linearization function, when implemented at Register-Transfer Level abstraction level in VHDL. The implementation uses `ieee.fixed_pkg` library to handle the fixed point numbers. From this code listing it can be observed that the design intent gets obscured by the pipelining needed to achieve a high operating frequency (f_{max}).

Listing 1: Core of the linearization module, implemented in VHDL

```
proc_stage: process (clk)
begin
  if rising_edge(clk) then
    -- multiply stage
    if stage_valid(2*i) = '1' then
    stage_out(2*i) <= resize(
        arg =>  to_sfixed(in_data_prev(2*i),
                          in_data'left,
                          in_data'right)
                * stage_in(2*i),
        size_res => stage_out(0)
      );
    end if;

    -- addition stage
    if stage_valid(2*i+1) = '1' then
      stage_out(2*i+1) <= resize(
        arg =>  C_COEFS(i) + stage_in(2*i+1),
        size_res => stage_out(0)
      );
    end if;
  end if;
end process;
```

On the other hand, Listing 2 shows that the implementation in C++ is much shorter, and the main arithmetic operations are also clearly visible.

Listing 2: Core of the linearization module, implemented in C++ for Vivado HLS

```
output_t linearize(input_t in) {
  internal_t tmp = 0;
  for (int i = COEFFS_LEN-1; i >= 0; i--) {
    tmp = COEFFS[i] + in * tmp;
  }
  return tmp;
}
```

Coefficients for this experiment are (2.2854652782872233, 0.9962862193648518, -2.506094726425692e-03).

Both implementations were synthesized and implemented for Xilinx KCU105 Evaluation Kit with XCKU040-2FFVA1156E device. Shown in Figure 2 is the result of the automatic pipelining of the `linearize()` function. The compiler (scheduler) has decided to use one clock cycle for each of the multiplication and one clock cycle for the

ISBN: 978-3-95450-201-1 doi:10.18429/JACoW-IBIC2018-THOA01

Table 1: Comparison Between Implementations of 2nd Order Polynomial in HLS vs RTL

resource	HLS (in C++)	RTL (in VHDL)
CLB	221	314
LUT	461	1081
FF	884	938
DSP	16	24
latency	6	6
interval	1	1
clk period	3.903 ns	4.954 ns
lines of code	52	170

addition. This helps satisfy the specified constraint for initialization interval of 1 and results in optimal utilization of DSP48E2 blocks.

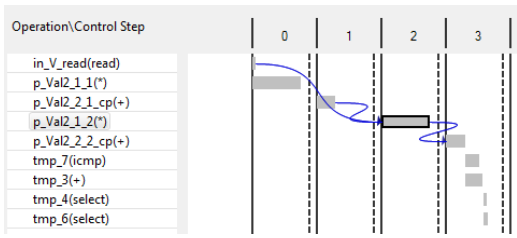

Figure 2: Report from Vivado HLS on implementation.

Shown in the Table 1 are the results of the implementation. Because the HLS compiler takes advantage of the constant propagation, the resource usage is lower compared to the hand-coded VHDL implementation. Vivado HLS was able to remove bits (and corresponding registers and wires) which were determined to be 0 for all inputs.

Example 2: Two-Dimensional Mean and Standard Deviation

The next investigated example will be an implementation of algorithm to determine the center (or mean) and size (or standard deviation) of two-dimensional Gaussian distribution. This module is used together with GigE Vision core on DAMC-TCK7 AMC board. An example of distribution, with annotated mean and standard deviation is shown in Figure 3.

Before we start investigating the implementation, several algorithms for calculation of standard deviation should be considered.

The naive implementation involves two passes, in the first one the mean of the dataset is determined, and in the second one the standard deviation is calculated, by using the mean value calculated in the first step.

The obvious downside of this algorithm is the need to visit each sample twice, which would mean that some memory is

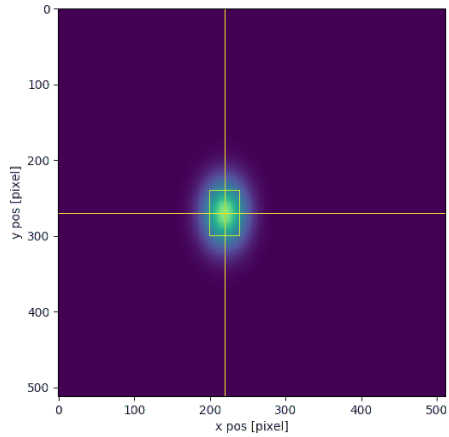

Figure 3: 2D Gaussian distribution; mean and standard deviation are annotated on the image.

required and the mean and the standard deviation cannot be calculated on-the-fly.

The algorithm proposed in [17] only requires one pass through the samples. It also provides better precision when using floating point numbers, but since our implementation will be implemented in integers this property is not relevant in this case.

Based on the equation (15) in [17], for each step a mean needs to be calculated according to the following formula:

$$M_k = M_{k-1} + (x_k - M_{k+1})/k$$

where k is the iteration count.

Because the division in FPGA usually takes several clock cycles and the algorithm is iterative in its nature (previously calculated values are required to calculate new values), this algorithm is not very suitable for implementation in FPGA.

To obtain an algorithm which mostly relies on multiplication and addition, two operations which map nicely to DSP48E1/2 primitives found in Xilinx FPGAs, the naive formula can be rewritten as:

$$\sigma^2 = \frac{1}{N}\sum x_i^2 - \mu^2$$

It can be noted that for each sample only the multiplication and summation is required, and division is only performed twice at the end of the entire frame.

The code for this module can be found in ANNEX A.

From the function signature of the top function `two_dim_stdev()`, it can be seen that the module accepts stream of data and provides the mean and the standard deviation both in X and Y direction. From the compiler directives at the beginning of the function it can be noted that the input is of type AXI-Stream, while the parameters are made available on the AXI4-Lite interface.

The main body of the function is composed of four `for` loops. The outermost two loops (`loop_x` and `loop_y`) traverse the frame in X and Y dimension. The innermost two loops are there to interleave the addition and multiplication

ISBN: 978-3-95450-201-1 doi:10.18429/JACoW-IBIC2018-THOA01

Table 2: Report of the Utilization of Resources From Implementation of Two-Dimensional Standard Deviation and Mean in Vivado HLS

Resource	HLS (in C++)	RTL (in VHDL)
slice	764	-
LUT	1714	-
FF	2507	-
DSP	18	-
BRAM	14	-
SRL	38	-
latency	65663 [1]	-
interval	65663	-
clk period	3.095 ns	-

operations on several accumulators, thus resolving the dependency through an operation as described in [18].

The results of the implementation of the algorithm in Vivado HLS are shown in Table 2. The VHDL module for comparison is unfortunately not provided.

Example 3: IIR Filter

The final example presented in this paper is a 2nd order IIR filter. The implementation in Vivado HLS will be compared to the VHDL implementation from FPGA Firmware Framework for MTCA.4 AMC Modules [19], used in LLRF at DESY. The module `ENT_IIR_TDF1_02` is a highly-optimized IIR filter, implemented in Transposed-Direct-Form-I to take full advantage of DSP48E1 modules. Figure 4 shows how 5 DSP48E1 are used to implement the filter in the VHDL implementation (each DSP48E1 is configured to calculate the result in the form of $B*(D-A)+C$).

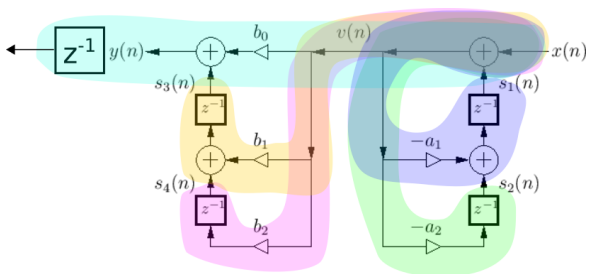

Figure 4: Filter implementation architecture in `ENT_IIR_TDF1_02`.

The C++ code for this module can be found in ANNEX B. The core of the implementation is a class `BiquadFilter` which provides `operator()` to calculate the next sample. The sizes of variables are tuned to use the maximal width of the DSP48E2 module.

[1] = 128 clock cycles (4 pixels on a parallel interface) * 512 lines + 32 clock cycles to initialize variables + 11 clock cycles to flush the multiply-and-accumulate pipeline + 24 clock cycles to gather the accumulators + 48 clock cycles to perform the divisions + 7 clock cycles to calculate the square root + 5 clock cycles used to shift the data from and to pipelines.

Table 3: Comparison Between Implementations of 2nd Order IIR Filter in HLS vs RTL

Resource	HLS (in C++)	RTL (in VHDL)
CLB	32	16
LUT	156	0
FF	147	149
DSP	5	5
BRAM	0	0
SRL	0	0
latency	4	4
interval	1	1
clk period	4.704 ns	4.732 ns
lines of code	60	246

For comparison, both VHDL implement and RTL implementation were compiled for XCKU040-FFVA1152-2 FPGA. The VHDL implementation was packaged with Vivado IP Packager and synthesized (with `-mode out_of_context`) and implemented on the FPGA.

Presented in Table 3 are the results of both implementation. The VHDL implementation is just slighly more optimized in resources, on the other hand the implementation in C++ is much shorter (in terms of lines of code) and the code is easier to read.

This example relies heavily on DSP48E1/2 modules. In the latest versions of Vivado HLS an undocumented library (`dsp_builtins.h`) was added with intrinsics targeting DSP48E1/2 module. Usage of these intrinsics could be another way of implementing this example.

CONCLUSION AND OUTLOOK

Three examples of algorithm implementations in C++ for Vivado HLS were presented. The examples presented here illustrate some quite challenging algorithms with demanding requirements. All examples were able to handle one or more samples per clock cycle, which requires careful organization of operations to allow for pipelining of the operations. It can be seen that Vivado HLS provides an useful alternative even in these challenging examples.

When the requirements are not so stringent, the usage of Vivado HLS provides even more significant improvement in productivity compared to the RTL workflow. This is specially true when the algorithms need to operate on floating-point numbers, since the operations on floating-point numbers usually take several clock cycles and scheduling those can be a non-trivial task.

Vivado HLS seems to be a useful tool to speed up the development. Because of peculiarity of FPGAs, Vivado

ISBN: 978-3-95450-201-1 doi:10.18429/JACoW-IBIC2018-THOA01

HLS cannot turn any C or C++ code into an FPGA module, the code needs to be written in a certain dialect of C or C++ (i.e. no memory allocation, careful selection of mathematic operations and variable types, handling the state to allow for pipelining, ...). Familiarity with the device primitives, such as multiply-and-add block DSP48E1/2, is also an advantage.

Some authors argue that C-like languages are a poor choice for High-Level Synthesis [20] and propose HLS compilers from functional languages, such as Haskell [21]. High-Level Synthesis remains an interesting field of research, where new ideas are still proposed and existing tools are still improved. Because of the decent Quality of Result and improved productivity, it can be expected that HLS workflow will become more and more popular.

ACKNOWLEDGEMENTS

I would like to thank Sven Stubbe (DESY) for the help with the examples and for productive discussions about High-Level Synthesis.

REFERENCES

[1] S. M. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of fpga technology," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, Mar. 2015, ISSN: 0018-9219. DOI: 10.1109/JPROC.2015.2392104.

[2] R. Sass and A. G. Schmidt, *Embedded Systems Design with Platform FPGAs: Principles and Practices*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010, ISBN: 9780080921785, 9780123743336.

[3] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, Apr. 2011, ISSN: 0278-0070. DOI: 10.1109/TCAD.2011.2110592.

[4] F. Sijstermans and J. Li, "Working smarter, not harder: NVIDIA closes design complexity gap with High-Level Synthesis," http://go.mentor.com/4N9cP

[5] R. Nane *et al.*, "A survey and evaluation of FPGA High-Level Synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016, ISSN: 0278-0070. DOI: 10.1109/TCAD.2015.2513673.

[6] Z. Jin, H. Finkel, K. Yoshii, and F. Cappello, "Evaluation of the FIR example using Xilinx Vivado High-Level Synthesis compiler," Jul. 2017. DOI: 10.2172/1375449.

[7] Z. Zhao and J. C. Hoe, "Using Vivado-HLS for structural design: A NoC case study," in *FPGA*, 2017.

[8] J. R. G. Ordaz and D. Koch, "On the HLS design of bit-level operations and custom data types," in *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*, Sep. 2017, pp. 1–8.

[9] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in *2013 International Conference on Field-Programmable Technology (FPT)*, Dec. 2013, pp. 362–365. DOI: 10.1109/FPT.2013.6718388.

[10] P. Prędki, M. Heuer, Ł. Butkowski, K. Przygoda, H. Schlarb, and A. Napieralski, "Rapid-X - an FPGA development toolset using a custom Simulink library for MTCA.4 modules," *IEEE Transactions on Nuclear Science*, vol. 62, no. 3, pp. 940–946, Jun. 2015, ISSN: 0018-9499. DOI: 10.1109/TNS.2015.2413673.

[11] E. Schubert and U. Langenbach, "FPGA-based hardware accelerators for 10/40 GigE TCP/IP and other protocols," in *4th MicroTCA Workshop for Industry and Research*.

[12] T. Marc-André, "Two FPGA Case Studies Comparing High Level Synthesis and Manual HDL for HEP applications," 2018. arXiv: 1806.10672 [physics.ins-det].

[13] J. Duarte *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, P07027, 2018. http://stacks.iop.org/1748-0221/13/i=07/a=P07027

[14] *CLOC - count lines of code*. https://github.com/AlDanial/cloc

[15] J. O. Smith, *Introduction to digital filters with audio applications*, online book, accessed (date accessed). http://ccrma.stanford.edu/~jos/filters/

[16] S. Xu, S. A. Fahmy, and I. V. McLoughlin, "Square-rich fixed point polynomial evaluation on FPGAs," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14, Monterey, California, USA: ACM, 2014, pp. 99–108, ISBN: 978-1-4503-2671-1. DOI: 10.1145/2554688.2554779. http://doi.acm.org/10.1145/2554688.2554779

[17] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, ISBN: 0-201-89684-2.

[18] J. Hrica, "XAPP599 floating-point design with vivado HLS," 2012. https://www.xilinx.com/support/documentation/application_notes/xapp599-floating-point-vivado-hls.pdf

[19] L. Butkowski, T. Kozak, P. Prędki, R. Rybaniec, and B. Yang, "FPGA Firmware Framework for MTCA.4 AMC Modules," in *Proceedings, 15th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2015): Melbourne, Australia, October 17-23, 2015*, 2015, WEPGF074. DOI: 10.18429/JACoW-ICALEPCS2015-WEPGF074.

[20] S. A. Edwards, "The challenges of synthesizing hardware from C-like languages," *IEEE Design Test of Computers*, vol. 23, no. 5, pp. 375–386, May 2006, ISSN: 0740-7475. DOI: 10.1109/MDT.2006.134.

[21] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards, "Hardware synthesis from a recursive functional language," in *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2015, pp. 83–93. DOI: 10.1109/CODESISSS.2015.7331371.

ISBN: 978-3-95450-201-1 doi:10.18429/JACoW-IBIC2018-THOA01

ANNEX A:
STATISTICS MODULE IMPLEMENTED IN C++ FOR VIVADO HLS

```
#include "two_dim_stdev.hpp"

void two_dim_stdev(
        hls::stream<input_t> &in, int &meanx, int &stdx, int &meany, int &stdy
) {

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis register both port=in
#pragma HLS DATA_PACK variable=in field_level
#pragma HLS INTERFACE s_axilite port=meanx bundle=ctrl
#pragma HLS INTERFACE s_axilite port=stdx  bundle=ctrl
#pragma HLS INTERFACE s_axilite port=meany bundle=ctrl
#pragma HLS INTERFACE s_axilite port=stdy  bundle=ctrl


    const int N = 512; // frame size
    const int PAR = 4; // number of parallel HW instances for accum

    ap_uint<30> accum[PAR*INPUT_W];
    ap_uint<38> accum_x[PAR*INPUT_W], accum_y[PAR*INPUT_W];
    ap_uint<45> accum_x2[PAR*INPUT_W], accum_y2[PAR*INPUT_W];

    loop_init: for (int i = 0; i < PAR*INPUT_W; i++) {
        accum[i] = 0; accum_x[i] = 0; accum_y[i] = 0;
        accum_x2[i] = 0; accum_y2[i] = 0;
    }

    loop_y: for (int y = 0; y < N; y++) {
        loop_x: for (int x = 0; x < N/PAR/INPUT_W; x++) {
            input_t z[PAR];
            loop_pipe: for (int i=0; i < PAR; i++) {
#pragma HLS PIPELINE II=1
                in >> z[i];
                loop_arr: for (int j = 0; j < INPUT_W; j++) {
                    accum[i]   += z[i].arr[j];
                    ap_uint<10> pos_x = (PAR*INPUT_W*x+INPUT_W*i+j);
                    ap_uint<10> pos_y = y;
                    accum_x[i]  += pos_x*z[i].arr[j];
                    accum_y[i]  += pos_y*z[i].arr[j];
                    accum_x2[i] += pos_x*pos_x*z[i].arr[j];
                    accum_y2[i] += pos_y*pos_y*z[i].arr[j];
                }
            }
        }
    }

    ap_uint<30> accum_tot = 0;
    ap_uint<38> accum_x_tot = 0, accum_y_tot = 0;
    ap_uint<45> accum_x2_tot = 0, accum_y2_tot = 0;

    loop_gather: for (int  i=0; i < PAR; i++) {
        accum_tot    += accum[i];
        accum_x_tot  += accum_x[i]; accum_y_tot  += accum_y[i];
        accum_x2_tot += accum_x2[i]; daccum_y2_tot += accum_y2[i];
    }

    meanx = accum_x_tot / accum_tot;
    meany = accum_y_tot / accum_tot;
    ap_int<16> varx = accum_x2_tot / accum_tot - meanx*meanx;
    ap_int<16> vary = accum_y2_tot / accum_tot - meany*meany;
    stdx  = hls::sqrt(varx);
    stdy  = hls::sqrt(vary);
}
```

ISBN: 978-3-95450-201-1 doi:10.18429/JACoW-IBIC2018-THOA01

ANNEX B:
IIR FILTER MODULE IMPLEMENTED IN C++ FOR VIVADO HLS

```
#include "iir_hls.hpp"

template<int FIX_W, int FIX_I>
class BiquadFilter {
public:
    BiquadFilter (
            ap_fixed<FIX_W, FIX_I> b0,
            ap_fixed<FIX_W, FIX_I> b1,
            ap_fixed<FIX_W, FIX_I> b2,
            ap_fixed<FIX_W, FIX_I> a1,
            ap_fixed<FIX_W, FIX_I> a2) :
        b0(b0), b1(b1), b2(b2), a1(a1), a2(a2) {
    }

    ap_fixed<FIX_W, FIX_I> operator()(ap_fixed<FIX_W, FIX_I> x){
        ap_fixed<25, 9> tmp_s1   = (x+s1) * -a1 + s2;
        ap_fixed<48, 9> tmp_s2   = (x+s1) * -a2     ;
        ap_fixed<48, 9> tmp_s3   = (x+s1) *  b1 + s4;
        ap_fixed<48, 9> tmp_s4   = (x+s1) *  b2     ;
        ap_fixed<FIX_W, FIX_I> y = (x+s1) *  b0 + s3;

        s1 = tmp_s1;
        s2 = tmp_s2;
        s3 = tmp_s3;
        s4 = tmp_s4;

        return y;
    }

private:
    ap_fixed<FIX_W, FIX_I> b0, b1, b2, a1, a2;
    ap_fixed<25, 9> s1;
    ap_fixed<48, 9> s2, s3, s4;
};

void iir_hls (
        ap_fixed<18, 2> coeffs_b0,
        ap_fixed<18, 2> coeffs_b1,
        ap_fixed<18, 2> coeffs_b2,
        ap_fixed<18, 2> coeffs_a1,
        ap_fixed<18, 2> coeffs_a2,
        ap_fixed<18, 2> data_in,
        ap_fixed<18, 2> &data_out
){

#pragma HLS INTERFACE ap_stable port=coeffs_b0
#pragma HLS INTERFACE ap_stable port=coeffs_b1
#pragma HLS INTERFACE ap_stable port=coeffs_b2
#pragma HLS INTERFACE ap_stable port=coeffs_a2
#pragma HLS INTERFACE ap_stable port=coeffs_a1

#pragma HLS INTERFACE ap_vld port=data_in
#pragma HLS INTERFACE ap_ovld register port=data_out
#pragma HLS INTERFACE ap_ctrl_none port=return

#pragma HLS PIPELINE II=1

    static BiquadFilter<18, 2> f0(coeffs_b0, coeffs_b1, coeffs_b2, coeffs_a1, coeffs_a2);

    data_out = f0(data_in);
}
```