Int. J. Appl. Math. Comput. Sci., 2013, Vol. 23, No. 4, 869–885
DOI: 10.2478/amcs-2013-0065

GRAPHICS PROCESSING UNITS IN ACCELERATION OF BANDWIDTH SELECTION FOR KERNEL DENSITY ESTIMATION

WITOLD ANDRZEJEWSKI *, ARTUR GRAMACKI **, JAROSŁAW GRAMACKI ***

* Institute of Computer Science
Poznań University of Technology, Piotrowo 2, 60-965 Poznań, Poland
e-mail: Witold.Andrzejewski@cs.put.poznan.pl

** Institute of Computer Science and Electronics
University of Zielona Góra, Licealna 9, 65-417 Zielona Góra, Poland
e-mail: A.Gramacki@iie.uz.zgora.pl

*** Computer Center
University of Zielona Góra, Licealna 9, 65-417 Zielona Góra, Poland
e-mail: J.Gramacki@ck.uz.zgora.pl

The Probability Density Function (PDF) is a key concept in statistics. Constructing the most adequate PDF from the observed data is still an important and interesting scientific problem, especially for large datasets. PDFs are often estimated using nonparametric data-driven methods. One of the most popular nonparametric method is the Kernel Density Estimator (KDE). However, a very serious drawback of using KDEs is the large number of calculations required to compute them, especially to find the optimal bandwidth parameter. In this paper we investigate the possibility of utilizing Graphics Processing Units (GPUs) to accelerate the finding of the bandwidth. The contribution of this paper is threefold: (a) we propose algorithmic optimization to one of bandwidth finding algorithms, (b) we propose efficient GPU versions of three bandwidth finding algorithms and (c) we experimentally compare three of our GPU implementations with the ones which utilize only CPUs. Our experiments show orders of magnitude improvements over CPU implementations of classical algorithms.

Keywords: bandwidth selection, graphics processing unit, probability density function, nonparametric estimation, kernel estimation.

1. Introduction

The paper is about implementing some *Kernel Density Estimator* (KDE) bandwidth selection techniques with the support of Graphics Processing Units (GPUs). We propose modifications of classical algorithms to perform parallel and concurrent computations to accelerate very time-consuming operations while calculating the optimal KDE, which heavily depends on the so-called *bandwidth* or *smoothing parameter*. These estimators estimate the *Probability Density Function* (PDF), which is an elegant tool in the exploration and presentation of data.

One of the most serious drawbacks of practical usage of KDEs is that exact kernel nonparametric algorithms scale quadratically (see Section 3.2 for precise mathematical formulas). To overcome this problem, two main approaches may be used. In the first one, many authors propose various *approximate* methods for KDEs. We give a short summary of these methods in Section 2.1. In this paper we investigate the second approach, where *exact* methods for KDEs are used. The problem of fast computation of PDFs is especially important when we work which really big datasets, typically stored in (relational) databases. Among the huge amount of applications of PDFs there are important ones related to databases and data exploration. They can be successfully used, for example, in *Approximate Query Processing* (AQP) (Gramacki *et al.*, 2010), *query selectivity estimation* (Blohsfeld *et al.*, 1999) and *clustering* (Kulczycki and Charytanowicz, 2010).

In the paper we concentrate on the problem of finding an optimal bandwidth and implement and test only exact algorithms since they can be treated as reference ones and they always give the most accurate results. Such

exact implementation might be a useful tool, for example, for verification of the results returned by approximate methods.

To make the exact approach practically usable, all substantial time consuming computations are performed using massively parallel computing capabilities of modern GPUs. We compare this implementation (later called the *GPU implementation*) with two other implementations: an *SSE implementation* and a *sequential implementation*. The former utilizes two benefits of modern CPUs: SSEs (Streaming SIMD Extensions), allowing us to perform the same instructions on multiple values in parallel, as well as a multicore architecture, allowing us to process several threads in parallel on several distinct cores. The latter is a simple implementation of algorithms presented in Sections 3.2 and 5.1. The speedups we obtain using GPUs are in the range of one to three orders of magnitude in comparison with non-GPU-based approach.

The remainder of the paper is organized as follows. In Section 2 we cover the necessary background material. In Section 3 we turn our attention to some preliminary information on kernel estimators of PDFs and the potential using of PDFs in the database area. We also give detailed mathematical formulas for calculating optimal bandwidth parameters using three different methods. In Section 4 we provide a brief review of the GPU architecture. In Section 5 we cover all the necessary details on the optimization of the base algorithms. In Section 6 we show how to utilize the algorithms presented in previous sections. The practical experiments carried out are presented in Section 7. In Section 8 we conclude our paper.

2. Related works

2.1. Computation of probability density functions. The PDF is one of the most important and fundamental concepts in statistics and it is widely used in exploratory data analysis. There exist a lot of *parametric* forms of density functions. A very complete list of PDFs can be found, for example, in the works of Johnson *et al.* (1994; 1995). On the other hand, if the parametric form of the PDF is unknown or difficult to assume, one should consider *nonparametric* methods. One of the most popular nonparametric those by methods is the KDE. Three classical books on KDEs are those by Silverman (1986), Simonoff (1996), as well as Wand and Jones (1995).

There are two main computational problems related to the KDE: (a) *evaluating the kernel density estimate* $\hat{f}(x,h)$ (or $\hat{f}(x,H)$; see Section 3.1 for a brief explanation on differences between h and H), (b) *finding an optimal bandwidth parameter* h (or H), which is investigated in this paper.

A plethora of techniques have been proposed for accelerating computational times of the first problem. The naive direct evaluation of the KDE at m evaluation points for n source points requires $O(mn)$ kernel evaluations. Evaluation points can be, of course, the same as source points, and then the computational complexity is $O(n^2)$. The most commonly used method to reduce the computational burden for the KDE is the technique known as *binning* or *discretising* (Wand and Jones, 1995). In such a case, the density estimation is evaluated at grid points $g \ll n$, rather than source points. The binning strategy reduces the required kernel evaluations to $O(mg)$ (or to $O(g)$ if the evaluation points are the same as grid points). Another approach is based on using the Fast Fourier Transformation (FFT), as proposed by Silverman (1982). Using the FFT requires that the source points are on an evenly spaced grid and then one can evaluate the KDE at an evenly spaced grid in $O(n \log n)$. If the source points are irregularly spaced, the pre-binning strategy should be applied first. The resulting KDE is also evaluated at regular evaluation points. If irregular target points are required, a sort of interpolation based on neighboring evaluation points should be applied. In the FFT-based approach, however, there is a potential setback connected with an aliasing effect which is not completely binning. This problem is investigated in detail by Hendriks and Kim (2003). Another technique is based on the Fast Gauss Transform (FGT) introduced by Greengard and Strain (1991), and can be viewed as an extension of the Improved Fast Gauss Transform (IFGT) (Yang *et al.*, 2003). The method is called by the authors ϵ-*exact* (Raykar *et al.*, 2010) in the sense that the constant hidden in $O(m+n)$ depends on the desired accuracy which can be chosen arbitrarily. Some preliminary work using the GPU approach was done by Michailidis and Margaritis (2013). However, the authors miss the most important part: bandwidth optimization, which is the main subject of this paper.

As for the problem of accelerating computational times for finding the optimal bandwidth h (or H), relatively less attention is paid in the literature. An attempt at using the Message Passing Interface (MPI) was presented by Łukasik (2007). Raykar and Duraiswami (2006) give an ϵ-*exact* algorithm (the idea is similar to the one presented by Raykar *et al.* (2010)) to accelerate finding the optimal bandwidth.

2.2. Bandwidth selection for kernel density estimates. The bandwidth selection problem is probably the most important one in the KDE area. Currently available selectors can be roughly divided into three classes (Sheather, 2004; Silverman, 1986; Wand and Jones, 1995). In the first class one uses very simple and easy to calculate mathematical formulas. They were developed to cover a wide range of situations but do not guarantee being close enough to the optimal bandwidth. They are often

called *rules-of-thumb* methods. The second class contains methods based on *least squares* and *cross-validation* ideas as well as 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. In the paper these methods will be abbreviated as *LSCV*. The third class contains methods based on plugging in estimates of some unknown quantities that appear in formulas for the asymptotically optimal bandwidth. Here we abbreviate these methods as *PLUGIN*.

3. Mathematical preliminaries

In this section we give some preliminary information on kernel density estimation as well as on how to use it in the database area. Most of this section is devoted to precise recipes for calculation of the optimal bandwidth h and H. We also propose some slight but important modifications of the reference mathematical formulas (see Section 5). The modifications play a very important role during GPU-based and SSE-based fast algorithm implementations for calculating the bandwidth.

3.1. Kernel density estimation. Let us consider a continuous random variable X (in general d-dimensional), and let us assume its PDF f exists but is not known. Its estimate, usually denoted by $\hat{f}$, will be determined on the basis of a random sample of size n, that is, $X_1, X_2, \ldots, X_n$ (our experimental data). In such a case, the d-dimensional kernel density estimator $\hat{f}(x,h)$ of the actual density $f(x)$ for the random sample $X_1, X_2, \ldots, X_n$ is given by the following formula:

$$\begin{aligned}\hat{f}(x,h) &= n^{-1}\sum_{i=1}^{n} K_h\left(x - X_i\right)\\ &= n^{-1}h^{-d}\sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right),\end{aligned} \tag{1}$$

where

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

and $K(\cdot)$ is the *kernel function*-a symmetric function that integrates to one. In practical applications, $K(\cdot)$ is very often the Gaussian normal formula

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

Here n is the number of samples, d is the task dimensionality, $x = (x_1, x_2, \ldots, x_d)^T$, $X_i = (X_{i1}, X_{i2}, \ldots, X_{id})^T$, $i = 1, 2, \ldots, n$, X_{ij} is the i-th observation of the random variable X_j, h is a positive real number called the smoothing parameter or bandwidth. Other commonly used kernel functions are Epanechnikov, uniform, triangular and biweight (Li and Racine, 2007). One can prove that selection of a particular kernel function is not critical in most practical applications as all of them guarantee obtaining similar results (Wand and Jones, 1995). However, in some applications, like, e.g., query selectivity estimation (Blohsfeld *et al.*, 1999), kernels with finite support may be more adequate. On the other hand, the bandwidth is the parameter which exhibits a strong influence on the resulting estimate. If we have bandwidth h, we can determine the estimator $\hat{f}(x,h)$ of the unknown d-dimensional density function $f(x)$ using the formula (1).

Equation (1) assumes that the bandwidth h is the scalar quantity, independently of the problem dimensionality. This is the simplest and the least complex variant of the bandwidth. On the other hand, the most general and complex version assumes that the so-called *bandwidth matrix* H is used instead of the bandwidth scalar h. The size of the matrix H is $d \times d$. This matrix is positive definite and symmetric by definition. So, an equivalent of the formula (1) is now defined as follows:

$$\begin{aligned}\hat{f}(x,H) &= n^{-1}\sum_{i=1}^{n} K_H\left(x - X_i\right)\\ &= n^{-1}|H|^{-1/2}\sum_{i=1}^{n} K\left(H^{-1/2}(x - X_i)\right),\end{aligned} \tag{4}$$

where

$$K_H(u) = |H|^{-1/2}K(H^{-1/2}u) \tag{5}$$

and $K(\cdot)$ is defined by (3).

A version between the simplest and the most complex is also considered in the literature (Duong, 2004) and is based on simplifying the unconstrained bandwidth matrix H to its constrained equivalent where all the off-diagonal entries are zeros by definition. In the paper we do not consider this case.

3.2. Formulas for bandwidth selection. In the following section we present detailed mathematical formulas for calculating bandwidths (optimal in some sense) using the PLUGIN and the LSCV methods mentioned earlier. The PLUGIN method is designed only for 1D problems, known in the literature as *univariate* problems. However, the method can be used also for solving nD problems when using the so-called *product kernel* approach. Then, the PLUGIN method can be used d times, for each dimension separately. Contrary to PLUGIN, the LSCV method is designed for both 1D and nD problems (known in the literature as *multivariate* problems). Three different LSCV versions can be considered, while only two were implemented by the authors. The simplest one (with the smallest

computational complexity) assumes that the bandwidth h is the scalar quantity, independently of the problem dimensionality (see Eqn. (1)). On the other hand, the most computational complex version assumes that the bandwidth matrix H is used, instead of the scalar h (see Eqn. (4)). This matrix is positive definite and symmetric.

Below, in the next three subsections, we give very condensed recipes for calculating the optimal bandwidth using the PLUGIN and two LSCV approaches: the simplified one for evaluating optimal h (that is, if the density estimate is calculated from (1); this variant will be called LSCV_h) and the general multivariate variant of the method (that is, if the density estimate is calculated from (4); this variant will be called LSCV_H). All the necessary details on the methods as well as details on deriving particular formulas can be found in many source materials (e.g., Silverman, 1986; Simonoff, 1996; Wand and Jones, 1995). Particular formulas for PLUGIN and LSCV_h approaches were adopted directly from Kulczycki (2005; 2008) with only a slight change in some symbols to make them compatible with the classical book by Wand and Jones (1995).

The formulas given in the next three subsections assume that the Gaussian kernel is used. Extension toward supporting other kernel types is also possible. However, we must point out that for non-Gaussian kernels the convolution formulas given by (16) and (23) have no analytical forms and numerical methods based on approximation theory should be used. Details can be found in the work of Kulczycki (2005).

All the methods presented below determine the optimal bandwidth on the basis of the input random variable and commonly used optimality criterion based on minimization of the *Mean Integrated Squared Error* (MISE) and its asymptotic approximation (AMISE).

3.2.1. PLUGIN. This method is used for calculating an optimal bandwidth h for univariate problems (using it for multivariate problems is briefly indicated in Section 3.2). First we calculate the variance and the standard variation estimators of the input data (Eqn. (6)). Then we calculate some more complex formulas (Eqns. (7)–(11)). The explanation of their essence is beyond the scope of the paper and can be found in many books on kernel estimators, e.g., the three above-mentioned books. Finally, after completing all the necessary components we can substitute them into Eqn. (12) to get the sought optimal bandwidth h.

1. Calculate the value of variance $\hat{V}$ and standard deviation $\hat{\sigma}$ estimators:

$$\hat{V} = \frac{1}{n-1}\sum_{i=1}^{n} X_i^2 - \frac{1}{n(n-1)}\left(\sum_{i=1}^{n} X_i\right)^2,$$

$$\hat{\sigma} = \sqrt{\hat{V}}. \tag{6}$$

2. Calculate the estimate $\hat{\Psi}_8^{NS}$ of the functional Ψ_8:

$$\hat{\Psi}_8^{NS} = \frac{105}{32\sqrt{\pi}\hat{\sigma}^9}. \tag{7}$$

3. Calculate the value of the bandwidth of the kernel estimator of the function $f^{(4)}$ (4th derivative of the function f, that is, $f^{(r)} = \mathrm{d}^r f/\mathrm{d}x^r$):

$$g_1 = \left(\frac{-2K^6(0)}{\mu_2(K)\hat{\Psi}_8^{NS} n}\right)^{1/9},$$
$$K^6(0) = -\frac{15}{\sqrt{2\pi}}, \quad \mu_2(K) = 1. \tag{8}$$

4. Calculate the estimate $\hat{\Psi}_6(g_1)$ of the functional Ψ_6:

$$\hat{\Psi}_6(g_1) = \frac{2}{n^2 g_1^7}\sum_{i=1}^{n}\sum_{j=1,i<j}^{n} K^{(6)}\left(\frac{X_i - X_j}{g_1}\right) + nK^{(6)}(0),$$
$$K^6(x) = \frac{1}{\sqrt{2\pi}}\left(x^6 - 15x^4 + 45x^2 - 15\right)e^{-\frac{1}{2}x^2}. \tag{9}$$

5. Calculate the bandwidth of the kernel estimator of the function $f^{(2)}$:

$$g_2 = \left(\frac{-2K^4(0)}{\mu_2(K)\hat{\Psi}_6(g_1) n}\right)^{1/7},$$
$$K^4(0) = \frac{3}{\sqrt{2\pi}}, \quad \mu_2(K) = 1. \tag{10}$$

6. Calculate the estimate $\hat{\Psi}_4(g_2)$ of the functional Ψ_4:

$$\hat{\Psi}_4(g_2) = \frac{2}{n^2 g_2^5}\sum_{i=1}^{n}\sum_{j=1,i<j}^{n} K^{(4)}\left(\frac{X_i - X_j}{g_2}\right) + nK^{(4)}(0),$$
$$K^4(x) = \frac{1}{\sqrt{2\pi}}\left(x^4 - 6x^2 + 3\right)e^{-\frac{1}{2}x^2}. \tag{11}$$

7. Calculate the final value of the bandwidth h:

$$h = \left(\frac{R(K)}{\mu_2(K)^2\hat{\Psi}_4(g_2) n}\right)^{1/5},$$
$$R(K) = \frac{1}{2\sqrt{\pi}}, \quad \mu_2(K) = 1. \tag{12}$$

3.2.2. LSCV_h. This method is used for calculating the optimal bandwidth h for both univariate and multivariate problems (determined by the parameter d), that is, applicable to the formula (1). The task is to minimize the objective function (13) which will be minimized according to the sought bandwidth h. After determining the search range of the bandwidth h (Eqn. (18)) as well as the starting bandwidth h (Eqn. (17)), we search for a h which minimizes the objective function (13).

1. Calculate covariance matrix Σ, its determinant $\det(\Sigma)$ and its inverse Σ^{-1}.
2. Let X_i be the i-th column of the input matrix X. The LSCV_h method yields minimization of the objective function $g(h)$:

$$g(h) = h^{-d}\Big[2n^{-2}\sum_{i=1}^{n}\sum_{j=1,i<j}^{n} T\left(\frac{X_i - X_j}{h}\right) + n^{-1}R(K)\Big], \tag{13}$$

where

$$T(u) = (K * K)(u) - 2K(u), \tag{14}$$

$$K(u) = (2\pi)^{-d/2}\det(\Sigma)^{-1/2}\exp\left(-\frac{1}{2}u^T\Sigma^{-1}u\right), \tag{15}$$

$$(K * K)(u) = (4\pi)^{-d/2}\det(\Sigma)^{-1/2}\exp\left(-\frac{1}{4}u^T\Sigma^{-1}u\right). \tag{16}$$

3. Calculate the approximate value of the bandwidth h:

$$h_0 = \left(\frac{R(K)}{\mu_2(K)^2 R(f'')n}\right)^{1/(d+4)},$$
$$\frac{R(K)}{\mu_2(K)^2} = \frac{1}{2^d\pi^{d/2}d^2}, \quad R(f'') = \frac{d(d+2)}{2^{d+2}\pi^{d/2}}. \tag{17}$$

4. Let the range in which we search for a minimum of $g(h)$ be heuristically found as

$$Z(h_0) = [h_0/4, 4h_0]. \tag{18}$$

The optimal bandwidth h is equal to

$$\arg\min_{h\in Z(h_0)} g(h). \tag{19}$$

3.2.3. LSCV_H. This method is used for calculating the optimal bandwidth matrix H for both univariate and multivariate problems, that is, applicable to the formula (4). In this variant of the LSCV method the objective function is defined by Eqn. (20). Now our goal is to find such a bandwidth matrix H which will minimize this objective function. This is a classical nonlinear optimization problem and it can be solved by using, for example, the well-known Nelder–Mead method (Nelder and Mead, 1965). This method needs a starting matrix which can be calculated from the *rule-of-thumb* heuristic equation (25) of Wand and Jones (1995).

We know that the bandwidth matrix H is always symmetric and its size is $d \times d$. So, only $d(d+1)/2$ independent entries exist. As a consequence, there is no need to evaluate the objective function for the full H matrix. It is sufficient to use $\mathrm{vech}(H)$, where vech (vector half) operator takes a symmetric $d \times d$ matrix and stacks the lower triangular half into a single vector of length $d(d+1)/2$.

1. Let

$$g(H) = 2n^{-2}\sum_{i=1}^{n}\sum_{j=1,i<j}^{n} T_H(X_i - X_j) + n^{-1}R(K), \tag{20}$$

where

$$T_H(u) = (K * K)_H(u) - 2K_H(u), \tag{21}$$

$$K_H(u) = (2\pi)^{-d/2}|H|^{-1/2}\exp\left(-\frac{1}{2}(u)^T H^{-1}(u),\right) \tag{22}$$

$$(K * K)_H(u) = (4\pi)^{-d/2}|H|^{-1/2}\exp\left(-\frac{1}{4}(u)^T H^{-1}(u)\right), \tag{23}$$

$$R(K) = 2^{-d}\pi^{-d/2}|H|^{-1/2}. \tag{24}$$

2. Find H which minimizes $g(H)$. Start from

$$H_{\text{start}} = (4/(d+2))^{1/(d+4)}n^{-1/(d+4)}\Sigma^{1/2}. \tag{25}$$

4. General processing on graphics processing units

Modern graphics cards are powerful devices performing highly parallel single-instruction multiple-data

computations. Newly developed APIs such as NVIDIA CUDA (NVIDIA Corporation, 2012) and OpenCL allow us to relatively easily develop programs which utilize this computing power for computations not necessarily related to computer graphics (Sawerwain, 2012). We prepared our implementations using NVIDIA CUDA API, which is very universal and has implementations specific for NVIDIA GPUs as well as for other platforms (Farooqui *et al.*, 2011). As we implemented and tested our solutions on NVIDIA GPUs, below we give a short description of these GPUs and their capabilities.

NVIDIA GPUs are composed of many multiprocessors. Each multiprocessor (SM) is composed of several streaming processors (SP). Each streaming processor is capable of performing logical and integer operations as well as single precision floating point ones. Each multiprocessor contains a number of warp schedulers (dependent on graphics cards' architecture). Each warp scheduler may contain one or two dispatch units which execute one or two independent instructions in groups of SPs. Consequently, each SP in a group performs the same instruction at the same time (SIMD). Current graphics cards may contain up to 15 SMs with 192 SPs each (Kepler architecture (NVIDIA Corporation, 2013)). This shows that the GPUs are capable of running thousands of threads in parallel (and even more concurrently). Each NVIDIA graphics card has an assigned compute capability (denoted as CC for brevity) version which specifies which features are supported by the given graphics card. Each multiprocessor also contains a small but fast on-chip memory called *shared memory*.

The tasks are not assigned to SMs directly. Instead, the programmer first creates a function called a kernel, which consists of a sequence of operations which need to be performed concurrently in many threads. To distinguish them from kernels used in kernel-based density estimates, we will call these functions *gpu-kernels*. Next, the threads are divided into equally sized *blocks*. A block is a one, two or three dimensional array of at most 1024 threads (or 512 threads on graphics cards with CC$\leq$1.3), where each thread can be uniquely identified by its position in this array. The obtained blocks form the so-called *computation grid*. When the gpu-kernel is started, all of the blocks are automatically distributed among the SMs. Each SM may process more than one block, though one block may be executed at only one SM. Threads in a single block may communicate by using the same portion of the SM shared memory. Threads run in different blocks may only communicate through the very slow *global memory* of the graphics card. Synchronization capabilities of the threads are limited. Threads in a block may be synchronized but global synchronization is not possible, though a (costly) workaround exists. Threads in a block are executed in 32 thread SIMD groups called warps (this is a consequence of having one warp scheduler per several SPs). Consequently, all of these threads should perform the same instruction. If the threads with a warp perform different code branches, all branches are serialized and threads not performing the branch are masked.

The programmer implementing an algorithm using CUDA API must take into account all of these low level GPU limitations to obtain the most efficient implementations. The GPU implementations of algorithms computing the smoothing coefficient presented in this paper adhere to all of above given efficient CUDA programming rules.

5. Optimization of basic algorithms

In this section we describe optimizations of standard equations presented in Section 3.2. First, we identify some compute-intensive parts in the mathematical formulas. Then we present some details on the parallel algorithm for the most costly parts of the LSCV method (which is the most computer power demanding method). For details on other methods, please refer to the work of Andrzejewski *et al.* (2013).

5.1. Basic formula modifications. The equations for algorithm LSCV_h may be reformulated to require fewer operations. Let us consider Eqn. (15):

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

As can be determined from Eqn. (13), u is always equal to $(X_i - X_j)/h$. Let us therefore reformulate Eqn. (15):

$$\begin{aligned}\tilde{K}(v) &= K(v/h)\\ &= (2\pi)^{-d/2}|\Sigma|^{-1/2}\exp\left(-\frac{1}{2}\frac{v}{h}^T\Sigma^{-1}\frac{v}{h}\right)\\ &= (2\pi)^{-d/2}|\Sigma|^{-1/2}\exp\left(-\frac{1}{2}\frac{1}{h^2}v^T\Sigma^{-1}v\right).\end{aligned} \tag{26}$$

Set

$$S(v) = v^T\Sigma^{-1}v. \tag{27}$$

If we substitute this into Eqn. (26), we obtain

$$\tilde{K}(v) = (2\pi)^{-d/2}|\Sigma|^{-1/2}\exp\left(-\frac{1}{2}\frac{1}{h^2}S(v)\right). \tag{28}$$

Analogous changes can be made to Eqn. (16):

$$\begin{aligned}&(\tilde{K} * \tilde{K})(v)\\ &\quad = (4\pi)^{-d/2}|\Sigma|^{-1/2}\exp\left(-\frac{1}{4}\frac{1}{h^2}S(v)\right).\end{aligned} \tag{29}$$

These changes can be next propagated to Eqns. (13) and (14):

$$\tilde{T}(v) = T\left(\frac{v}{h}\right) = (\tilde{K} * \tilde{K})(v) - 2\tilde{K}(v), \tag{30}$$

$$\begin{aligned} g(h) &= h^{-d}\Big[2n^{-2}\sum_{i=1}^{n}\sum_{j=1,i<j}^{n}\tilde{T}\left(X_i - X_j\right) \\ &\quad + n^{-1}R(K)\Big] \end{aligned} \tag{31}$$

It is easy to notice that the values of $S(v)$ are scalars, and moreover, they are constant, independent of the value of the parameter h of the function $g(h)$. Consequently, they may be precalculated for each combination of two vectors X at the start of the algorithm and used multiple times during the search for the minimum of $g(h)$.

Let us determine the complexity of evaluating the function $g(h)$ before and after modifications. Calculating a single exponent value in either $K(u)$ or $(K * K)(u)$ requires $O(d^2)$ operations. These functions need to be calculated $O(n^2)$ times, which leads to the complexity of $O(n^2d^2)$. Let n_h be the number of times the function $g(h)$ needs to be calculated during the search for its minimum. The total complexity of the unmodified algorithm LSCV_h is therefore $O(n_h n^2 d^2)$

Precalculating a single value of $S(v)$ requires $O(d^2)$ operations. As $n(n-1)/2$ of $S(v)$ values need to be precomputed, the complexity of precomputing of these values is $O(n^2d^2)$. However, since the values of $S(v)$ may be reused, computing $g(h)$ has only the complexity of $O(n^2)$. Consequently, the total complexity of the modified algorithm LSCV_h is $O(n^2(d^2 + n_h))$.

The approach described above cannot unfortunately be used for optimizing the algorithm LSCV_H. This is due to the fact that the expression $(X_i - X_j)^T H^{-1}(X_i - X_j)$ found in Eqns. (22) and (23) (which is an equivalent of $S(v)$) depends on H. Consequently, this expression must be recomputed each time the function $g(H)$ is evaluated.

5.2. Identification of some compute-intensive parts in the mathematical formulas. Let us take a closer look at Eqns. (6), (9), (11), (20) and (31). All of these (among other operations) compute sums of a large number of values. As such, sums are performed multiple times and constitute a large part of the number of basic mathematical operations computed in these equations. Accelerating them would significantly increase the algorithm performance. Consequently, in general, we need an algorithm which, for a given single-row matrix A, would compute

$$R(A) = \sum_{i=1}^{n} A_i. \tag{32}$$

The process of using the same operation multiple times on an array of values to obtain a single value (sum, multiplication, minimum, maximum, variance, count, average, etc.) is called the *reduction of an array*. Parallel reduction of large arrays is a known problem (Harris, 2007; Xavier and Iyengar, 1998).

Let us consider the first equation in (6). It contains two sums. One of these sums is a sum of values of a scalar function computed based on values stored in a matrix. Formally, given a single-row matrix A and a function $fun(x)$, this sum is equivalent to

$$R_{fun}(A) = \sum_{i=1}^{n} fun(A_i). \tag{33}$$

Parallel computation of such sums can be performed by using a simple modification of parallel reduction algorithms. We omit here the technical details.

Let us now consider Eqns. (9) and (11). Given a single row matrix A of size n and a function $fun(x)$, both of these equations contain double sums of function values equivalent to

$$RR_{fun}(A) = \sum_{i=1}^{n}\sum_{j=1,i<j}^{n} fun(A_i - A_j). \tag{34}$$

As can be easily noticed, the function $fun(x)$ is evaluated for differences between combinations of values from the single-row matrix A. These differences may be computed using an algorithm similar to the matrix multiplication algorithm presented by NVIDIA Corporation (2012). Consecutive reduction of the obtained values is performed using the known and previously mentioned algorithm. To cope with the fact that we compute differences between each pair of values (we compute a triangle matrix of differences), we utilize a scheme similar to the one used for accelerating LSCV methods (see Section 5.3).

Similar sums can also be found in Eqns. (20) and (31). These sums, given any matrix A and the function $fun(x)$, are equivalent to

$$RR^{v}_{fun}(A) = \sum_{i=1}^{n}\sum_{j=1,i<j}^{n} fun(A_{:,i} - A_{:,j}),^{1} \tag{35}$$

where $A_{:,i}$ is the i-th column of matrix A. In these equations, however, each argument of the function $fun(x)$ is a vector and evaluation of this function is much more complex. Moreover, in both cases the function $fun(x)$ can be expressed as $fun(x) = fun1(fun2(x))$, where

$$fun2(x) = x^T M x, \tag{36}$$

M is any matrix and $fun1(y)$ is any scalar function.

[1]The v superscript stands for *vector*.

Consider now Eqn. (31). Here, the function $fun(x)$ is an equivalent of the function $\tilde{T}(v)$ presented in Eqn. (30). The function $\tilde{T}(v)$ is evaluated using functions $\tilde{K}(v)$ (Eqn. (28)) and $(\tilde{K} * \tilde{K})(v)$ (Eqn. (29)). These functions, in turn, can be evaluated based on the value of the function $S(v)$ (Eqn. (27)), which is an equivalent of $fun2(x)$. As was mentioned in Section 5.1, the $S(v)$ values can be precomputed and used each time Eqn. (31) is computed. We can therefore split the problem of evaluating the sums in Eqn. (31) into two problems: (a) evaluating $fun2(x)$ ($S(v)$) and (b) finding a sum of values of a scalar function introduced earlier.

Similar observations can also be made for Eqn. (20). Here, the function $fun(x)$ is an equivalent of the function $T_H(X_i - X_j)$ presented in Eqn. (21). The function $T_H(X_i - X_j)$ is evaluated using functions $K_H(X_i - X_j)$ (Eqn. (22)) and $(K * K)_H(X_i - X_j)$ (Eqn. (23)). Exponents of both functions $K_H(X_i - X_j)$ and $(K * K)_H(X_i - X_j)$ contain $(X_i - X_j)^T H^{-1}(X_i - X_j)$, which is an equivalent of $fun2(x)$. Unfortunately, here the values of $fun2$ cannot be precomputed as matrix H^{-1} is different every time Eqn. (20) is evaluated.

5.3. Parallel processing of computation intensive parts of selected formulas of LSCV_H. Let us recall Eqns. (20) and (31). Both contain sums which are equivalent to

$$\begin{aligned} sum(A) &= \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} fun(A_{:,i} - A_{:,j}) \\ &= \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} fun1(fun2(A_{:,i} - A_{:,j})), \end{aligned} \tag{37}$$

where A is any $d \times n$ matrix, $fun1$ is any scalar function and $fun2(x) = x^T M x$, where M is also any matrix.

As was suggested in Section 5.2, the parallel processing of such sums can be split into two problems: evaluating $fun2$ and then reducing the resulting values using the previously introduced algorithm. Note also that n (the size of the database sample) can be very large. Consequently, these equations (among other operations) compute sums of a large number of values. As such sums are performed multiple times and constitute a large part of the basic mathematical operations performed in these equations, accelerating them would significantly increase the algorithm performance. The process of using the same operation multiple times on an array of values to obtain a single value (sum, multiplication, minimum, maximum, variance, count, average, etc.) is called the reduction of an array. Parallel reduction of large arrays is a known problem (Xavier and Iyengar, 1998) and has an efficient solution for GPUs (Harris, 2007). Before the reduction of values is performed, however, the values of function fun need to be computed.

Consequently, we need an algorithm which, for a given A matrix, would find a triangular matrix $B = [b_{i,j}]$, $i = 1, \ldots, n$, $j = i+1, \ldots, n$, such that

$$b_{i,j} = fun(A_{:,i} - A_{:,j}) = fun1(fun2(A_{:,i} - A_{:,j})). \tag{38}$$

As each value of fun function value may be computed independently, the parallel computation algorithm seems obvious. We propose a scheme derived from the matrix multiplication algorithm presented by NVIDIA Corporation (2012). Consider the scheme presented in Fig. 1. A matrix is divided into chunks of k vertical vectors (columns). The triangular result matrix is divided into tiles of size $k \times k$ (notice that tiles on the matrix diagonal contain excessive positions). Each tile corresponds to some combination of two chunks of matrix A. The row of a tile within the triangular matrix is indexed by q and the column is indexed by l. For each tile, a group of $k \times k$ threads is started. First, a subset of threads in a block copies the corresponding chunks into the cache memory. Next, each thread in the tile computes a function value based on two vectors retrieved from the cached chunks. Each thread detects whether or not it is over or on the main diagonal. If it is below the diagonal, it stops further computations. If it is over the main diagonal, it computes of value fun and stores it in the output array. The linear position in the output array may be computed using the equation for a sum of the arithmetic progression, based on the thread coordinates within the triangular array. Notice that the order of stored values is unimportant, as they are subsequently only arguments for functions whose values are later reduced (summed up).

First, as $fun(\vec{x}) = fun1(fun2(\vec{x}))$ and the $fun1$ function is scalar, it is easy to notice that the main problem is parallel evaluation of $fun2$. When a thread finalizes computation of $fun2$, it can use the obtained value to compute $fun1$ and therefore fun. Consequently, we shall now derive an efficient order of performing operations needed to evaluate $fun2$ within a tile for an array storing A that is row-major aligned in the computers' memory.

For simplicity, let us assume that the tile considered does not lie on the main diagonal. We shall denote the tile of the matrix containing the values of fun as a $k \times k$ submatrix Y. Each such tile corresponds to two $d \times k$ (d is the number of rows in A) chunks E and F of the matrix A. Let us assume that chunks E and F start at columns $qk+1$ and $lk+1$, respectively.

Let

$$i = qk + r, \quad j = lk + p, \tag{39}$$

where $r, p = 1, \ldots, k$. Consequently, $A_{:,i} = A_{:,qk+r} = E_{:,r}$ and $A_{:,j} = A_{:,lk+p} = F_{:,p}$. Let $v^{r,p} = E_{:,r} - F_{:,p} = A_{:,i} - A_{:,j}$ be all of the arguments of the $fun2$ function within a tile. From the definition of the function $fun2$ and

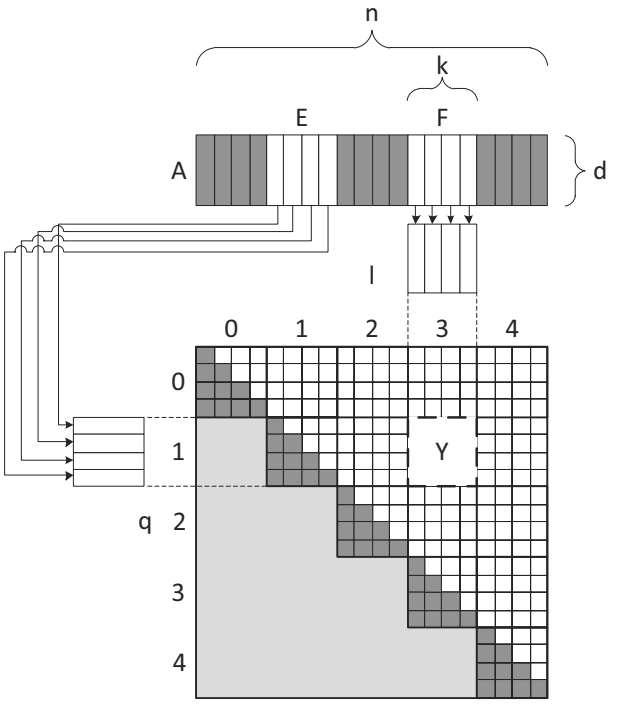


Fig. 1. Parallel computation of the fun function values.

the new notation introduced above, we know that

$$y_{r,p} = fun2(v^{r,p}) = (v^{r,p})^T M v^{r,p}. \quad (40)$$

Let us extract the first matrix multiplication operation from Eqn. (40):

$$z^{r,p} = (v^{r,p})^T M. \quad (41)$$

The $z^{r,p}$ value is a horizontal vector of d scalars:

$$z^{r,p} = [z_a^{r,p}]_{a=1,\ldots,d}, \quad (42)$$

where each value $z_a^{r,p}$ is a result of a dot product between the vector $v^{r,p}$ and the a-th column of the M matrix:

$$z_a^{r,p} = (v^{r,p})^T M_{:,a} = \sum_{c=1}^{d} v_c^{r,p} m_{c,a}. \quad (43)$$

Let us now substitute Eqn. (42) into Eqn. (40):

$$y_{r,p} = z^{r,p} v^{r,p} = [z_a^{r,p}]_{a=1,\ldots,d}\, v^{r,p}. \quad (44)$$

As $v^{r,p}$ is a vertical vector of d values, the above expression is a dot product of vectors $z^{r,p}$ and $v^{r,p}$

$$y_{r,p} = [z_a^{r,p}]_{a=1,\ldots,d}\, [v_a^{r,p}]_{a=1,\ldots,d} = \sum_{a=1}^{d} z_a^{r,p} v_a^{r,p}. \quad (45)$$

Substituting Eqn. (43) into Eqn. (45), we obtain

$$y_{r,p} = \sum_{a=1}^{d} \left(\sum_{c=1}^{d} v_c^{r,p} m_{c,a} \right) v_a^{r,p}. \quad (46)$$

Recall that $v_x^{r,p} = e_{x,r} - f_{x,p}$. Substituting this into (46), we obtain

$$y_{r,p} = \sum_{a=1}^{d} \left(\sum_{c=1}^{d} (e_{c,r} - f_{c,p}) m_{c,a} \right) (e_{a,r} - f_{a,p}). \quad (47)$$

As each $y_{r,p}$ value is computed independently of other $y_{r,p}$ values, we can extend the above equation to compute the whole row $Y_{r,:}$ of matrix Y. This is accomplished by replacing each occurrence of the column number p with the colon, which means "all available values". To retain the correctness of terms that are not dependent on p (such as $e_{c,r}$), we introduce the following notation. By $[x]_k$ we denote a horizontal vector of k values equal to x. All terms that are not dependent on p are converted into horizontal vectors of k values. Consequently, the row r of the tile matrix Y may be computed as follows:

$$Y_{r,:} = \sum_{a=1}^{d} \left(\sum_{c=1}^{d} ([e_{c,r}]_k - F_{c,:}) m_{c,a} \right) ([e_{a,r}]_k - F_{a,:}). \quad (48)$$

Notice that (48) expresses a single tile row in terms of either single matrix values ($e_{x,r}$ and $\bar{\sigma}_{c,a}$) or chunk rows ($F_{x,:}$). Let us rewrite the above equation in algorithmic form:

For each $r = 1, \ldots, k$, perform the following steps:

1. $Y_{r,:} \leftarrow [0]_k$.
2. For each $a = 1, \ldots, d$, perform the following steps:
   (a) $part \leftarrow [0]_k$;
   (b) for each $c = 1, \ldots, d$, perform the following step:
   $part \leftarrow part + m_{c,a} * ([e_{c,r}]_k - F_{c,:})$;
   (c) $Y_{r,:} \leftarrow Y_{r,:} + part \cdot ([e_{a,r}]_k - F_{a,:})$.
3. Output $Y_{r,:}$.

As we assume row-major order storage of the matrix A, the rows $F_{x,:}$ are stored in linear portions of the memory, which allows efficient accesses. Notice that each access to a chunk row is accompanied by an access to a single value in both the M matrix and the second chunk (E). Unfortunately, these accesses to memory are not to consecutive memory addresses. Notice, however, that there are only two such accesses per one large linear access to the chunk row. Moreover, as M and E are small, they can be easily fit within the cache memory for faster access.

A GPU implementation of this scheme is constructed as follows. Each tile is processed by a block of 256 threads ($k = 16$). Such a block size allows the best utilization of GPU multiprocessors for modern computer

capabilities. Each block caches E and F chunks in shared memory for efficient retrieval. We have limited the lengths of vectors to up to 16 values ($d \leq 16$) to simplify the copying of data to shared memory and subsequent processing (256 threads can copy 16 vectors of 16 values in parallel). The array for M is stored in row major order in constant memory. Each thread warp (32 consecutive threads) processes two rows of Y. It can be easily shown that a proper implementation avoids any global and shared memory access conflicts which would cause serialization of thread execution. Threads in blocks which process tiles on the main diagonal detect whether they are above or below the main diagonal and stop further computations where appropriate.

As starting triangular computation grids is not possible, we adapted the following solution. First, the number of tiles needed to perform all of computations is calculated. Next, a two dimensional grid containing at least the required number of blocks is started. Each block, based on its location within the grid, computes its unique number. Finally, based on this number, the block is assigned to some tile in the triangular matrix.

6. Algorithm implementations

In this section we describe how to utilize the algorithms presented in Sections 3 and 5 to create efficient GPU implementations of the PLUGIN, LSCV_h and LSCV_H algorithms.

6.1. PLUGIN. In our implementation we utilize parallel reduction algorithms we implemented to compute the variance estimator (Step 1, Section 3.2.1, Eqn. (6)) and for computing sums in Steps 4 and 6 (Section 3.2.1, Eqns. (9) and (11)). The remaining steps (2, 3, 5 and 7) are all simple equations that are inherently sequential and therefore cannot be further optimized. Nonetheless, they require a very small number of operations and can therefore be performed on a CPU in negligible time.

6.2. LSCV_h. Our implementations of the LSCV_h algorithm use the modified equations presented in Section 5.1. Consequently, the algorithm is performed using the following steps:

1. Compute the covariance matrix of X: Σ.
2. Compute the determinant of the matrix Σ: $\det(\Sigma)$.
3. Compute the inverse of the matrix Σ: Σ^{-1}.
4. Compute the approximate value of the bandwidth (see Eqn. (17)).
5. Determine the range in which we search for a minimum of the objective function $g(h)$ (see Eqn. (18)).
6. Compute $S(v^{i,j})$ for all $v^{i,j} = X_i - X_j$ such that $i = 1, \ldots, n$ and $j = i+1, \ldots, n$ (see Eqn. (27)).
7. Search for a minimum of the objective function $g(h)$ within the range computed previously (Step 5). Each time the objective is evaluated, its modified version (see Eqn. (31)) should be used, which can be computed based on precomputed values of $S(v)$.

Steps 1 to 5 of the algorithm in all implementations are performed sequentially on a CPU without using SSE instructions. Although computing the covariance matrix could be performed easily in parallel by using an algorithm similar to the one presented in Section 5.3, we did not implement that as this step takes very little time compared with the last steps of the algorithm LSCV_h presented above. The values of the $S(v)$ function are precomputed using the algorithm presented in Section 5.3 (the function $S(v)$ is an equivalent of the $fun2$ function). The last step (minimization of the function $g(h)$) is performed by a "brute force" search for a minimum on the grid, where the density of a grid is based on a user specified parameter and should be sufficiently dense. Note that, as was stated in Section 3.2.2, other approaches to minimization of $g(h)$ are also possible. In this paper, however, we present implementations of the "brute force" method.

Searching for a minimum of $g(h)$ is performed as follows. First, the user specifies n_h, the number of different h values to be tested. Based on n_h and the width of the range $Z(h_0)$ (Eqn. (18)) the h values to be tested are determined. The values of $g(h)$ for each distinct h are computed in parallel. This is done by a *gpu-kernel* started on a grid with n_h thread rows and a sufficient number of threads in each row to perform complete reduction of all of the precomputed values of $S(v)$. The gpu-kernel performs the following operations:

- Each thread based on its y-coordinate within a computation grid computes the tested argument h.
- Reduction is performed on the precomputed values of $S(v)$. For each retrieved value of $S(v)$ and based on the grid row dependent h argument value, the function $\tilde{T}(v)$ (fun function equivalent) is computed and these computed values are then added.
- Reduction is performed independently in each row of the computational grid, i.e., each row reduces the same set of $S(v)$ values but for different h argument values.

As computation grids are limited to 65535 rows, if $n_h > 65535$, the work is split into several consecutive kernel executions.

To find the final values of $g(h)$, for each value obtained during the reduction step a single thread

is started, which performs remaining operations of Eqn. (31). The computed values of $g(h)$ are copied to computer memory and an argument for which the function $g(h)$ has a minimal value is found.

6.3. LSCV_H. The LSCV_H algorithm can use any numerical function minimization algorithm capable of minimizing the function $g(H)$ (see Eqn. (20)). Such algorithms are often inherently sequential as they are based on an iterative improvement of results of the previous iteration. Consequently, the only parallel approach to such algorithms would require to start multiple parallel instances of this algorithm, each starting from a different starting point in the hope of finding a better result after a fixed number of steps. However, the number of steps needed to converge to a local optimum cannot be reduced. Possibly, for some specific algorithms, some steps could be parallelized, but this is not a subject of this paper.

Still, there is one thing that can be improved here. Notice that an iterative optimization algorithm needs to compute the objective function at least once per iteration to assess the currently found solution. Our objective function $g(H)$ can take a long time to evaluate, and while it is being computed other optimization algorithm steps cannot be processed. Consequently, the time of finding an optimal matrix H can be improved if we optimize computing of $g(H)$. As was mentioned in Section 5.3, the most expensive part of this equation is basically the same as in the function $g(h)$ (the nested sums of the function fun). Consequently, the same algorithms may be utilized to compute this function.

There are some differences, however. In LSCV_h the matrix used to evaluate $S(v)$ ($fun2$ function equivalent) was constant and the values of $S(v)$ could be precalculated. In LSCV_H, the same matrix is different each time $g(H)$ is computed. Consequently, to evaluate $g(H)$, both steps, computing exponents and reducing the value of T have to be performed each time. To make this solution a little bit more cache friendly, we combine both: the exponent finding algorithm described in Section 5.3 and the reduction algorithm presented by Harris (2007) into one kernel.

7. Experiments

7.1. Environment and implementation versions. For the purpose of experiments we implemented the PLUGIN, LSCV_h and LSCV_H algorithms (see Section 6), each in three versions: (a) sequential implementation, (b) SSE implementation and (c) GPU implementation. LSCV_H only implemented the computing of the objective function $g(H)$, as this is the only element of this algorithm that has influence on its performance. The objective function in implementation our of LSCV_h is evaluated a fixed number of times ($n_h = 150$). In all versions we used the ALGLIB library (Bochkanov and Bystritsky, 2013) to perform the matrix square root. Each implementation uses single precision arithmetic. Below we give a short description of each of the versions.

- **Sequential implementation**: A "pure C" straightforward sequential implementation of the formulas presented in Sections 3.2.1 (algorithm PLUGIN), 3.2.2 and 5.1 (algorithm LSCV_h) as well as 3.2.3 (algorithm LSCV_H). This implementation does not take into account any knowledge about the environment it is going to be executed in. It is not cache aware. The only cache awareness in this implementation is reflected in loop construction and memory storage which avoids non linear memory access (up to transposing matrices if necessary). This implementation does not use any SSE instructions and is single threaded.

- **GPU implementation**: Implementation that executes costly computations on GPU using CUDA API. This implementation is highly parallel and uses thousands of threads. Implementation tries to utilize multiple GPU memory types, including very fast shared memory, which may be treated as a user programmable cache. Implementation also uses C++ templates to automatically create several versions of each gpu-kernel with unrolled loops.

- **SSE implementation**: Implementation that tries to utilize all ways of accelerating performance available on CPUs, i.e., it utilizes SSE instructions, multiple threads (to utilize multiple CPU cores and HyperThreading capabilities) and is cache aware. OpenMP (Chapman *et al.*, 2007) is used for thread management. Our implementation also uses C++ templates to automatically create several versions of each function with unrolled loops.

Experiments were performed on a computer with an Intel Core i7 CPU working at 2.8 GHz, an NVIDIA GeForce 480GTX graphics card and 8 GB of DDR3 RAM. The CPU supports SSE4 instruction set, has 4 cores and HyperThreading capability. The GPU has 15 multiprocessors, each composed of 32 streaming processors. All implementations were run under the Linux operating system (Arch Linux distribution, 3.3.7 kernel version).

7.2. Experiments and datasets. We performed several experiments testing the influence of the number of samples (n) and their dimensionality (d) on the performance of all of the implementations introduced in Section 7.1. The input sample sizes and dimensionalities for each of the algorithms were as follows:

- algorithm PLUGIN: $n = 1024, 2048, \ldots, 32768$, $d = 1$,
- algorithm LSCV_h: $n = 64, 128, \ldots, 1024$, $d = 1, \ldots, 16$,
- algorithm LSCV_H: $n = 1024, 2048, \ldots, 16384$, $d = 1, \ldots, 16$.

All processing times measured for GPU implementations include data transfer times between the GPU and the CPU. From the obtained processing times we calculated speedups achieved for each algorithm and its implementation:

- SSE over sequential implementation,
- GPU over sequential implementation,
- GPU over SSE implementation.

The results are presented in the next section.

As the input data do not influence the processing times of the tested implementations (except maybe for the LSCV_H method, where the performance of a minimization algorithm depends on the dataset, but in our implementation we skip this part), we supply random dataset for each algorithm/sample size/dimensionality size combination. In other words, each implementation of an algorithm is tested in the same conditions.

Some of the tested instances might seem too small, especially the ones with high dimensionality. We would like to stress, however, that the purpose of these experiments was only to test the influence of the instance size on the implementation performance (as stated earlier, algorithms are not data dependent). Moreover, sizes of instances were limited by the slowest (sequential) implementation.

7.3. Experimental results. Let take a look at Fig. 2(a). It presents speedups of our GPU implementation of PLUGIN over the sequential implementation. As can be noticed, the GPU implementation is about 500 times faster than the sequential equivalent. Moreover, notice that the bigger the instance, the greater speedup is obtained. Figure 2(b) presents speedups of our SSE implementation of PLUGIN over the sequential equivalent. The obtained speedups are about 32. Notice that similarly as with the GPU implementation, the obtained speedup grows as the size of the instance increases. Figure 2(c) presents speedups of our SSE implementation of PLUGIN over the sequential implementation. The maximum obtained speedup is about 16 and the speedup grows as the size of the instance increases.

Let us now take a look at Fig. 3. It presents speedups of GPU and SSE implementation of LSCV_h over the sequential equivalent (Figs. 3(a) and 3(b), respectively)

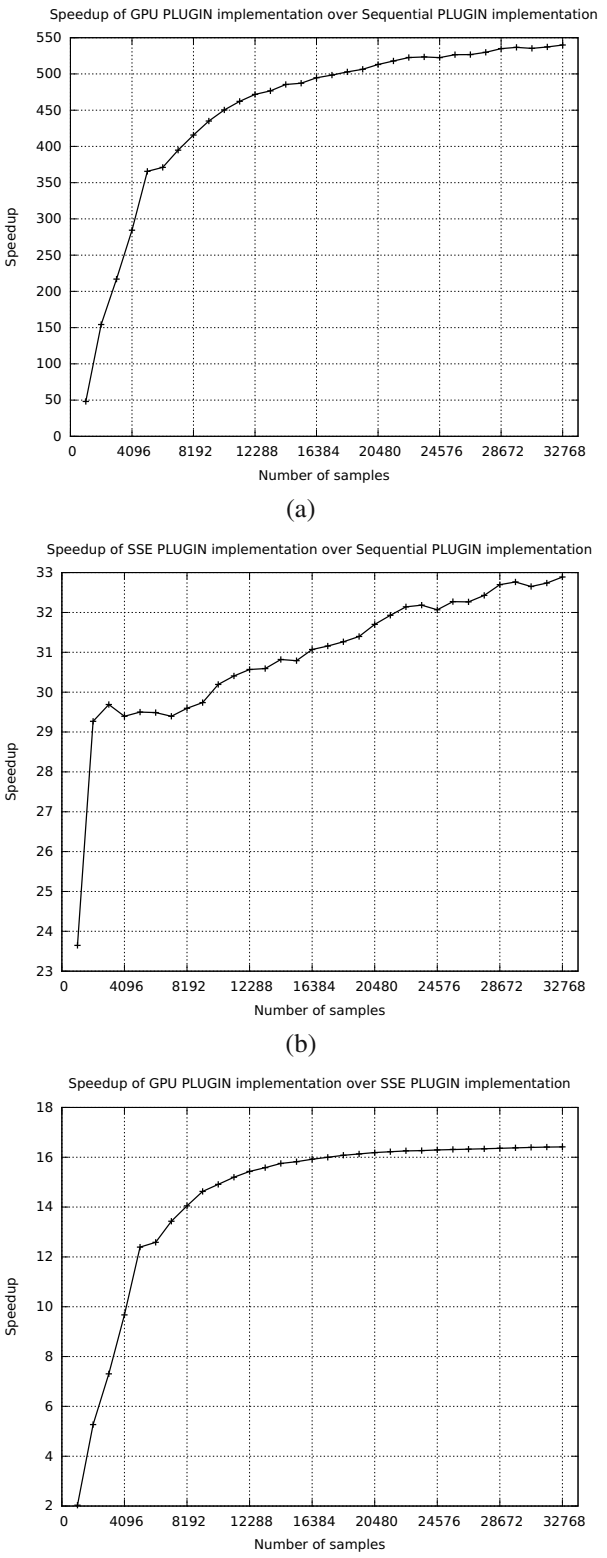


Fig. 2. Total speedups of all implementations of PLUGIN.

and the speedup of the GPU implementation over the SSE implementation (Fig. 3(c)). Each curve represents a different data dimensionality. The speedups achieved here are as follows:

- The GPU implementation is about 550 times faster than the sequential implementation.
- The SSE implementation is about 20 times faster than the sequential implementation.
- The GPU implementation is about 20 times faster than the SSE implementation.

Notice that, in all cases, the speedup increases with respect to the number of samples. This is due to the following observations. The algorithm's complexity is $O(n^2(d^2 + n_h))$ (see Section 5.1). Given the fact that d and n_h are constant, the complexity is reduced to $O(n^2)$. This means that the processing time of any implementation is given by a second-order polynomial. As the speedup here is determined by a ratio of slower to faster implementations, its value could be determined by a ratio of two polynomials. As $n \to 0$, this ratio tends to be equal to a ratio of constant terms whereas for $n \to \infty$ the same ratio tends to be equal to the ratio of coefficients of the highest order term (here, the second order).

Another interesting observation is that for each dimensionality the speedup limit is slightly different. This is due to the compiler optimization in which loops are unrolled, and for each dimensionality the compiler creates a different version of used procedures. Consequently, each curve represents in fact a slightly different program. Moreover, the bigger the dimensionality, the higher the speedup but the difference between consecutive speedups is diminishing. Let us recall that the algorithm complexity LSCV_h is $O(n^2(d^2+n_h))$. For specified constant values of n and n_h (arbitrarily set as 150), the complexity is reduced to $O(d^2)$. Consequently, the processing times may be determined by a second-order polynomial where the dimensionality is an independent variable. As the speedup is calculated by dividing the processing times of the slower implementation by the times achieved by the faster implementation, it is basically a proportion of two second order polynomials, which has a limit as $d \to \infty$.

Finally, let us analyze Fig. 4, which presents speedups of our GPU implementation and SSE implementations of LSCV_H over the sequential implementation (Figs. 4(a) and 4(b), respectively) and the speedup of the GPU implementation over the SSE implementation (Fig. 4(c)). Similarly as in the algorithm LSCV_h, we tested processing times for data dimensionalities equal to $d = 1, \ldots, 16$ and computed speedups based on the obtained values. The speedups achieved here are as follows:

- The GPU implementation is 290 times faster than the sequential implementation.

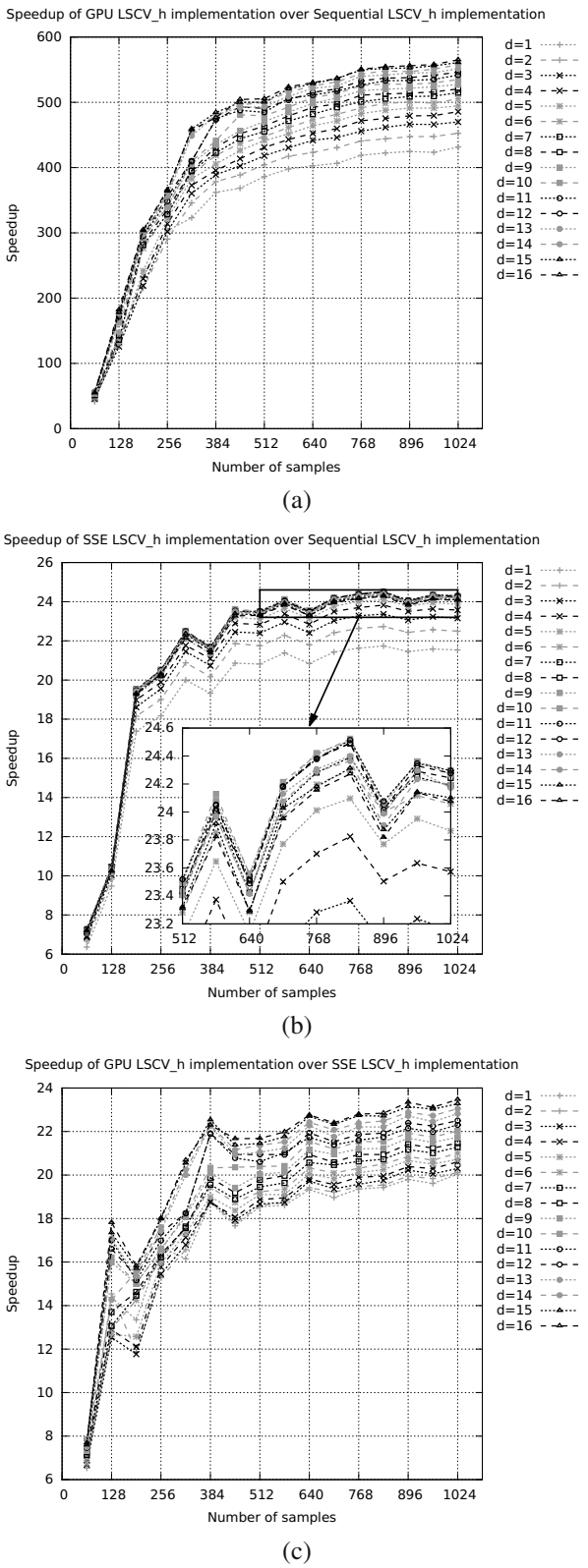


Fig. 3. Total speedups of all implementations of LSCV_h.

- The SSE implementation is 20 times faster than the sequential implementation.
- The GPU implementation is 10 times faster than the SSE implementation.

Most of the discussion for the algorithm LSCV_h presented earlier can also be used to explain the results obtained for implementations of LSCV_H. Recall that our implementation LSCV_H is not complete. We only tested the processing time needed to evaluate $g(H)$ (Eqn. (20)). Let us determine the complexity of evaluating $g(H)$. The function $g(H)$ contains double sums which add $O(n^2)$ $T_H(X_i - X_j)$ function values. Evaluation of $T_H(X_i - X_j)$ requires $O(d^2)$ operations. Consequently, the computation order of the function $g(H)$ is $O(n^2 d^2)$. This, in turn, leads to the earlier discussion of representing function processing times with second order polynomials with either n or d as an independent variable, and the second variable constant.

Several other observations can be made. First, take a look at Fig. 4(a). Curves for dimensionalities $d = 1, \ldots, 3$ are different from the rest of the observed ones. Notice that here the speedup seems to drop with an increase in the number of samples (n). These observations, however, though look differently than previously analysed on other figures, also fit our earlier discussion regarding limits of the speedup function defined as a ratio between two second-order polynomials. Such an observation means that probably the constant part of the polynomial is smaller for the GPU implementation than for the sequential implementation. However, in such a case this would mean that the GPU implementation requires a smaller initialization time than the sequential version. This is certainly not true, as the GPU implementation needs to perform several additional tasks, such as data transfer to device memory. This phenomenon may be explained as follows. Even though the number of dimensions is low, the outer loops of the sequential implementation are dependent on n^2 (they iterate over every combination of two samples from the dataset). This means that there is a constant (we assume $n = 16384$) processing time required for processing those loops (branch prediction, etc.). Low values of d mean that the inner loops, which evaluate Eqn. (21), require less time and therefore constitute a lesser percentage of the whole algorithm processing time. The same situation does not influence the GPU implementation much, as outer loop iterations are performed in parallel. Consequently, less time is wasted on n^2 dependent loop processing costs. This, in turn, leads to the conclusion that for high n and low d the speedup is higher. Now, let us assume that $n = 1024$. Low values of n mean low outer loop processing costs and GPU typical initialization costs start to dominate, which leads to smaller speedups for low values of d. This can be also noticed in Fig. 4(a). One could ask why the same phenomenon was not observed for the LSCV_h algorithm. This stems from the fact that in that algorithm the values of Eqn. (30) (counterpart of Eqn. (21)) are computed only once, and other algorithm tasks dominate.

Table 1. Processing times for largest instances in the experiments.

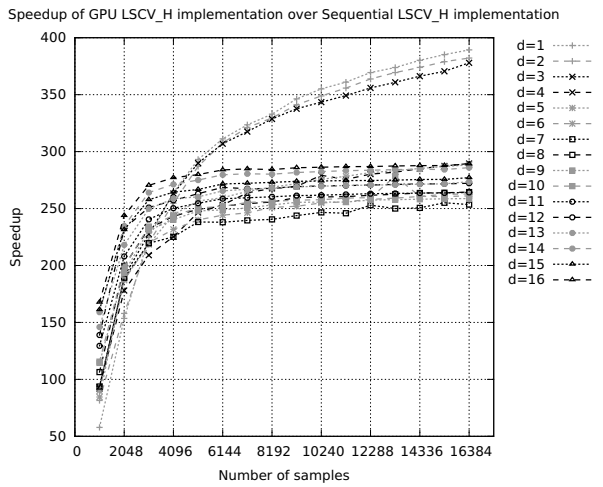
Algorithm (instance size)	Implementation [ms]		
	GPU	SSE	Seq.
PLUGIN ($n = 32768$)	87.9	1442.3	47435.3
LSCV_h ($n = 1024$, $d = 16$)	14.7	344.1	8283.6
LSCV_H ($n = 16384$, $d = 16$)	184.2	2320	53258.8

Finally, in Table 1, for illustrative purposes only, we present a comparison of processing times obtained for largest instances in each of the experiments.

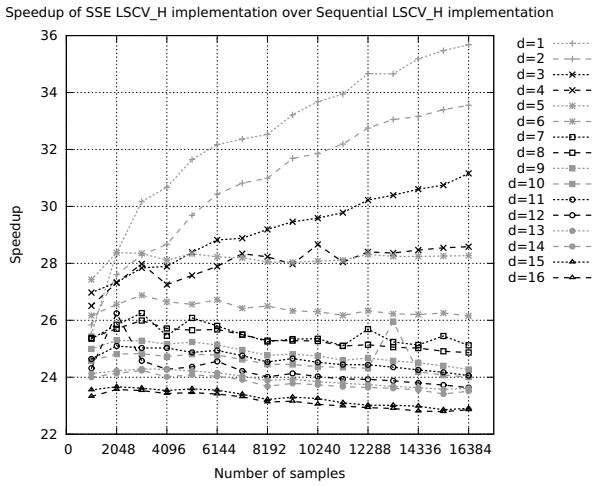
8. Conclusions and future work

In the paper we have presented some methods of how to efficiently compute the so-called bandwidth parameter used in computing Kernel Probability Density Functions (KPDFs). The functions can be potentially used in various database and data exploration tasks. One possible application is the task known as approximate query processing. However, a serious drawback of the KPDF approach is that computations of the optimal bandwidth parameter (crucial in the KPDF) are very time consuming. To solve this problem, we investigated several methods of optimizing these computations. We utilized two SIMD architectures, the *SSE CPU* architecture and the *NVIDIA GPU* architecture, to accelerate computations needed to find the optimal value of the bandwidth parameter. We tested our SSE and GPU implementations using three classical algorithms for finding the optimal bandwidth parameter: PLUGIN, LSCV_h and LSCV_H. Detailed mathematical formulas are presented in Section 3.2. As for the algorithm LSCV_h, we proposed some slight but important changes in the basic mathematical formulas.

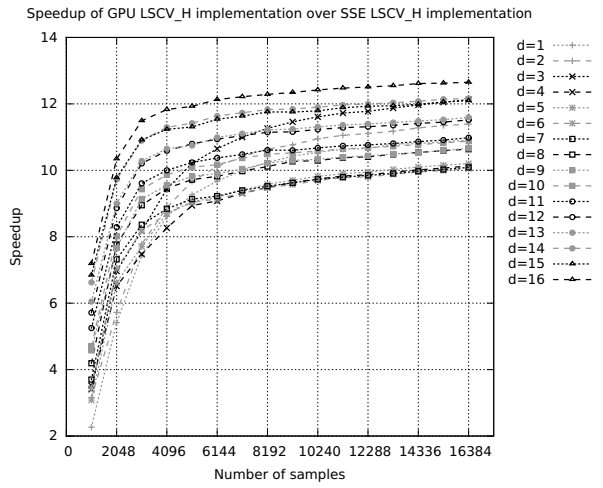
The changes allow us to precompute some values which may be later reused many times. The details can be found in Section 5.1. The fast SSE and CUDA implementations are also compared with a simple sequential one. All the necessary details on using the GPU architecture for fast computation of the bandwidth parameter are presented in Section 5, and the final notes on how to utilize the algorithms are given in Section 6. Our GPU implementations were about 300–500 times faster than their sequential counterparts and 12–23 times faster than their SSE counterparts. SSE implementations were about 20–30 times faster than sequential implementations.



(a)



(b)



(c)

Fig. 4. Total speedups of all LSCV_H algorithm implementations.

The above results confirm the great usability of modern processing units. All the codes developed have been made publicly available and can be downloaded from `www.cs.put.poznan.pl/wandrzejewski/_sources/hcode.zip`.

In the future, we plan to extend our research in many areas. With respect to GPU utilization, we plan on (i) utilizing new capabilities offered by the Kepler architecture (such as dynamic parallelism), (ii) extending our solutions to use multiple graphics cards simultaneously, (iii) utilizing the high performance of double precision computations on the Fermi and Kepler architectures and testing the observed trade-off between accuracy and performance, and finally (iv) testing the performance of our solutions on many different graphics cards models (with different numbers of CUDA cores and different compute capabilities). With respect to the versatility of our solutions, we plan on (i) extending our algorithms to support different kernel types (currently we support only Gaussian kernels) and (ii) developing incremental methods which would allow updating the bandwidth instead of recalculating it every time when a dataset is changed (sometimes very insignificantly). This would allow us to use our solutions even in real transactional database systems.

Acknowledgment

This research was funded by the Polish National Science Center (NCN), grant no. 2011/01/B/ST6/05169.

References

Andrzejewski, W., Gramacki, A. and Gramacki, J. (2013). Density estimations for approximate query processing on SIMD architectures, *Technical Report RA 03/13*, Poznań University of Technology, Poznań.

Blohsfeld, B., Korus, D. and Seeger, B. (1999). A comparison of selectivity estimators for range queries on metric attributes, *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, Philadelphia, PA, USA*, pp. 239–250.

Bochkanov, S. and Bystritsky, V. (2013). ALGLIB, `http://www.alglib.net`.

Chapman, B., Jost, G. and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*, MIT Press, Cambridge, MA.

Duong, T. (2004). *Bandwidth Selectors for Multivariate Kernel Density Estimation*, Ph.D. thesis, University of Western Australia, Perth.

Farooqui, N., Kerr, A., Diamos, G., Yalamanchili, S. and Schwan, K. (2011). A framework for dynamically instrumenting GPU compute applications within GPU Ocelot, *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4, Newport Beach, CA, USA*, pp. 9:1–9:9.

Gramacki, A., Gramacki, J. and Andrzejewski, W. (2010). Probability density functions for calculating approximate aggregates, *Foundations of Computing and Decision Sciences* **35**(4): 223–240.

Greengard, L. and Strain, J. (1991). The fast Gauss transform, *SIAM Journal on Scientific and Statistical Computing* **12**(1): 79–94.

Harris, M. (2007). Optimizing parallel reduction in CUDA, `http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`.

Hendriks, H. and Kim, P. (2003). Consistent and efficient density estimation, *in* V. Kumar, M.L. Gavrilova, C.J.K. Tan and P. L'Ecuyer (Eds.), *Proceedings of the 2003 International Conference on Computational Science and Its Applications, ICCSA 2003: Part I*, Lecture Notes in Computer Science, Vol. 2667, Springer-Verlag, New York, NY, Berlin/Heidelberg, pp. 388–397.

Johnson, N., Kotz, S. and Balakrishnan, N. (1994). *Continuous Univariate Distributions, Volume 1*, Probability and Statistics, John Wiley & Sons, Inc, New York, NY.

Johnson, N., Kotz, S. and Balakrishnan, N. (1995). *Continuous Univariate Distributions, Volume 2*, Probability and Statistics, John Wiley & Sons, Inc, New York, NY.

Kulczycki, P. (2005). *Kernel Estimators in Systems Analysis*, Wydawnictwa Naukowo-Techniczne, Warsaw, (in Polish).

Kulczycki, P. (2008). Kernel estimators in industrial applications, *in* B. Prasad (Ed.), *Studies in Fuzziness and Soft Computing. Soft Computing Applications in Industry*, Springer-Verlag, Berlin, pp. 69–91.

Kulczycki, P. and Charytanowicz, M. (2010). A complete gradient clustering algorithm formed with kernel estimators, *International Journal of Applied Mathematics and Computer Science* **20**(1): 123–134, DOI: 10.2478/v10006-010-0009-3.

Li, Q. and Racine, J. (2007). *Nonparametric Econometrics: Theory and Practice*, Princeton University Press, Princeton, NJ.

Łukasik, S. (2007). Parallel computing of kernel density estimates with MPI, *in* Y. Shi, G.D. van Albada, J. Dongarra and P.M.A. Sloot (Eds.), *Computational Science—ICCS 2007*, Lecture Notes in Computer Science, Vol. 4489, Springer, Berlin/Heidelberg, pp. 726–734.

NVIDIA Corporation (2012). NVIDIA CUDA programming guide, `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`.

NVIDIA Corporation (2013). NVIDIA's next generation CUDA compute architecture: Kepler GK110, `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

Michailidis, P.D. and Margaritis, K.G. (2013). Accelerating kernel density estimation on the GPU using the CUDA framework, *Applied Mathematical Sciences* **7**(30): 1447–1476.

Nelder, J.A. and Mead, R. (1965). A simplex method for function minimization, *The Computer Journal* **7**(4): 308–313.

Raykar, V. and Duraiswami, R. (2006). Very fast optimal bandwidth selection for univariate kernel density estimation, *Technical Report CS-TR-4774/UMIACS-TR-2005-73*, University of Maryland, College Park, MD.

Raykar, V., Duraiswami, R. and Zhao, L. (2010). Fast computation of kernel estimators, *Journal of Computational and Graphical Statistics* **19**(1): 205–220.

Sawerwain, M. (2012). GPU-based parallel algorithms for transformations of quantum states expressed as vectors and density matrices, *in* R. Wyrzykowski, J. Dongarra, K. Karczewski and J. Waśniewski (Eds.), *Parallel Processing and Applied Mathematics*, Lecture Notes in Computer Science, Vol. 7203, Springer-Verlag, New York, NY/Berlin/Heidelberg, pp. 215–224.

Sheather, S. (2004). Density estimation, *Statistical Science* **19**(4): 588–597.

Silverman, B. (1986). *Density Estimation for Statistics and Data Analysis*, Chapman & Hall/CRC Monographs on Statistics & Applied Probability, London.

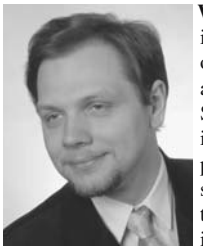
Silverman, B.W. (1982). Algorithm AS 176: Kernel density estimation using the fast Fourier transform, *Journal of the Royal Statistical Society: Series C (Applied Statistics)* **31**(1): 93–99.

Simonoff, J. (1996). *Smoothing Methods in Statistics*, Springer Series in Statistics, Springer-Verlag, New York, NY/Berlin/Heidelberg.

Wand, M. and Jones, M. (1995). *Kernel Smoothing*, Chapman & Hall/CRC Monographs on Statistics & Applied Probability, Chapman&Hall, London.

Xavier, C. and Iyengar, S. (1998). *Introduction to Parallel Algorithms*, Wiley Series on Parallel and Distributed Computing, Wiley.

Yang, C., Duraiswami, R. and Gumerov, N. (2003). Improved fast Gauss transform, *Technical Report CS-TR-4495*, University of Maryland, College Park, MD.



Witold Andrzejewski received the Ph.D. degree in computer science from the Poznań University of Technology in 2008. Since 2008 he has been an assistant professor at the Institute of Computer Science at that university. His research interests include optimization of query processing of complex data structures and data mining via auxiliary structures (indices), as well as hardware acceleration and parallelization (GPU based) of some typical database and data mining operations.

Artur Gramacki received the Ph.D. degree from the Technical University of Zielona Góra, Poland, in 2000. Since then he has been working as an assistant professor at the Institute of Computer Engineering and Electronics of the same university. His research interests focus on database systems, data modeling and exploratory data analysis, including statistical methods.

Jarosław Gramacki received his Ph.D. degree in electrical engineering from the Technical University of Zielona Góra, Poland, in 2000. Up to 2012 he was an assistant professor at the Faculty of Electrical Engineering, Computer Science and Telecommunications at the same university. Currently, he manages the main database systems in the Computer Center of the University of Zielona Góra. His research interests include database systems, data modeling, exploratory data analysis, description of dependencies in data and applications of statistics.

Received: 20 March 2013
Revised: 26 June 2013