FPGA-Based Design of Numerical Algorithms for Kernel Density Estimation Using High Level Synthesis Approach

Artur Gramacki*, Marek Sawerwain*, and Jarosław Gramacki**

*Institute of Control and Computation Engineering
**Computer Centre
University of Zielona Góra, ul. Licealna 9, Zielona Góra 65-417, Poland
{a.gramacki,m.sawerwain}@issi.uz.zgora.pl
{j.gramacki}@ck.uz.zgora.pl

Abstract. FPGA technology can offer significantly higher performance at much lower power than is available from CPUs and GPUs in many computational problems. Unfortunately, programming for FPGA (using hardware description languages, HDL) 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 promoting by many 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 selected numerical algorithm (bandwidth selection for kernel density estimators, KDE) using HLS and show techniques which were used to optimise the final FPGA implementation.

Keywords: FPGA, High Level Synthesis (HLS), numerical algorithms, kernel density estimators, bandwidth selection

1 Introduction

The probability density function (PDF) is a key concept in statistics with many practical applications, see for example [7], [12]. Constructing the most adequate PDF from the observed data is still an important and interesting scientific 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 estimator (KDE) [17]. However, a very serious drawback of using KDEs is the large number of calculations required to compute them, especially to find the optimal bandwidth (smoothing) parameter. In this paper we investigate the possibility of utilizing field-programmable gate arrays (FPGA) to accelerate finding of the optimal bandwidth.

Towards the needs of the paper we have selected one popular and often used algorithm for calculating the optimal bandwidth (time complexity of $O(n^2)$). The method is called in literature PLUGIN [17]. This work can be considered as a continuation and extension of the paper [1], where authors utilize GPUs for

speeding up optimal bandwidth selection. One of the algorithm analyzed there was the above mentioned PLUGIN one.

Generally there are two methods 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 based on (a) computing on multicore CPUs, (b) computing on distributed environments (clusers, grids, etc.), (c) computing on GPUs, [14] and (d) computing on reconfigurable field-programmable gate arrays (FPGA) [8], [9], [11], [16] and [6], [18] as an example of using FPGA for solving some specific computational statistics problems.

In this paper we are concerned with FPGA approach only. To develop the final FPGA design we use the High Level Synthesis (HLS) approach [5], [10], where no direct hardware description language (HDL) coding is needed (typically in VHDL or Verilog languages[1]).

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

2 Kernel Density Estimation

Let us consider a continuous random variable X (in general d-dimensional) and let assume its probability density function (PDF) f exists but is unknown. Its estimate, usually denoted by $\hat{f}$, will be determined on the basis of a random sample of size n, that is $X_1, X_2, ..., X_n$ (our experimental data). In such a case, a d-dimensional kernel density estimator $\hat{f}(x, h)$ of a real density $f(x)$ for random sample $X_1, X_2, \ldots, X_n$ is given by the following formula

$$\hat{f}(x,h) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right). \tag{1}$$

h is a positive real number called *smoothing parameter* or *bandwidth* and $K(\cdot)$ is the *kernel function* – a symmetric function that integrates to one. In practical applications $K(\cdot)$ often has the Gaussian normal form, that is

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

[1] It is worth to note that OpenCL framework, which is commonly used by GPU programmers, becomes also 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++. However, in the time of writing this paper the tool is not yet generally available.

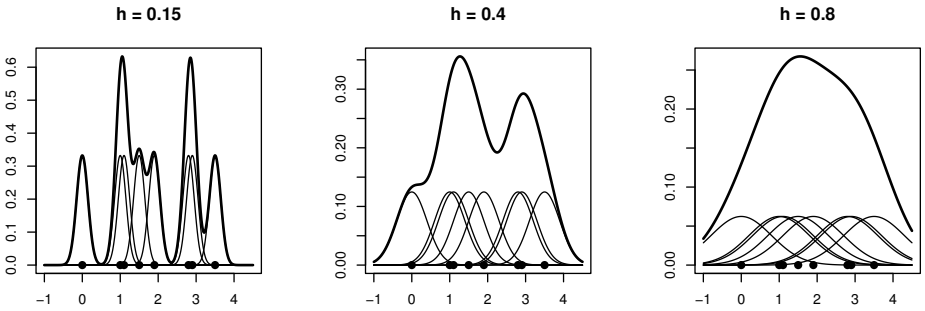

Fig. 1. An example of using kernel density estimators for determining the probability density function.

If we have the bandwidth h, we can determine the estimator $\hat{f}(x,h)$ of the unknown d-dimensional density function $f(x)$ using formula (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 8 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 Figure 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 $K(x)$ (Gaussians). Dots represent the data points X_i. Choosing the best value of h is not a trivial task and this problem was and still is extensively studied in literature [3], [4].

2.1 The PLUGIN Method and Data Preprocessing

In Algorithm 1 we give recipe for calculation of an optimal bandwidth using the PLUGIN method. All the necessary details on the method, as well as details on deriving of particular mathematical formulae can be found in many source materials, see for example books [12,13,17]. First we calculate the variance and the standard deviation estimators (3) of the input data. Then we calculate some more complex formulae (4) to (8). Finally, we can substitute them into equation (9) to get the searched optimal bandwidth h.

Our implementation of the algorithm is carried out in fixed-point arithmetic (see chapter 3.1). Unfortunately, using the raw data while conducting the required calculations, threatens a potential problems with overflow, especially while calculating the value of (4). Note that the estimate of standard deviation (3) in (4) 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 easily avoided if the input datasets are *standardized* using the *z-score* formula, that is

$$Z_i = \frac{X_i - \mu}{\sigma}, \tag{10}$$

where μ and σ are mean and standard deviation of the original vector X respectively. Z-score guarantees that $\hat{\sigma} = 1$ in (4) and, consequently, $\hat{\Psi}_8^{NS}$ entity has

Data: data set X, contains n elements
Result: value h represents the bandwidth for kernel density estimation
Step I: *Calculate estimates of variance* $\hat{V}$ *and standard deviation* $\hat{\sigma}$:

$$\hat{V} \leftarrow \frac{1}{n-1}\sum_{i=1}^{n} X_i^2 - \frac{1}{n(n-1)}\left(\sum_{i=1}^{n} X_i\right)^2, \quad \hat{\sigma} \leftarrow \sqrt{\hat{V}}. \tag{3}$$

Step II: *Calculate estimate* $\hat{\Psi}_8^{NS}$ *of functional* Ψ_8:

$$\hat{\Psi}_8^{NS} \leftarrow \frac{105}{32\sqrt{\pi}\hat{\sigma}^9}. \tag{4}$$

Step III: *Calculate value of bandwidth of kernel estimator of function* $f^{(4)}$ *(4th derivative of function* f*, that is* $f^{(r)} = \frac{d^r f}{dx^r}$*)*:

$$g_1 \leftarrow \left(\frac{-2K^6(0)}{\mu_2(K)\hat{\Psi}_8^{NS} n}\right)^{1/9}, \quad K^6(0) = -\frac{15}{\sqrt{2\pi}}, \quad \mu_2(K) = 1. \tag{5}$$

Step IV: *Calculate estimate* $\hat{\Psi}_6(g_1)$ *of functional* Ψ_6:

$$\begin{aligned} \hat{\Psi}_6(g_1) &\leftarrow \frac{1}{n^2 g_1^7}\left[\sum_{i=1}^{n}\sum_{j=1}^{n} K^{(6)}\left(\frac{X_i - X_j}{g_1}\right)\right], \\ K^6(x) &= \frac{1}{\sqrt{2\pi}}\left(x^6 - 15x^4 + 45x^2 - 15\right) e^{-\frac{1}{2}x^2}. \end{aligned} \tag{6}$$

Step V: *Calculate value of bandwidth of kernel estimator of function* $f^{(2)}$:

$$g_2 \leftarrow \left(\frac{-2K^4(0)}{\mu_2(K)\hat{\Psi}_6(g_1) n}\right)^{1/7}, \quad K^4(0) = \frac{3}{\sqrt{2\pi}}, \quad \mu_2(K) = 1. \tag{7}$$

Step VI: *Calculate estimate* $\hat{\Psi}_4(g_2)$ *of functional* Ψ_4:

$$\begin{aligned} \hat{\Psi}_4(g_2) &\leftarrow \frac{1}{n^2 g_2^5}\left[\sum_{i=1}^{n}\sum_{j=1}^{n} K^{(4)}\left(\frac{X_i - X_j}{g_2}\right)\right], \\ K^4(x) &= \frac{1}{\sqrt{2\pi}}\left(x^4 - 6x^2 + 3\right) e^{-\frac{1}{2}x^2}. \end{aligned} \tag{8}$$

Step VII: *Calculate the final value of bandwidth* h:

$$h \leftarrow \left(\frac{R(K)}{\mu_2(K)^2\hat{\Psi}_4(g_2) n}\right)^{1/5}, \quad R(K) = \frac{1}{2\sqrt{\pi}}, \quad \mu_2(K) = 1. \tag{9}$$

Algorithm 1: Main computational steps of the PLUGIN algorithm. It is necessary to stress that the PLUGIN algorithm is a strictly sequential computational process as every step depends on the results obtained in the previous steps.

simply a constant value. Another possible solution analyzed by the authors is vector *normalization*, that is

$$\hat{X} = \frac{X}{|X|}. \tag{11}$$

This, however, doesn't solve the problem with too big value of (4), but significantly reduce the variance (3).

Applying the data standardization requires an extra operation on the h value (9), that is simply

$$h_{\text{final}} = h \cdot \sigma. \tag{12}$$

Similarly, in the case of normalization we have

$$h_{\text{final}} = h \cdot |\mathbf{x}|, \tag{13}$$

where in both cases h is the bandwidth calculated for the normalized dataset.The correctness of the two above equations can be easily proofed algebraically.

To reduce the calculation burden we can also slightly change equations (6) and (8). It is easy to notice a symmetry, that is $K^{(6)}((X_i - X_j)/g_1) = K^{(6)}((X_j - X_i)/g_1)$. So, the double summations can be changed and, consequently, the final formulae for $\hat{\Psi}_6(g_1)$ has now the following form

$$\hat{\Psi}_6(g_1) \leftarrow \frac{1}{n^2 g_1^7}\left[2\left(\sum_{i=1}^{n}\sum_{j=1,i<j}^{n} K^{(6)}\left(\frac{X_i - X_j}{g_1}\right)\right) + nK^{(6)}(0)\right]. \tag{14}$$

Obviously, the same concern $K^{(4)}(\cdot)$ and $\hat{\Psi}_4(g_2)$.

3 FPGA-based Implementation

HLS is an automated design process that interprets an algorithmic description of a problem (given in relatively high level languages like C/C++) and translates this problem into a register-transfer level (RTL) HDL code. Than 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* [15].

3.1 Implementation Preliminaries

Before implementing the PLUGIN algorithm it is important to take some decisions affecting both performance and resource consumption.

The first decision is about the 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. From the other hand the fixed-point arithmetic is much less resource demanding but its range and precision are more limited. However, many practical experiments carried by the authors proof that the properly wide signed two's complement fixed-point numbers give accuracy similar to single-precision floating-point numbers. We decided to use $Q15.32$ representation (that is: whole part length $m = 15$, fractional part length $n = 32$, word length $N = 48$ and 1 bit for the sign). Such a representation gives the range from -2^{15} to $(-2^{15} - 2^{-32})$ with the resolution of 2^{-32}.

The second decision is about choosing the most adequate methods for calculating individual components of equations from (3) to (9) 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 typical logical blocks (Look Up Table (LUT), Flip-Flop (FF), Block RAM (BRAM), dedicated units for DSP tasks (DSP48E)) which can be connected each other and which can implement only low-level logical function (logical gates level). As a consequence, even very basic operations, like for examples the adder for adding two integers must be implemented from scratch. In our mathematical formulae one can easily indicate such operations like (a) addition, (b) subtraction, (c) multiplication, (d) division, (e) reciprocal, (f) exponent, (g) logarithm[2], (h) power, (i) square roots, (j) higher order roots.

Our implementation utilizes the following methods: *CORDIC* for calculating exponents and logarithms, divisions were replaced by multiplications and reciprocals, difference operators were replaced by summation of negative operands.

3.2 Implementation Details and Results

In Figure 2 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. Five different versions of the PLUGIN algorithm were implemented and tested. The summary is given in Table 1. All experimental results are given for one sample dataset. The dataset size in not critical as the implementations are strictly linearly scalable and the main goal was to show relative differences between FPGA and CPU implementations, not an absolute performance in clock cycles and/or (mili)seconds. The values for the *Clock* column were taken by direct counting of the total clock cycles presented by Vivado Design Suite during simulation (FPGA implementation) and by using the *perf* tool available for Linux kernels (CPU implementation)[3].

The first implementation is just a literal rewriting of Algorithm 1. No additional actions were taken toward optimization of both execution time and

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

[3] Processor Intel i7 4790k 4.0 GHz, Linux kernel v3.16.0 x86_64, gcc v4.9.1, compiler option used: `-O2 -march=corei7-avx`

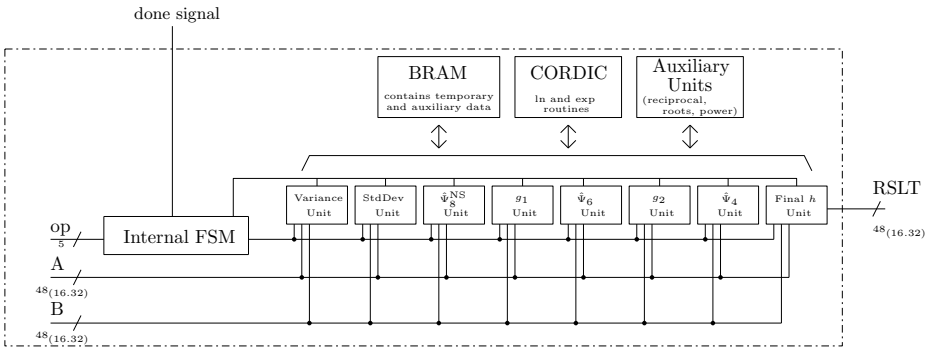

Fig. 2. Main units of FPGA-based PLUGIN algorithm implementation

Table 1. Resources usage for five different implementations of the PLUGIN algorithm. Vivado HLS 2014.4 was used for the FPGA implementations with target device Xilinx Artix-7 xc7a100tcsg324-1 and Intel i7 4790k 4.0 GHz for PCU implementations

Method	BRAM 18k	DSP48E	FF	LUT	Clock
Unoptimised direct FPGA implementation	0	1489	103688	98556	97598
Optimised FPGA version with revised normalisation routines	4	135	13296	17706	68027
CPU, fixed-point	-	-	-	-	4390820
CPU, floating-point – float	-	-	-	-	1546914
CPU, floating-point – double	-	-	-	-	1598769

resource requirements. This version can operate with any unscaled input data (assuming that all the inputs as well as all the internal results fulfill the fixed-point ranges that have been set). The improvement shown in (14) is not used. This version automatically (Vivado decides) utilizes pipelining, which results in relatively fast implementation but with very large number of DSP48E blocks usage. FFs and LUTs usage is also quite high.

The second version is an optimized one and requires the standardization of the input data. In this version both variance and standard deviation (3) are not required to be calculated. The following techniques were used during optimization process: (a) all constant values (used for example in CORDIC algorithm) were stored in BRAM, (b) two dedicated functions (*fADD*, *fMUL*) for realization of additions and multiplications were implemented (that gives possibility for closer monitoring of DSP48E usage), (c) Horner's method was used for calculating polynomials in (6) and (8), (d) a dedicated function for reciprocal was implemented and all explicit divisions were replaced by reciprocals and multiplications (usually it is more economical to calculate (a/b) as $(a \cdot reciprocal(b))$. Vivado HLS offers its own division implementation but our original one, based on reciprocal, gives better properties. (e) the improvement (14) was used. The

resource usage, comparing to the unoptimized version, is significantly reduced as well as the total clock cycles.

All CPU implementations are in fact literal rewriting of Algorithm 1 without the improvement (14). Fixed-point calculations were implemented using arbitrary precision integer and fixed-point data types supported by Vivado HLS (*ap_fixed.h* header). CPU implementations use several dozens more clock cycles comparing to FPGA implementations. Of course we must keep in mind that at the same time current CPUs clock rates are typically higher than current FPGAs (single GHz vs. hundreds of MHz). From the other hand, FPGAs typically use less energy than CPUs (single Watts vs. several dozens Watts).

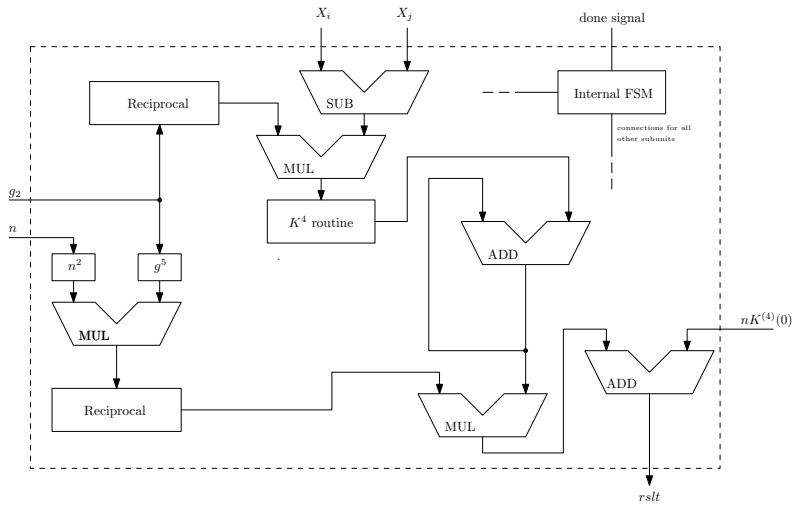

Fig. 3. General architecture of $\hat{\Psi}_4(g_2)$ unit (block-level view)

Figure 3 presents general architecture of the 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 above. In Figure 4 we show selected code fragments which implement $\hat{\Psi}_4(g_2)$ estimator. It must be also stressed that the most demanding calculations are those needed for setting of $\hat{\Psi}_6(g_1)$ and $\hat{\Psi}_4(g_2)$ estimators.

4 Conclusions

HLS tools are competitive with manual design techniques using HDLs. Implementation time of complex numerical algorithms can be essentially reduced. 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 like DSP48E, BRAM,

```
data_t fnc_k4x_FixPt(const data_t &x) {
 const data_t t1 = 0.3989422;

 data_t x2 = x*x, x4 = x2*x2;

 return (t1 * (x4 - 6.0*x2 + 3.0) * exp( ( -0.5 * x * x) ) ) ;
}

data_t estimate_psi4g2_FixPt(
     const data_t &v1,  const data_t &v2,
     const uint8_t &op ) {
 data_t _N;
 // ...

 data_t gg2;
 // ...

 if ( op == estpsi4_reset_op ) {
  _N = 0.0;
  // ...
 }

 if ( op == estpsi4_setup_op ) {
  g2 = v1;
  _N = v2;

  gg2 = (g2*g2);
  gg3 = (gg2*g2);
  gg4 = (gg3*g2);
  gg5 = (gg4*g2);

  t = (1.0 / ( (_N *_N) * gg5)) ;
  s = 0.0;

  rslt = s;
 }

 if ( op == estpsi4_col_op ) {
  s = s + fnc_k4x_FixPt( ( v1 - v2 ) / g2 );
  rslt = s;
 }

 if ( op == estpsi4_fin_op ) {
  rslt = t * s;
 }
 return rslt;
}
```

```
static data_t fADD(const data_t &a, const data_t &b) {
#pragma HLS INLINE off
   ...
}

data_t expFixPt ( const data_t &x ) {
 #pragma HLS INLINE off
 #pragma HLS RESOURCE variable=expFP_a core=ROM_1P_BRAM
 ...
}

data_t fnc_k4x_FixPt(const data_t &x) {
 const data_t t1 = 0.398942280401433;

 k4xFP_x2 = fMUL(x, x);
 k4xFP_x4 = fMUL(k4xFP_x2, k4xFP_x2);

 return fMUL(t1, fMUL(fADD(fADD(k4xFP_x4,
           -fMUL((data_t)6.0, k4xFP_x2)), (data_t)3.0),
           expFixPt( fMUL((data_t)-0.5, k4xFP_x2))) );
}

data_t estimate_psi4g2_FixPt( const data_t &v1,
       const data_t &v2,
       const uint8bit_t &op ) {
    uint8bit_t i;

    if ( op == estpsi4_reset_op ) {
          psi4g2_FP__N = (data_t)0.0;
          // ...
    }

    if ( op == estpsi4_setup_op ) {
        psi4g2_FP_g2 = v1;
        // ...

        estpsi4g2_loop: for(i=0;i<4;i++)
             psi4g2_FP_gg5 = fMUL(psi4g2_FP_gg5, v1);

        psi4g2_FP_t = reciprocalFixPt( fMUL(
            fMUL(psi4g2_FP__N, psi4g2_FP__N), psi4g2_FP_gg5) )  ;
        psi4g2_FP_s = (data_t)0.0;
        psi4g2_FP_rslt = psi4g2_FP_s;
    }

    if ( op == estpsi4_col_op ) {
            psi4g2_FP_s = fADD(psi4g2_FP_s, fnc_k4x_FixPt(
                 fMUL(fADD( v1, -v2 ),
                      reciprocalFixPt( psi4g2_FP_g2 )) ) );
            psi4g2_FP_rslt = (data_t) psi4g2_FP_s;
    }

    if ( op == estpsi4_fin_op ) {
            psi4g2_FP_rslt = fMUL(psi4g2_FP_t, psi4g2_FP_s);
    }

    return psi4g2_FP_rslt;
}
```

Fig. 4. Sample codes for unoptimized (left panel) and optimized (right panel) versions of routines for $\hat{\Psi}_4(g_2)$ estimator used in the PLUGIN algorithm.

FF and 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.

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

In our work we give priority on minimizing FPGA resources usage, rather than calculation times. This is because it gives a potential possibility to implement many instances of the same algorithm into one FPGA chip. As a consequence, it allows to do calculations simultaneously on different datasets.

References

1. Andrzejewski, W., Gramacki, A., Gramacki, J.: Graphics processing units in acceleration of bandwidth selection for kernel density estimation. Int. J. Appl. Math. Comput. Sci., Vol. 23, No. 4, pp. 869–885 (2013)
2. Bachrach, J. Vo, H., Richards, B. Lee, Y., Waterman, A., Avizienis, R., Wawrzynek, J., Asanovi, K.: Chisel: Constructing Hardware in a Scala Embedded Language, In: Design Automation Conference, IEEE, pp. 1212 – 1221, San Francisco (2012)
3. Chacón, J. E., Duong, T.: Multivariate plug-in bandwidth selection with unconstrained pilot bandwidth matrices. TEST (Springer), Vol. 19, Issue 2, pp 375–398 (2010)
4. Chacón, J. E., Duong, T.: Unconstrained pilot selectors for smoothed cross-validation. Aust. N. Z. J. Stat., Vol. 53, Issue 3, pp 331–351 (2012)
5. Coussy, P., Morawiec, A.: High-Level Synthesis From Algorithm to Digital Circuit. Springer, Heidelberg (2008)
6. Fahmy, S., A.: Architecture for Real-Time Nonparametric Probability Density Function Estimation. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 21, No. 5, pp. 910 – 920 (2013)
7. Kulczycki, S., Charytanowicz, M.: A complete gradient clustering algorithm formed with kernel estimators. Int. J. Appl. Math. Comput. Sci., Vol. 20, No. 1, pp. 123–134 (2010)
8. Lei, Y., Dou, Y., Dong, Y., Zhou, J., Xia, F.: FPGA implementation of an exact dot product and its application in variable-precision floating-point arithmetic. J. Supercomput., Vol. 64, Issue 2, pp. 580–605 (2013)
9. Lin Y., Wang, F., Zheng, X., Gao, H., Zhang L.: Monte Carlo simulation of the Ising model on FPGA, Journal of Computational Physics, Vol. 237, pp. 224–234 (2013)
10. Matai, J., Richmond, D., Leey, D., Kastner, R.: Enabling FPGAs for the Masses. 1st Int. Workshop on FPGAs for Software Programmers, Munich, Germany (2014), arXiv:1408.5870
11. Pedro Ferlin E., Silvério Lopes H., Erig Lima, C.R., Perretto, M.: PRADA: a high-performance reconfigurable parallel architecture based on the dataflow model, Int. J. of High Performance Systems Architecture, Vol. 3, No. 1, pp. 41-55 (2011)
12. Silverman, B.W.: Density Estimation For Statistics And Data Analysis. Chapman & Hall (1986)
13. Simonoff, J.S.: Smoothing Methods in Statistics. Springer Series in Statistics (1996)
14. Steffen, P., Giegerich, R., Giraud, M.: GPU Parallelization of Algebraic Dynamic Programming, PPAM 2009, LNCS 6068, pp. 290 – 299 (2010)
15. Synflow Cx, `www.synflow.com`, last access April 2015
16. Taherkhani, S.; Ever, E.; Gemikonakli, O.,: Implementation of Non-Pipelined and Pipelined Data Encryption Standard (DES) Using Xilinx Virtex-6 FPGA Technology, Computer and Information Technology (CIT), In: 10th International Conference on Computer and Information Technology, pp. 1257 – 1262, IEEE Press, Bradford (2010)
17. Wand, M.P., Jones, M.C.: Kernel Smoothing. Chapman & Hall (1995)
18. Wyrwoł, B., Hryniewicz, E.: Decomposition of the fuzzy inference system for implementation in the FPGA structure. Int. J. Appl. Math. Comput. Sci., Vol. 23, No. 2, pp. 473–483 (2013)