BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 64, No. 4, 2016
DOI: 10.1515/bpasts-2016-0091

FPGA-based bandwidth selection for kernel density estimation using high level synthesis approach

A. GRAMACKI[1]*, M. SAWERWAIN[1], and J. GRAMACKI[2]

[1]Institute of Control and Computation Engineering, University of Zielona Góra, Licealna 9 St., 65-417 Zielona Góra, Poland
[2]Computer Center, University of Zielona Góra, Licealna 9 St., 65-417 Zielona Góra, Poland

Abstract. Field-programmable gate arrays (FPGA) technology can offer significantly higher performance at much lower power consumption than is available from single and multicore CPUs and GPUs (graphics processing unit) in many computational problems. Unfortunately, the pure programming for FPGA using hardware description languages (HDL), like VHDL or Verilog, is a difficult and not-trivial task and is not intuitive for C/C++/Java programmers. To bring the gap between programming effectiveness and difficulty, the high level synthesis (HLS) approach is promoted by main FPGA vendors. Nowadays, time-intensive calculations are mainly performed on GPU/CPU architectures, but can also be successfully performed using HLS approach. In the paper we implement a bandwidth selection algorithm for kernel density estimation (KDE) using HLS and show techniques which were used to optimize the final FPGA implementation. We are also going to show that FPGA speedups, comparing to highly optimized CPU and GPU implementations, are quite substantial. Moreover, power consumption for FPGA devices is usually much less than typical power consumption of the present CPUs and GPUs.

Key words: FPGA, high level synthesis, kernel density estimation, bandwidth selection, plug-in selector.

1. Introduction

The probability density function (PDF) is a key concept in statistics. with many practical applications, see for example [14] and many others. Constructing the most adequate PDF from the observed data is still an important and interesting research problem, especially for large datasets. PDFs are often calculated using nonparametric data-driven methods. One of the most popular nonparametric method is the kernel density estimation (KDE) [21–23, 28]. However, a very serious drawback of using KDE is the large number of calculations required to compute density estimates, as well as to find the optimal bandwidth (computational complexity $O(n^2)$).

In this paper we investigate the possibility of utilizing field-programmable gate arrays (FPGA) to accelerate finding of such the optimal bandwidth. Towards the needs of the paper we have selected one popular and often used algorithm called *plug-in* in literature [13, 28]. This work can be considered as a continuation and extension of the paper [1], where the authors utilize graphics processing units (GPU) for speeding up optimal bandwidth selection. One of the algorithms analysed in that paper was the above mentioned *plug-in*.

Generally, there are two methodologies for speeding up complex numerical algorithms: software-based and hardware-based. In this paper we concentrate only on hardware-based methods. The commonly known approaches are as follows: (a) computing on general purpose single and multicore CPU microprocessors, (b) computing on distributed environments (e.g. clusers, grids, etc.), (c) computing on GPUs [25, 24, 20] (d) computing on digital signal processors (DSP) units and (e) computing on FPGA chips [11, 15, 17, 18, 27, 29].

In the paper we are concerned with FPGA approach. In [10] the author considers a problem how to use FPGA for fast computing of PDFs using direct very high speed integrated circuits hardware description language (VHDL) programming approach. However, the problem we are concerning is of different nature, as we concentrate our attention for computing the optimal bandwidth for PDF (see Section 2).

To develop the final FPGA design we use the high level synthesis (HLS) approach [8, 16], in which no direct hardware description language (HDL) coding is needed (typically, VHDL or Verilog languages[a] are used).

The remainder of the paper is organized as follows. In Section 2, we turn our attention to give the reader some preliminary information on KDE and bandwidth selection. In Section 3 we provide detailed mathematical formulas for calculating optimal bandwidth using the PLUGIN method. In Section 4 we cover all the necessary details on our FPGA-based implementation. We also present practical experiments carried out and discuss the results. In Section 5, we conclude the paper.

2. Kernel density estimation and bandwidth selection

The univariate kernel density estimator $\hat{f}$ for a random sample X_i $(i = 1, 2, \dots, n)$, drawn from a common and usually unknown density function f is given by

*e-mail: a.gramacki@issi.uz.zgora.pl

[a] It is worth to note that OpenCL framework, which is commonly used by GPU programmers, also becomes available for FPGA devices. Nowadays, OpenCL is offered by Altera SDK for OpenCL to easily implement OpenCL applications for FPGA. Recently, Xilinx announced a similar solution, namely SDAccel Development Environment for OpenCL, C, and C++.

$$\hat{f}(x) = \frac{1}{n}\sum_{i=1}^{n} K_h\left(x - X_i\right), \tag{1}$$

where

$$K_h(u) = h^{-1}K\left(h^{-1}u\right). \tag{2}$$

h is a positive real number called smoothing parameter or bandwidth. K is the kernel function – a symmetric function that integrates to one. The scaled (K_h) and unscaled (K) kernels are related in Eq. (2). In most cases the kernel K has the form of a standard Gaussian normal density, that is

$$K(u) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}u^2\right). \tag{3}$$

If we have the bandwidth h, we can determine the estimator $\hat{f}$ of the unknown density function f using (1). The bandwidth h is the parameter which exhibits a strong influence on the resulting KDE.

As an example of how KDE works consider a toy dataset of eight data points: $X_i = \{0, 1, 1.1, 1.5, 1.9, 2.8, 2.9, 3.5\}$. Three different KDEs based on these data are depicted in Fig. 1. It is easy to notice how the bandwidth h influences the shape of the KDE curve. Lines in bold show the estimated PDFs, while normal lines show the shapes of individual kernel functions K (Gaussians). Dots represent the data points X_i.

Eq. (1) can be obviously extended to the multivariate case. In the most general variant, the scalar bandwidth h is replaced by the unconstrained bandwidth matrix H (which is symmetric and positive definite). However, the multivariate case is not considered in the paper. The monographs [21, 28] provide an overview of the research in the area of multivariate KDE. Choosing the best value of H is not a trivial task and this problem was and still is extensively studied in literature [3–5].

Currently available selectors can be roughly divided into three classes [12, 28]. The first class uses very simple and easy to calculate mathematical formulas. They were developed to cover a wide range of situations, but do not guarantee being enough close to the optimal (under certain criteria) bandwidth. They are often called rules-of-thumb methods. The second class contains methods based on cross-validation ideas with more precise mathematical arguments, but they require much more computational effort. However, in reward for it, we get bandwidths more accurate for a wider range of density functions. The third class contains methods based on plugging in estimates of some unknown quantities that appear in formulas for the asymptotically optimal bandwidth. They are often called plug-in.

One selected method from the third class (for the univariate case) is investigated in the paper. The method is briefly presented in Section 3 and from now on it will abbreviated as the PLUGIN.

3. The PLUGIN method and data preprocessing

In Algorithm 1 we provide a recipe for calculation of the optimal bandwidth using the PLUGIN method (the symbols used are exactly such as in the book [28]). All the necessary details on the method, as well as details on deriving of particular mathematical formulas can be found in many source materials, see for example books [13, 28].

It is important to stress that the PLUGIN algorithm is a strictly sequential computational process (see Fig. 2; parallel processing is possible only internally in Steps IV and VI) as every step depends on the results obtained in the previous steps. First we calculate the variance and the standard deviation estimators of the input data, see Step I in Algorithm 1. Then we calculate some more complex formulas from Step II to Step VI. Finally, we can substitute them into equation given in Step VII to get the searched optimal bandwidth value h.

Our implementation of the Algorithm 1 is carried out in fixed-point arithmetic (see section 4.2). Unfortunately, using the raw data while conducting the required calculations, threatens a potential problems with overflow, especially while calculating the value of $\hat{\Psi}_8^{NS}$, see Step II in Algorithm 1. Note that the estimate of standard deviation in $\hat{\Psi}_8^{NS}$ is raised to the power of 9. For large values of σ it results in extremely small values of $\hat{\Psi}_8^{NS}$. The above problems can be successfully overcome if the input datasets are standardized using the z-score formula, that is

$$Z_i = \frac{X_i - \mu}{\sigma} \tag{4}$$

where μ and σ are mean and standard deviation of the original vector X respectively. Z-score guarantees that $\hat{\sigma} = 1$ in $\hat{\Psi}_8^{NS}$ and, consequently, $\hat{\Psi}_8^{NS}$ entity has simply a constant value.

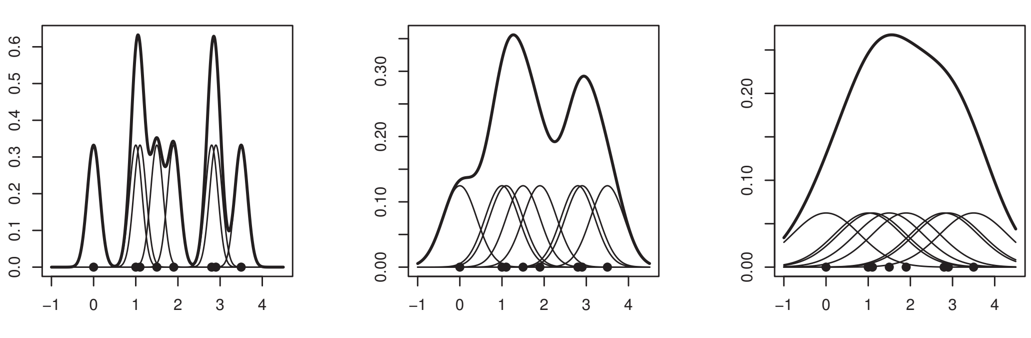

Fig. 1. An example of using kernel density estimators for determining the probability density function

Algorithm 1: Main computational steps of the PLUGIN algorithm

```
Data: data set $X$, contains $n$ elements
Result: value $h$ represents the optimal bandwidth for kernel density
    estimation
Step I: Calculate the estimates of variance ($\hat{V}$) and standard deviation
    ($\hat{\sigma}$):
        $\hat{V} \leftarrow \frac{1}{n-1}\sum_{i=1}^{n} X_i^2 - \frac{1}{n(n-1)}\left(\sum_{i=1}^{n} X_i\right)^2, \ \hat{\sigma} \leftarrow \sqrt{\hat{V}}.$
Step II: Calculate the estimate $\hat{\Psi}_8^{NS}$ of functional $\Psi_8$:
        $\hat{\Psi}_8^{NS} \leftarrow \frac{105}{32\sqrt{\pi}\hat{\sigma}^9}.$
Step III: Calculate the bandwidth of the kernel estimator of function
    $f^{(4)}$ (4th derivative of function $f$, that is $f^{(r)} = \frac{d^r f}{dx^r}$):
        $g_1 \leftarrow \left(\frac{-2K^6(0)}{\mu_2(K)\hat{\Psi}_8^{NS} n}\right)^{1/9}, \ K^6(0) = -\frac{15}{\sqrt{2\pi}}, \ \mu_2(K) = 1$
Step IV: Calculate the estimate $\hat{\Psi}_6(g_1)$ of functional $\Psi_6$:
        $\hat{\Psi}_6(g_1) \leftarrow \frac{1}{n^2 g_1^7}\left[\sum_{i=1}^{n}\sum_{j=1}^{n} K^{(6)}\left(\frac{X_i - X_j}{g_1}\right)\right],$
        $K^6(x) = \frac{1}{\sqrt{2\pi}}\left(x^6 - 15x^4 + 45x^2 - 15\right)e^{-\frac{1}{2}x^2}.$
Step V: Calculate the bandwidth of the kernel estimator of function $f^{(2)}$:
        $g_2 \leftarrow \left(\frac{-2K^4(0)}{\mu_2(K)\hat{\Psi}_6(g_1) n}\right)^{1/7}, \ K^4(0) = \frac{3}{\sqrt{2\pi}}, \ \mu_2(K) = 1$
Step VI: Calculate the estimate $\hat{\Psi}_4(g_2)$ of functional $\Psi_4$:
        $\hat{\Psi}_4(g_2) \leftarrow \frac{1}{n^2 g_2^5}\left[\sum_{i=1}^{n}\sum_{j=1}^{n} K^{(4)}\left(\frac{X_i - X_j}{g_2}\right)\right],$
        $K^4(x) = \frac{1}{\sqrt{2\pi}}\left(x^4 - 6x^2 + 3\right)e^{-\frac{1}{2}x^2}.$
Step VII: Calculate the final value of the bandwidth $h$:
        $h \leftarrow \left(\frac{R(K)}{\mu_2(K)^2\hat{\Psi}_4(g_2) n}\right)^{1/5}, \ R(K) = \frac{1}{2\sqrt{\pi}}, \ \mu_2(K) = 1$
```

Applying the data standardization requires an extra operation on the h value in Step VII in Algorithm 1, that is

$$h_{\text{final}} = h \cdot \hat{\sigma}, \tag{5}$$

where h is the bandwidth calculated for the standardized dataset and $\hat{\sigma}$ is the standard deviation of the original vector X. The correctness of the above equation can be easily proofed algebraically.

To reduce the calculation burden we can also slightly change equations $\hat{\Psi}_6(g_1)$ and $\hat{\Psi}_4(g_2)$ in Algorithm 1. It is easy to notice a symmetry, that is

$$K^{(6)}\left(\frac{X_i - X_j}{g_1}\right) = K^{(6)}\left(\frac{X_j - X_i}{g_1}\right). \tag{6}$$

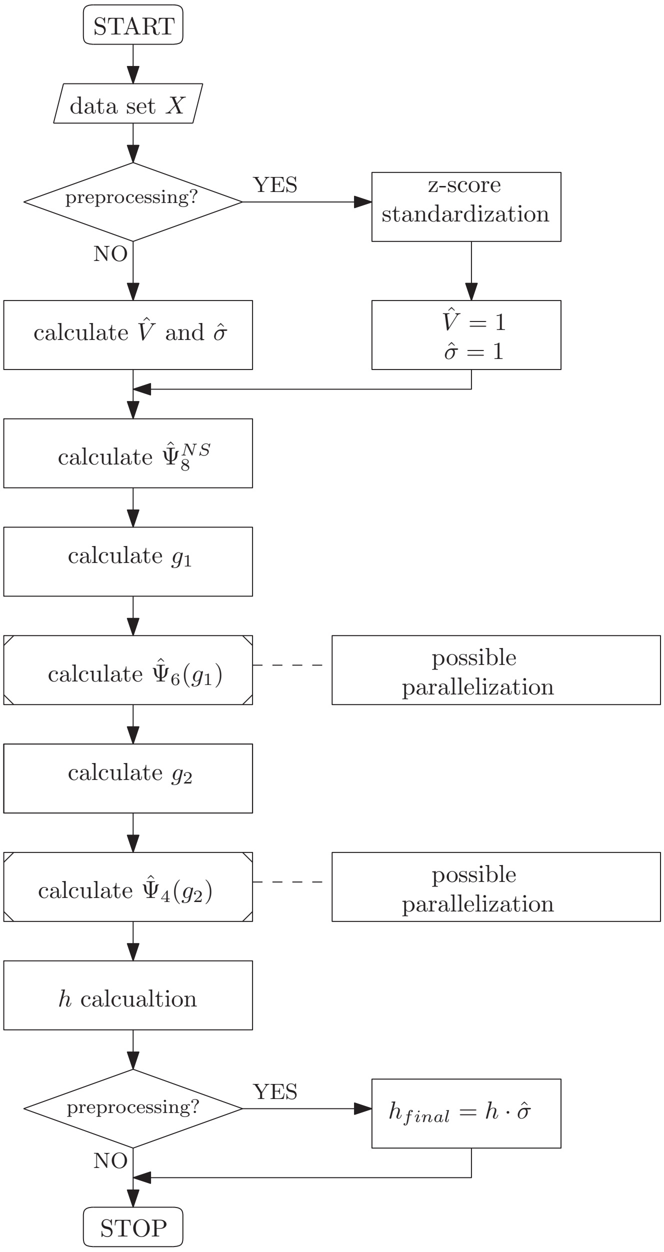

Fig. 2. Flowchart of the PLUGIN algorithm with optional data preprocessing (z-score standardization)

So, the double summations can be changed and, consequently, the final formula for $\hat{\Psi}_6(g_1)$ has now the following form

$$\hat{\Psi}_6(g_1) \leftarrow \frac{1}{n^2 g_1^7}\left[2\left(\sum_{i=1}^{n}\sum_{j=1,i<j}^{n} K^{(6)}\left(\frac{X_i - X_j}{g_1}\right)\right) + \right. \\ \left. + nK^{(6)}(0)\right] \tag{7}$$

(note that for different summation ranges, the 2 before sums an extra factor added, that is $nK^{(6)}(0)$). Obviously, the same concerns $K^{(4)}$ and $\hat{\Psi}_4(g_2)$

$$\hat{\Psi}_4(g_2) \leftarrow \frac{1}{n^2 g_2^5}\left[2\left(\sum_{i=1}^{n}\sum_{j=1,i<j}^{n} K^{(4)}\left(\frac{X_i - X_j}{g_2}\right)\right) + \right.$$
$$\left. + nK^{(4)}(0)\right]. \tag{8}$$

Computation complexity of Steps IV and VI (double summations), where the symmetry property is used, still belongs to $O(n^2)$ complexity class

$$T(n) = \sum_{i=1}^{n}\sum_{j=i}^{n} T_k = \frac{1}{2}(n^2+n)T_k \tag{9}$$

where $T_k = T_1 + T_2 + T_3$, and T_1 represents computation time for the differences, T_2 represents division time, and T_3 represents time for computing $K^{(6)}$ and $K^{(4)}$ polynomials.

4. FPGA-based implementation

4.1. Xilinx's high level synthesis. High level synthesis (HLS) is an automated design process that interprets an algorithmic description of a problem (given in high level languages C/C++) and translates this problem into a so called register-transfer level (RTL) HDL code. Then in turn this HDL code can be easily synthesized to the gate level by the use of a logic synthesis tool, like for example Xilinx ISE Design Suite, Xilinx Vivado Design Suite, Altera Quartus II.

In this paper we discuss results obtained using a tool called Xilinx Vivado High Level Synthesis, a feature of Vivado Design Suite. This tool supports C/C++ inputs, and generates VHDL/Verilog/SystemC outputs. Other solutions are offered by *Scala* programming language [2] and a specialised high level synthesis language called *Cx* [26]. It should also be mentioned that a similar tool called A++ is also available for Altera FPGA devices.

4.2. Implementation preliminaries. Before implementing the PLUGIN Algorithm 1 it is important to take some assumptions affecting both performance and resource consumption.

The first assumption is about a proper arithmetic used. The floating-point one gives very good range and precision. Unfortunately, from FPGA's point of view, this representation is very resource demanding. In contrast, the fixed-point arithmetic is much less resource demanding but its range and precision are more limited.

Hence, the exact fixed point representation was determined based on a careful analysis of the particular intermediate values taken during calculations. If the input dataset does not contain extremely large outliers (which suggests that such dataset should be first carefully analysed before any statistical analysis taken) and if the z-score standardization is used, $Q32.32$ fixed point representation is sufficient for all calculations (that is: integer part length $m = 31$, fractional part length $n = 32$, word length $N = 64$ and the first bit represents the sign). Also, note that as a result of the z-score standardization, the vales of $\hat{V}$, $\hat{\sigma}$, $\hat{\Psi}_8^{NS}$ are constant and this significantly simplifies the calculations. The fractional part does give the required precision. However, the integer part must also be sufficiently large, as n^2 factors are present in the PLUGIN algorithm.

The second assumption is about choosing the most adequate methods for calculating individual steps in Algorithm 1. Now it needs to be stressed that programming for FPGA devices differs considerably from programming for CPUs/GPUs devices. FPGA devices are built from a large number of simple logical blocks like: look-up tables (LUT), flip-flops (FF), block RAM memory (BRAM), specialized DSP units (DSP). These blocks can be connected each other and can implement only relatively low-level logical functions (the so called gates level). As a consequence, even very basic operations, like for examples the adder for adding two numbers must be implemented from scratch. In description of the PLUGIN Algorithm \ref{alg:plugin} one can easily indicate such operators like (a) addition, (b) subtraction, (c) multiplication, (d) division, (e) reciprocal, (f) exponent, (g) logarithm[b], (h) power, (i) square roots, (j) higher order roots.

Our implementation utilizes the following methods: CORDIC [6, 7] for calculating exponents and logarithms, divisions were replaced by multiplications and reciprocals, difference operators were replaced by addition of negative operands. Additionally, one extra implementation of the exponent function was used for calculations of $K^{(6)}$ and $K^{(4)}$ in Algorithm 1. This implementation is based on the Remez algorithm [9, 19] and is open to pipelining. As a consequence, a significant speedup can be achieved during calculations of Steps IV and VI in Algorithm 1.

It is also worth to note that the authors' implementation of the division operator (base on multiplications and reciprocals; the reciprocal is based on the Newton method) is significantly faster than the default division operator available in Vivado HLS. Moreover, the another advantage of using our own operators, is that intellectual property core IPCore (Xilinx's library of many specialized functions available for FPGA projects) is not needed. As a consequence, the generated VHDL codes are more portable for FPGA chips from different than Xilinx vendors.

The third assumption during implementing of the PLUGIN algorithm was to enable the nominal clock frequency of an FPGA chip used (see chapter 4.4 for details). During experiments it was turned out that the usage of the original division operator resulted in problems with reaching the required frequency. The authors' original implementation of the division operator (base on multiplications and reciprocals) solved this problem.

The forth assumption was that all the input datasets must be stored in the BRAM memory, which are available in almost all current FPGA chips. They have enough capacity to store truly large data, like even 500,000 elements or more.

[b] Logarithm is not directly present in the PLUGIN mathematical formulas, but it is used while implementing higher order roots from the following definition $x^y = \exp(y \ln x)$.

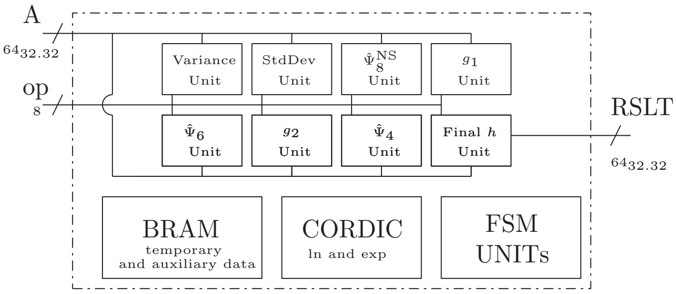

Fig. 3. General overview of the main units for the FPGA-based PLUGIN algorithm implementation

Table 1
Resources usage for three different FPGA implementations of the PLUGIN algorithms as well as CPU and GPU implementations. Additionally power consumption is included. For FPGA implementations, this is an estimate value called total on-chip power; the power consumed internally within the FPGA, equal to the sum of device static power and design power. It is also known as thermal power

Method	BRAM 18k	DSP	FF	LUT	Watts
literal	128	1164	80753	81995	3.938
minimal	128	240	15889	22895	1.153
fast	128	1880	85775	38050	6.963
CPU	–	–	–	–	≈ 88
GPU	–	–	–	–	≈ 250

4.3. Implementation details. In Fig. 3 we show the scheme of the PLUGIN implementation where all the main components are presented. They correspond literally to the seven steps shown in Algorithm 1.

Figure 4 presents general architecture of the functional unit for computing $\hat{\Psi}_4(g_2)$ (Step VI in Algorithm 1). It is worth to note that the proper architecture of this unit must be reached during careful coding in Vivado HLS, using techniques like listed in section 4.2.

We developed three different versions of the PLUGIN algorithm. The complete source codes are available for download in [30].

The first implementation, called literal, is just a literal rewriting of Algorithm 1 (with the improvements (7) and (8)). No additional actions were taken toward optimization of both execution time and resource requirements. This version can operate with any unscaled input data (assuming that all the inputs as well as all the internal results fulfil the fixed-point ranges that have been set). This version automatically (Vivado decides) utilizes pipelining. However, the pipelining doesn't make the implementation enough fast and additionally, large number of DSP blocks is used. FFs and LUTs usage is also quite big (see Table 1).

The second implementation, called minimal, is written so that it is optimized for resource utilization, mainly the DSP units. To reduce the number of the DSP units some dedicated functions for addition and multiplication are required. Using

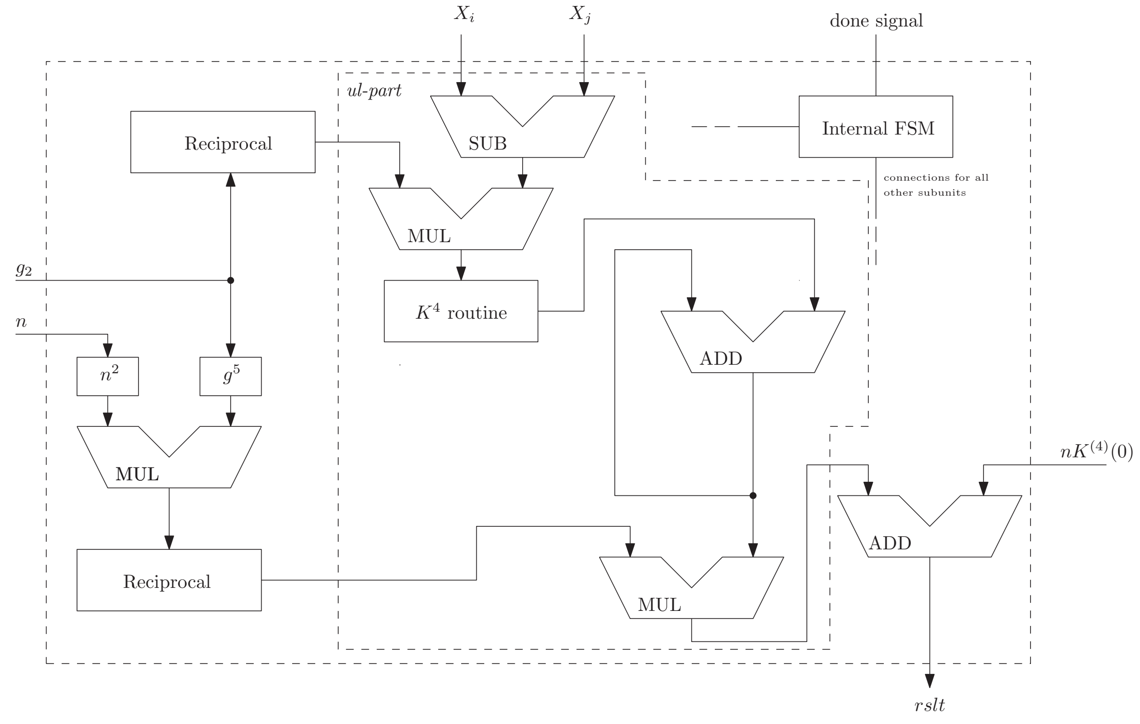

Fig. 4. General architecture of the $\hat{\Psi}_4(g_2)$ unit at the block-level view. The extra frame called ul-part shows the part of the Step VI in Algorithm 1 where loop unrolling can be used

Vivado HLS compiler's pragmas (#pragma HLS INLINE off) pipelining can be disabled (on default, during translation of the high level codes into HDL ones pipelining is enabled whenever it is possible). As can be observed in Table 1, a significant reduction of the DSP units was achieved. It confirms the fact that Vivado HLS is very sensitive for the structure of the high level codes being translated into HDL ones. So that, to achieve good performance and resource usage many modifications of the high level codes are required.

The third implementation, called fast, is written so that it is optimized for time execution. Addition and multiplication functions were implemented in two ways. In the first way (similar as in minimal implementation) the pipelining is disabled, while in the second way it is enabled. The pipelined versions of the functions are used in Steps IV and VI in Algorithm 1 as these two steps are crucial for the final performance. Additionally, in these two steps a dedicated implementation of the exponent function was used (based on Remez algorithm which is more likely to pipelining). Also, a technique known as loop unrolling was used in a manual manner (see sample codes in Fig. 6). Although Vivado HLS uses automatic loop unrolling, this feature doesn't work correctly in our algorithm (as it can operate with datasets of any size and the exact number of loops is not known in advance).

Table 2
Execution times (in sec.) for three different FPGA implementations of the PLUGIN algorithm and for CPU and GPU implementations. The literal implementation is just a literal rewriting of Algorithm 1 (with the improvements (7) and (8)). The minimal implementation is written so that it is optimized for resource utilization, mainly the DSP units. The fast implementation is written so that it is optimized for time execution. More details on particular implementations can be found in the text

n	literal	minimal	fast	CPU	GPU
128	0.0555	0.0324	0.000276	0.0210	0.00699
256	0.2266	0.1363	0.000560	0.0252	0.00788
384	0.5155	0.3152	0.000889	0.0322	0.00947
512	0.9112	0.5513	0.001257	0.0346	0.00962
640	1.4466	0.8968	0.001667	0.0361	0.01063
768	2.1023	1.3205	0.002114	0.0375	0.01172
896	2.8771	1.8232	0.002606	0.0405	0.01447
1024	3.7666	2.3926	0.003140	0.0427	0.01641

```
// literal implementation
psi4_f1: for( i=0; i<N; i++ ) {
  psi4_f2: for( j=i+1; j<N; j++ ) {
    s = s + k4( ( ( x[i] - x[j] ) / g2) );
  }
}
// minimal implementation
rg2 = reciprocal( g2 );
psi4_f1: for( i=0; i<N; i++ ) {
  psi4_f2: for( j=i+1; j<N; j++ ) {
    s = fADD( s, k4( fMUL( fADD( x[i], -x[j] ), rg2 ) ) );
  }
}
// fast implementation
rg2 = reciprocal( g2 );
psi4_f1: for( i=0; i<N; i++ ) {
  psi4_f2: for( j=i+1; j<N; j+=2 ) {
    #pragma HLS EXPRESSION_BALANCE
    #pragma HLS PIPELINE
    if( j == i+1 ) tmp = 0.0;
    if( j<N ) { tmp1 = 0.0; tmp2 = 0.0; }
    psi4_f1_b0: {
      tmp1a = pfADD( x[i], -x[j] );
      tmpva = pfMUL( tmp1a, rg2 ); tmp1 = k4( tmpva );
    }
    psi4_f1_b1: {
      if( (j+1) < N ) {
        tmp1b = pfADD( x[i], -x[j+1] );
        tmpvb = pfMUL( tmp1b, rg2 ); tmp2 = k4( tmpvb );
      }
    }
    if( j<N ) {
      tmp = pfADD( tmp, tmp1 ); tmp = pfADD( tmp, tmp2 );
    }
    if( j+2>=N ) s = pfADD (s, tmp );
  }
}
```

Fig. 6. Three fundamental methods of the for loop implementation used in $\hat{\Psi}_4(g_2)$ calculation (step VI in Algorithm 1, step IV is implemnented in the same way). In the fast implementation the loop unrolling is used twice. fADD, fMUL functions do not utilize pipelining, while pfADD i pfMUL functions do it

The fourth and fifth implementations used during experiments (called *CPU* and *GPU* respectively) are the ones implemented and investigated in [1]. CPU implementation utilizes the SSE (Streaming SIMD Extensions) of the current multicore CPUs.

4.4. Results. During all practical experiments the target *Xilinx Virtex-7 xc7vx690tffg1761–2* device was used. Its nominal working frequency is 200MHz (or 5 ns for a single clock tact). CPU implementation was run on *Intel Processor i7 4790k 4.0 GHz*. *Geforce 480GTX* graphics card was used for GPU implementation. *Vivado HLS ver. 2015.2* was used for developing all the FPGA implementations.

The summary of the resource consumption is given in Table 1. Additionally, power consumption is included. It is a real power (in Watts) taken by the FPGA chip after physical implementation of the PLUGIN algorithm using Vivado Design Suite. This is an estimate value and is called total on-chip power; the power consumed internally within the FPGA, equal to the sum of device static power and design power. It is also known as thermal power. The power consumption of the FPGA implementations is significantly smaller comparing with the power consumption of the CPU and GPU implementations. The power consumption for the CPU and GPU used in our experiments are an average (catalogue-like) values.

The summary of the execution times for three different implementations of the PLUGIN algorithm, as well as CPU and GPU ones is given in Table 2. The minimal and the fast

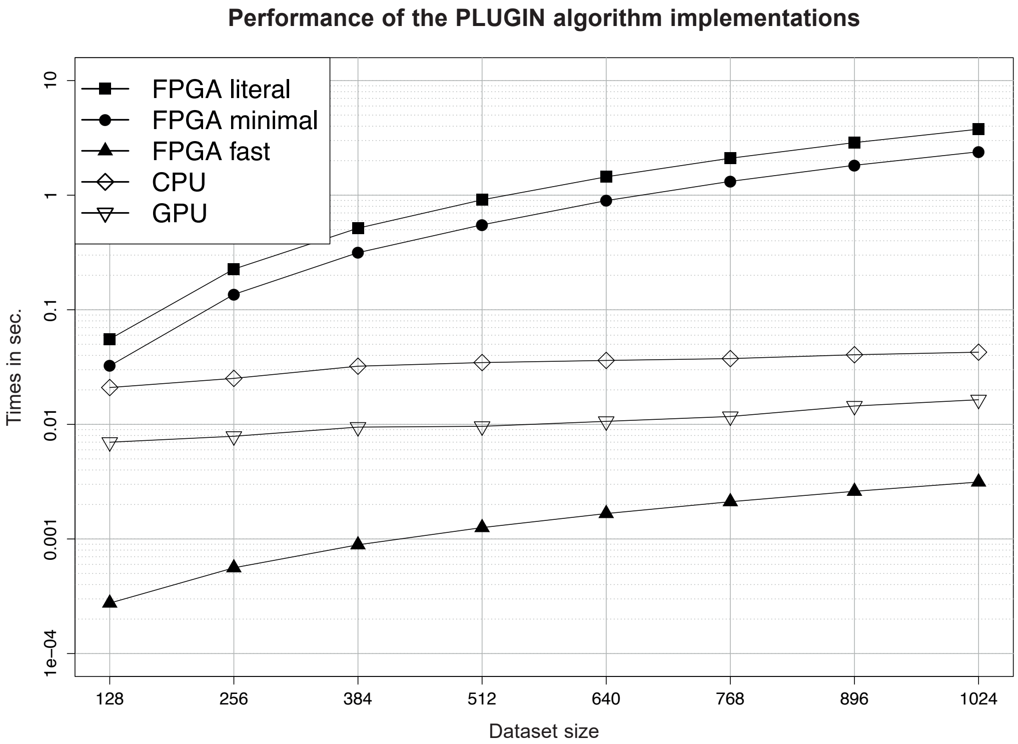

Fig. 5. Performance and scalability of different PLUGIN algorithm implementations (for better readability log scale for *Y* axis is used)

implementations were run on 200MHz nominal clock while the literal implementation was run with 166 MHz nominal clock. This frequency degradation was caused mainly because of some limitations of the original division operator implemented in Vivado HLS.

Of course the best performance was achieved for the fast implementation (even compared to the *CPU* and to the *GPU* implementations). This is the result of combination of the following three optimization techniques used: (a) implementation of some dedicated arithmetic operators, (b) a proper exponential function approximation and (c) the loops unrolling.

Table 3
Speedups for three different FPGA implementations of the PLUGIN algorithm and for CPU and GPU implementations. Also the mean values are calculated

n	literal/fast	minimal/fast	CPU/fast	GPU/fast
128	201	118	76	25
256	404	243	45	14
384	580	354	36	11
512	725	439	28	8
640	868	538	22	6
768	994	625	18	6
896	1104	700	16	6
1024	1200	762	14	5
mean value	**literal/fast**	**minimal/fast**	**CPU/fast**	**GPU/fast**
–	759.5	472.4	31.9	10.125

A very significant speedup was achieved comparing the fast and the literal implementation (average speedup about 760, see Table 3). The fast implementation is faster then the CPU implementation (average speedup about 32, see Table 3). The fast implementation is also faster then the GPU implementation (average speedup about 10, see Table 3).

The summary of the accuracy for three different implementations of the PLUGIN algorithm is given in Table 4. h_{ref} is the reference bandwidth calculated in double floating point arithmetic (in C++ program, 15–17 significant decimal digits). It is worth to note that the relative errors for literal, minimal and fast implementations are very small (not more than 0.004%). In practical applications such small values can be in fact neglected.

The summary of the scalability of different PLUGIN algorithm implementations is presented in Fig. 5. Scalability of the FPGA implementations is nearly linear, which is a very welcome behaviour. The corresponding results for *CPU* and *GPU* implementations can be found in [1]. The figure is in fact a graphical summary of data given in Table 2.

Simplified source codes of the three FPGA implementations are presented in Fig. 6. Complete source codes (C++ and resulted Vivado HLS translations into VHDL) are available in [30]. The first version is just the literal implementation of the step VI in Algorithm 1 in C language. Unfortunately, as can be observed in Table 2 and in Fig. 5 such implementation is very slow. In the second version multiplications and additions are

Table 4
Accuracy (relative error) for three different FPGA implementations of the PLUGIN algorithms. h_{ref} was calculated in C++ direct implementation of Algorithm 1 in floating point double arithmetic (15–17 significant decimal digits). $|\delta_x| = \frac{|h_{method} - h_{ref}|}{|h_{ref}|} * 100\%$ where h_{method} is $h_{literal}$, $h_{minimal}$ or h_{fast}

n	$h_{literal}$	h_{re}	$\|\delta_x\|$ (%)
128	0.304902711650357	0.304902701728222	3.25e-06
256	0.227651247521862	0.227651285449348	1.67e-05
384	0.202433198224753	0.202433187549741	5.27e-06
512	0.242707096505910	0.242707026022425	2.9e-05
640	0.190442902734503	0.190443702342891	0.00042
768	0.175199386896566	0.175199406819444	1.14e-05
896	0.172251554206014	0.172251524317464	1.74e-05
1024	0.174044180661440	0.174044236921001	3.23e-05
n	$h_{minimal}$	h_{re}	$\|\delta_x\|$ (%)
128	0.304902980336919	0.304902701728222	9.14e-05
256	0.227651586290449	0.227651285449348	0.000132
384	0.202433346537873	0.202433187549741	7.85e-05
512	0.242707266006619	0.242707026022425	9.89e-05
640	0.190443017752841	0.190443702342891	0.000359
768	0.175199396442622	0.175199406819444	5.92e-06
896	0.172251742798835	0.172251524317464	0.000127
1024	0.174044403014705	0.174044236921001	9.54e-05
n	h_{fast}	h_{ref}	$\|\delta_x\|$ (%)
128	0.304901758907363	0.304902701728222	0.000309
256	0.227651913650334	0.227651285449348	0.000276
384	0.202433891594410	0.202433187549741	0.000348
512	0.242707268567756	0.242707026022425	9.99e-05
640	0.190443484811112	0.190443702342891	0.000114
768	0.175199736841023	0.175199406819444	0.000188
896	0.172251721611246	0.172251524317464	0.000115
1024	0.174044031649828	0.174044236921001	0.000118

realized using dedicated functions (*fADD*, *MUL*). Also a dedicated function for reciprocal operator was implemented. In the third version much more modification was implemented. First, loop unrolling was used, second, Vivado HLS pragmas were used and third, multiplications and additions were realized using dedicated functions with pipelining enabled (*pfADD*, *pfMUL*).

5. Conclusions

HLS tools are competitive with manual design techniques using HDLs. Implementation time of complex numerical algorithms can be essentially reduced (comparing to direct coding in HDL languages).

Unfortunately, to obtain efficient FPGA implementations, many changes to source codes are required, comparing to equivalent implementations for CPUs and/or GPUs. This is because FPGA devices use specific primitives (DSP, BRAM, FF, LUT blocks) and programmers should control their utilization manually. However, this control is performed on the level of C/C++ codes, not the HDL ones. It is also worth to stress that using the HLS approach allows to obtain implementations which are often faster than CPU and/or GPU counterparts.

Another crucial motivation for replacing GPU or CPU solutions by their FPGA equivalents is power consumption. FPGA can settle for single Watts, while CPU or GPU counterparts typically take tens/hundreds of Watts or even more.

Another possible step toward fast implementations of numerical algorithms could be considering of a direct HDL implementation of the PLUGIN algorithm. This will definitely be much more difficult and will require much more time to complete this work. From the other hand, this could be an excellent occasion to evaluate the quality and effectiveness of the codes generated by Vivado.

Last but not least, one could consider using modern DSP chips which offer many interesting possibilities and are potentially interesting for implementing pure numerical algorithms.

Acknowledgements. We would like to thank for useful discussions with colleagues at the Institute of Control and Computation Engineering (ISSI) of the University of Zielona Góra, Poland. We would like also to thank to anonymous referees for useful comments on the preliminary version of this paper. The numerical results were done using the hardware and software available at the "FPGA/GPU μ-Lab" located at the ISSI.

References

[1] W. Andrzejewski, A. Gramacki and J. Gramacki, "Graphics processing units in acceleration of bandwidth selection for kernel density estimation", *Int. J. Appl. Math. Comput. Sci.* 23(4), 869–885 (2013).

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A.Waterman, R. Avizienis, J. Wawrzynek and K. Asanovi, "Chisel: constructing hardware in a scala embedded language", *Design Automation Conference IEEE,* 1212–1221 (2012).

[3] J. E. Chacón and T. Duong, "Multivariate plug-in bandwidth selection with unconstrained pilot bandwidth matrices", *TEST (Springer)* 19(2), 375–398 (2010).

[4] J. E. Chacón and T. Duong, "Unconstrained pilot selectors for smoothed cross validation", *Australian & New Zealand Journal of Statistics* 53, 331–351 (2011).

[5] J. E. Chacón and T. Duong, "Efficient recursive algorithms for functionals based on higher order derivatives of the multivariate Gaussian density", *Statistics and Computing* 25, 959–974 (2015).

[6] J. E. Volder, "The CORDIC trigonometric computing technique", *IRE Transactions on Electronic Computers* EC-8, 330– 334, (1959).

[7] J. S. Walther, "A unified algorithm for elementary functions", *Proc. of Spring Joint Computer Conference*, 379–385 (1971).

[8] P. Coussy and A. Morawiec, *High-Level Synthesis From Algorithm to Digital Circuit*, Springer, Heidelberg (2008).

[9] N. Daili and A. Guesmia, “Remez algorithm applied to the best uniform polynomial approximations”, *Gen. Math. Notes* 17(1), 16–31 (2013).

[10] S. A. Fahmy and A. R. Mohan, “Architecture for real-time nonparametric probability density function estimation”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21(5), 910–920 (2013).

[11] I. Grobelna, R. Wiśniewski, M. Grobelny and M. Wiśniewska, “Design and verification of real-life processes with application of Petri nets”, *IEEE Transactions on Systems, Man, and Cybernetics: Systems* PP(99), 1–14, DOI: http://dx.doi.org/10.1109/TSMC.2016.2531673 (2016).

[12] M. C. Jones, J. S. Marron and S. J. Sheather, “A brief survey of bandwidth selection for density estimation”, *Journal of the American Statistical Association* 91(433), 401–407 (1996).

[13] P. Kulczycki, *Kernel Estimators in Systems Analysis*, Wydawnictwo Naukowo-Techniczne, Warsaw, 2005 [in Polish].

[14] P. Kulczycki and M. Charytanowicz, “A complete gradient clustering algorithm formed with kernel estimators”, *Int. J. Appl. Math. Comput. Sci.* 20(1), 123–134 (2010).

[15] Y. Lei, Y. Dou, Y. Dong, J. Zhou and F. Xia, “FPGA implementation of an exact dot product and its application in variableprecision floating-point arithmetic”, *J. Supercomput.* 64(2), 580– 605 (2013).

[16] J. Matai, D. Richmond, D. Leey and R. Kastner, “Enabling FPGAs for the masses”, *1st Int. Workshop on FPGAs for Software Programmers, Munich*, arXiv:1408.5870 (2014).

[17] E. P. Ferlin, H. S. Lopes, C. R. Erig Lima and M. Perretto, “PRADA: a high-performance reconfigurable parallel architecture based on the dataflow model”, *Int. J. of High Performance Systems Architecture* 3(1), 41–55 (2011).

[18] A. Pułka and A. Milik, “An efficient hardware implementation of smith-waterman algorithm based on the incremental approach”, *International Journal of Electronics and Telecommunications* 57(4), 489–496 (2011).

[19] E. Y. Remez, “Sur la détermination des polynômes d’approximation de degré donnée”, *Comm. Soc. Math. Kharkov* 10, 41–63 (1934) [in French].

[20] M. Sawerwain and R. Gielerak, “GPGPU based simulations for one and two dimensional quantum walks”, *Computer Networks: 17th Conference, Ustroń*, 29–38 (2010).

[21] D.W. Scott, *Multivariate Density Estimation: Theory, Practice, and Visualization*, John Wiley & Sons, Inc. (1992).

[22] B. W. Silverman, *Density Estimation For Statistics And Data Analysis*, Chapman & Hall (1986).

[23] J. S. Simonoff, *Smoothing Methods in Statistics*, Springer, 1996.

[24] J. Spiechowicz, M. Kostur and L. Machura, “GPU accelerated Monte Carlo simulation of Brownian motors dynamics with CUDA”, *Computer Physics Communications* 191, 140–149 (2015).

[25] P. Steffen, R. Giegerich and M. Giraud, “GPU parallelization of algebraic dynamic programming”, *PPAM 2009*, LNCS 6068, 290–299 (2010).

[26] “Synflow Cx”, www.synflow.com, last access April 2015.

[27] S. Taherkhani, E. Ever and O. Gemikonakli, “Implementation of non-pipelined and pipelined data encryption standard (DES) using Xilinx Virtex-6 FPGA technology”, *IEEE 10th International Conference on Computer and Information Technology*, 1257–1262, (2010).

[28] M. P.Wand and M. C. Jones, *Kernel Smoothing*, Chapman & Hall (1995).

[29] B. Wyrwoł and E. Hryniewicz, “Decomposition of the fuzzy inference system for implementation in the FPGA structure”, *Int. J. Appl. Math. Comput. Sci.* 23(2), 473–483 (2013).

[30] “The PLUGIN source codes”, https://github.com/qMSUZ/ plugin (2016).