Density Estimations for Approximate Query Processing on SIMD Architectures

Witold Andrzejewski*, Artur Gramacki** and Jarosław Gramacki***

*Poznań University of Technology, Institute of Computer Science
**University of Zielona Góra, Institute of Computer Science and Electronics
***University of Zielona Góra, Computer Center
`Witold.Andrzejewski@cs.put.poznan.pl`
`a.gramacki@iie.uz.zgora.pl`
`j.gramacki@ck.uz.zgora.pl`

Abstract. Approximate query processing (AQP) is an interesting alternative for exact query processing. It is a tool for dealing with the huge data volumes where response time is more important than perfect accuracy (this is typically the case during initial phase of data exploration). There are many techniques for AQP, one of them is based on probability density functions (PDF). PDFs are typically calculated 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. The shape of final density function is very sensitive to an entity called bandwidth or smoothing parameter. Calculating it's optimal value is not a trivial task and in general is very time consuming. In this paper we investigate the possibility of utilizing two SIMD architectures: SSE CPU extensions and NVIDIA's CUDA architecture to accelerate finding of the bandwidth. Our experiments show orders of magnitude improvements over a simple sequential implementation of classical algorithms used for that task.

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

1 Introduction

The paper is about implementing an *approximate query processing* (AQP) technique with the support of two SIMD architectures: SSE extensions of modern CPUs and Graphics Processing Units (GPUs). We propose modifications of classical algorithms which perform parallel and concurrent computations to accelerate very time-consuming operations while calculating the optimal *kernel density estimators* (KDE), which heavily depends on the so called *bandwidth* or *smoothing parameter*. These estimators estimate probability density functions (PDF) which can be used in the AQP task.

The perfect situation is if every query that is sent to the database engine could return an exact solution in no more than seconds. However, if the database stores really huge amount of data (as it is the most typical in data warehouses, DW) such a perfect behavior is not a general rule. A lot of theoretical as well as practical works have been done so far in the area of DW optimization [23]. The research is usually concentrated on such aspects as designing a dedicated logical and physical data structures (novel schemes for indexing, materialized views, partitioning, etc.). Relatively less attention is paid for getting approximate results. Is seems obvious that if we have the following dilemma "what is better: getting the exact solution in 1 hour or getting only approximate solution in less then seconds", we probably lean toward the second possibility. Getting only approximate solutions, instead of exact ones, in many practical situations is absolutely acceptable. For example, if our goal is to calculate the total gross income within the whole fiscal year, the roundoff to full thousands is obviously correct approach. There are at least a few different schemes for implementing AQP. We supply a brief review of them in section 2.1.

In the paper we concentrate on a technique based on analyzing *statistical properties* of data. We use probability density functions which give an elegant and efficient framework for implementing AQP. However, one of the most serious drawback of this approach is that typical kernel nonparametric algorithms used in evaluating optimal PDFs scale quadratically (see section 4.4 for precise mathematical formulas). If data sizes increase, such kernel methods scale poorly. To overcome this problem, two main approaches may be used. In the first one, many authors propose various methods of evaluating *approximate kernel density estimates*. We give a short summary of these methods in section 2.2. In this paper we investigate the second approach where kernel density estimates are evaluating *exactly*. To make this approach practically usable, all the time consuming computations are performed using two modern SIMD architectures. We describe two implementations of each of three different bandwidth finding algorithms. The first implementation (later called *SSE implementation*) utilizes two benefits of modern CPUs: SSE extensions allowing to perform the same instructions on multiple values in parallel as well as multicore architecture allowing to process several threads in parallel on several distinct cores. The second implementation (later called *GPU implementation*) utilizes NVIDIA's CUDA architecture to start multiple threads and perform as many computations concurently or in parallel as possible. The speedups we obtain are in the range of one to two orders of magnitude in comparison with classical sequential CPU-based implementation (later called *Sequential implementation*).

The remainder of the paper is organized as follows. In section 2 we cover the necessary background material on APQ, computation of PDFs and bandwidth selection. In Section 3 we supply a brief review of modern SIMD architectures. In Section 4 we turn our attention to give the reader some preliminary informations on PDFs, kernel estimators of PDFs, using PDFs in database area. We also give the very detail mathematical formulas for calculating optimal bandwidth parameters using three different methods. We also propose some modifications of the basic formulas to improve calculations performance. In section 5 we cover all the necessary details on parallel algorithms for bandwidth selection. In section 6 we show how to utilize the algorithms presented in section 5. In section 7 we give details on hardware and software used in experiments performed and present the results of the experiments (presented as speedups over different implementations, that is SSE, GPU and sequential ones). In section 8 we conclude our paper. In appendix A we give the details on how we derived some important formulas later used in our algorithms.

2 Related works

2.1 Approximate query processing

Approximate query processing can be done in many different schemes [11]. They all assume applying a kind of preliminary data reduction which gives as a result a synopsis of the original input data. All the following data queries operate on this synopsis instead of the original data. Probably the simplest method of obtaining synopses is sampling. In this case we believe that data after sampling remain still sufficiently representative. This method is often called *numerosity reduction* [14]. There is also a complementary to numerosity reduction method called *dimensionality reduction*, but this technique is rather not used in the AQP area [7,10]. Note also that many commercial RDBMS use sampling in the process of determining the best query execution plans. The synopsis can be built using such techniques as: histograms [18], wavelets [34] or investigating statistical properties of the data [29]. The latter approach is investigated in the paper.

2.2 Computation of probability density functions

The probability density function (also called *probability distribution*, *density function* or simply *density*) is one of the most important and fundamental concept in statistics and it is widely used in exploratory data analysis. Fast calculation of the optimal PDF is still an interesting scientific problem. There exist a lot of parametric forms of density functions, that is if their shapes can be described by a mathematical formula. A very complete list of probability density functions can be found for example in [20,21]. On the other hand, if the parametric form of the PDF is unknown (or difficult to calculate) one should consider usage of nonparametric methods.

The task of the PDF estimation is to compute an estimate $\hat{f}$ based on the given n sample points drawn from an unknown population of data with density f. One of the most popular nonparametric method is the kernel density estimator, see for example [35] for a complete review.

There are two main computational problems related to KDE. Firstly, the calculation of the estimate $\hat{f}(x, h)$ (or $\hat{f}(x, H)$, see chapter 4.2 for shortened explanation on differences beetwen h and H) and secondly, estimation of the optimal (in some sense) bandwidth parameter h (or H). 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 KDE is to use a technique known as *binning* or *discretising*. In such a case, the density estimation is evaluated at grid points rather than source points. The idea relies on generating grid points (not necessarily equally spaced) of the size g, where $g \ll n$ and $g_1 < \cdots < g_m$. Then the original $x_1, \cdots x_n$ source points are replaced by *grid counts* $c_1, \cdots c_g$, where the value of c_i depends on the "mass of the data" near g_i. Binning strategy reduces the required kernel evaluations to $O(mg)$. Furthermore, if the evaluation points are the same as grid points, the further kernel evaluation from $O(g^2)$ to $O(g)$ is possible (as certain evaluations use the same arguments and don't need to be calculated again). Another approach toward saving computational complexity is based on using Fast Fourier Transformation (FFT), first proposed in [32]. Using FFT requires that the source points are on an evenly spaced grid and then one can evaluate KDE at an evenly spaced grid in $O(nlog_n)$. If the source points are irregularly spaced the pre-binning strategy described above 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 bening. This problem is investigated in details in [16]. Another technique which reduces the computational complexity is based on Fast Gauss Transform (FGT) introduced in [13] and can be viewed as an extension of the Improved Fast Gauss Transform (IFGT) [37]. The method is called by the authors $\epsilon - exact$ [28] in the sense that the constant hidden in $O(m + n)$ depends on the desired accuracy which can be chosen arbitrary.

As for the problem of accelerating computational times for finding the optimal bandwidth h relatively less attention is payed in literature. An attempt at using Message Passing Interface (MPI) was presented in [25]. In [31] the author gives an FFT-based algorithm for accelerating a method (least-square cross validation one) for finding the optimal bandwidth h.

2.3 Bandwidth selection for kernel density estimates

Bandwidth selection problem is probably the most important one in the KDE area. Fast and correct bandwidth selection is the clue to practical usability of kernel-based density estimates of PDFs. Currently available selectors can be roughly divided into 3 classes [35,31,30]. The first class contains very simple and easy to calculate mathematical formulas. They were developed to cover a wide rage of situations, but do not guarantee being close to the optimal bandwidth. They are however willingly and often used as a starting point in more accurate bandwidth selection process. These methods are sometimes called *rules-of-thumb*. The second class contains methods based on *least square* and *cross-validation* ideas and more precise mathematical arguments. But unfortunately they require much more computational effort. However, in reward for it, we get bandwidths more accurate for a wider range of density functions. The method will be abbreviated as *LSCV*. The third class contains methods based on pluging in estimates of some unknown quantities that appear in formulas for the asymptotically optimal bandwidth. The methods are called *plug-in* ones and hereafter will be denoted as *PLUGIN*. These methods are also computationally difficult because there is a need for computation of some functionals and the direct algorithm involves $O(n^2)$ operations. The computational burden can be reduced by using binning strategy as briefly described in section 2.2.

3 Single Instruction Multiple Data architectures

Single Instruction Multiple Data (SIMD) processing systems process multiple streams of data based on a single instruction stream thereby exploiting the data level parallelism. First instroduced as a feature of vector supercomputers such as CDC Star-100 and Thinking Machines CM-1 and CM-2 SIMD processing was later implemented in INTEL's commodity processors. A similar approach, though a little more flexible was implemented in modern GPUs.

3.1 Parallel data processing on commodity CPUs

Starting with Pentium MMX, commodity CPUs have started supporting Single Instruction Multiple Data processing. This was later extended by both Intel[3] and AMD [2,1] in subsequent extensions called 3DNow!, SSE, SSE2, SSE3, SSSE3, SSE4, XOP, FMA4, CVT16 (former SSE5) and AVX. CPUs supporting these technologies contain additional registers capable of storing multiple values. These are essentially vectors, which may be processed by specialized instructions as a whole. For example two 4 value vectors stored in two registers may be added by using a single CPU instruction.

SSE1 (Streaming Simd Extensions) introduced 128bit registers capable of storing four single precision floating point values as well as a set of 70 CPU instructions for processing them. SSE2 added the possibility to store in registers two double precision values instead of four single precision and added additional 144 instructions. SSE3 introduces 13 new instructions, including the ones with capability to perform operations on values stored within the same register. SSE4 added 54 instructions which were (among others) useful for operations performed in HD codecs and for string processing. SSE5 instruction set extension was proposed by AMD on 30 August 2007 as a supplement to the 128-bit SSE core instructions in the AMD64 architecture. In May 2009, AMD replaced SSE5 with three smaller instruction set extensions named as XOP, FMA4, and CVT16, which retain the proposed functionality of SSE5, but encode the instructions differently for better compatibility with Intel's proposed AVX instruction set. AVX extension increases the length of SIMD registers to 256 bits and adds three operand instructions where the destination register is distinct from the source registers.

SIMD is of course not the only level of parallel data processing on modern CPUs. Other solutions used in conujnction with SIMD are: superscalar architecture (multiple execution units and multiple pipelines) as well as multiple cores. Our implementations, aside from SIMD also utilize modern CPUs capability to run multiple threads in parallel on multiple cores.

In our implementations we primarily use `__m128` type variables which correspond to 128bit registers holding 4 single precision values.

3.2 General processing on graphics processing units - GPGPU

Rapid development of graphics cards driven by the video game market as well as many professional applications (such as movie special effects) has led to creating devices far more powerful than standard CPUs. Graphics cards are of course more specialized than general-purpose CPUs. Typical processing of graphics algorithms involves performing of the same algorithm steps for many different input values (such as applying geometrical transformations to vertices or computing pixel colors) and is akin to SIMD concept. However, if any algorithm (not necessarily related to computer graphics) may be mapped to such an approach to data processing, it may be efficiently executed on a graphics card. In the beginning, programs utilizing GPUs for general purpose processing used a graphics API such as OpenGL+GLSL or Direct3D+HLSL. This caused some restrictions on the implemented algorithms (lack of the scatter operation) as well as required from the programmer to create some mapping between the algorithm operations and graphics operations. These problems vanished when NVIDIA CUDA and OpenCL were developed. Both of these APIs allow the programmer to completely omit the graphics API and use the standard programming language constructs to create programs for GPUs. In our paper we use the NVIDIA CUDA API, which is closely related to the NVIDIAs graphics cards architecture and therefore allows for some low level optimization.

Let us now roughly describe the NVIDIA GPU architecture and its relation to the CUDA API[4]. 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. Groups of SPs on a single SM share a the so-called *warp scheduler* which issues successive instruction or each of the SPs in the group. Consequently, each SP in a group performs the same instruction at the same time (SIMD). Current (2012) graphics cards contain 30 SM with 8 SP and 1 warp scheduler each (NVIDIA GeForce 285GTX), or 16 SM with 32 SP and 2 warp schedulers each (NVIDIA GeForce 580GTX). This shows, that the GPUs are capable of running several hundred threads in parallel (and even more concurrently). Each NVIDIAs graphics card has assigned a compute capability (denoted 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 *the 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 from kernels used in kernel-based density estimates, we will call these functions the *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 SMs 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 the consequence of having one warp scheduler per several SPs). Consequently all of these threads should perform the same intruction. If the threads with a warp perform different code branches, all branches are serialized, and threads not performing the branch are masked. Perfomance of an implementation is therefore determined by:

1. the number of global memory accesses (the smaller, the better, use shared memory in favor of global memory).
2. the number of global synchronization points (the smaller, the better, use in-block synchronization in favor of global synchronization).
3. the parallelism degree (the bigger, the better).
4. each group of 32 consecutive threads should follow the same code branches.

There are also several other efficiency guidelines, which do not stem from the above description, but are related to some lower level details of graphics cards hardware:

1. the smaller, the number of conditional code executions the better.
2. each thread in a group of 16 (or 32 for graphics cards with CC$\geq$2.0) consecutive threads should follow a conflict-free shared memory access pattern (see [4]).
3. each thread in a group of 16 (or 32 for graphics cards with CC$\geq$2.0) consecutive threads should use global memory access patterns which allow for coalesced accesses (see [4]).
4. use single precision floating point computations, if possible.

All of our implementations comply with the above guidelines. Regarding the last requirement, our implementations use single precision, but in the near future we can expect, that double precision will be efficient on GPUs as well. Our solutions can be then easily ported to support double precision. It should also be noted, that the efficiency of double precision has already increased dramatically between two recent generations of graphics cards (compare for example NVidia GeForce 285GTX and NVidia GeForce 580GTX).

4 Mathematical preliminaries

In this section we give some preliminary informations on some basic statistical concepts (probability density function and kernel density estimation) as well as how to use them in the database area. Most of this section is devoted to give the precise recipes for calculation of the so called optimal bandwidth which plays the most important role while estimating kernel-based probability density functions. We also propose some slight but important modifications of the reference mathematical formulas for calculating the bandwidth. The modifications play very important role during GPU-based and SSE-based fast algorithm implementations for calculating of the bandwidth.

4.1 Probability density function

Let X be a random variable and let f be the aforementioned probability density function. The probability P that a random variable X takes a value in the given interval $[a, b]$ is defined as follows:

$$P(a \leq X \leq b) = \int_a^b f(x)dx. \tag{1}$$

Density function must fulfill two basic properties: must be not negative over the whole domain, that is $f(x) \geq 0$ for every x and the value of the integral (1) must be exactly 1, that is $P(-\infty \leq X \leq +\infty) = \int_{-\infty}^{+\infty} f(x)dx = 1$. In the context of the main subject of the paper, we also need to know a formula for calculating mean value of the random variable. This is defined as follows:

$$\mu_X = \int_{-\infty}^{+\infty} xf(x)dx. \tag{2}$$

The above formulas can be easily generalized into two or more dimensions. In practice, three different cases can be considered: (a) if analytical formula of the PDF is known and the integrals (1) and (2) can be solved analytically. In such a case the solution we get is the exact one. (b) if we know the analytical formula of the PDF, but solving the aforementioned integrals is very difficult or even not possible. In such a case one must use methods for numerical integration (a broad family of algorithms exist, for example rectangle rule, trapezoidal rule, Newton-Cotes formula, Gaussian quadrature, Monte Carlo methods and other). (c) if we don't know at all analytical formula of the PDF, then the nonparametric methods for its estimation should be considered.

4.2 Kernel density estimation

Let us assume that we have a set of source points to be a sample from an unknown density function. *Density estimate* is simply the process of construction of an estimate of the density function (1) from the observed data [31]. There are two main approaches to density estimation: *parametric* and *nonparametric* ones. The first assumes that the data are drawn from a known family of distributions. For example the normal distribution is fully described by only two parameters (mean value μ and standard deviation σ). All we need to do is to find estimates of these parameters from the observed data. In the paper this approach is not considered. In the second approach, we assume no preliminary knowledge about a form of a density function. It should be discovered from the set of source points. This approach is sometimes vividly described as *Let the Data Speak for Themselves.*

A trivial example of a nonparametric PDF is *histogram*. However, its practical usability is rather poor (in the context of using them as PDFs estimators), because in general, and with its basic form, it is difficult to give an optimal number of the bins, their width and the starting point of the first bin. The shape of histogram is very sensitive to these three parameters. In that case, it is much more convenient to use the so called *kernel density estimators* (KDE) which are much more adequate for building nonparametric PDFs. Three classical books on KDEs are [31,33,35]. There are, however, known families of so called optimal histograms (e.g. v-optimal histograms [19]), but we don't consider them in the paper. We only mention that their construction algorithms are very time and memory consuming. It should be also noted that basic histograms are commonly used in database management systems (DBMSs). This is a feature in cost based optimizers and help the optimizer to decide which SQL execution plan should be used to get the best query execution.

Now let us consider a random variable X (in general d-dimensional) and let assume its probability density function f 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 $X_1, X_2, ..., X_n$ (our experimental data). In such a case, the d-dimensional kernel density estimator $\hat{f}(x,h)$ of the real density $f(x)$ for random sample $X_1, X_2, \ldots, X_n$ is given by the following formula:

$$\hat{f}(x,h) = n^{-1} \sum_{i=1}^{n} K_h\left(x - X_i\right) = n^{-1} h^{-d} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right) \tag{3}$$

where

$$K_h(u) = h^{-1} K(h^{-1} u) \tag{4}$$

and

$$K(u) = (2\pi)^{-d/2} exp\left(-\frac{1}{2} u^T u\right) \tag{5}$$

where n – number of samples, d – task dimensionality, $x = (x_1, x_2, \ldots, x_d)^T$, $X_i = (X_{1i}, X_{2i}, \ldots, X_{di})^T$, $i = 1, 2, \ldots, n$. X_{1i}, X_{2i}, …, X_{di} denote the consecutive elements of d-dimensional vector X. h is a positive real number called *smoothing parameter* or *bandwidth.* $K(\cdot)$ is the *kernel function* – a symetric but not necessarily positive function that integrates to one. In practical applications $K(\cdot)$ is usually the Gaussian normal formula as given in (5). Other commonly used kernel functions are Epanechnikov, uniform, triangular, biweight [24]. One can prove that selection of a particular kernel function is not critical, as all of them guarantee obtaining similar results (Epanechnikov kernel function is theoretically the most effective, but others are only slightly worse, see [35] for details). However, the bandwidth is the parameter which exhibits a strong influence on the resulting estimate (shape of the curve). 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 (3).

Equation (3) 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. From the other side, 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 H matrix is $d \times d$. This matrix is positive definite and symmetric by definition. So, the equivalent of the formula (3) is now defined as follows:

$$\hat{f}(x, H) = n^{-1} \sum_{i=1}^{n} K_H\left(x - X_i\right) = n^{-1}|H|^{-1/2} \sum_{i=1}^{n} K\left(H^{-1/2}(x - X_i)\right) \tag{6}$$

where

$$K_H(u) = |H|^{-1/2} K(H^{-1/2}u) \tag{7}$$

and $K(\cdot)$ is defined by (5).

Is easy to note that (7) is not a pure equivalent to (4) as for univariate case the 1×1 bandwidth matrix is $H = h^2$. So, now we are dealing with so called 'squared bandwidths'.

A version between the simplest and the most complex is also considered in literature and is based on simplifying the unconstrained bandwidth matrix H to is constrained equivalent where all the off-diagonal entries are zeros by definition. In the paper we do not consider this case. All the possible versions of the bandwidth selectors are investigated in details in [9]. Below we only sum up the three main versions of the bandwidth matrices.

$$h^2 I = \begin{bmatrix} h^2 & 0 \\ 0 & h^2 \end{bmatrix}, \quad diag H = \begin{bmatrix} h_1^2 & 0 \\ 0 & h_2^2 \end{bmatrix}, \quad H = \begin{bmatrix} h_1^2 & h_{12} \\ h_{12} & h_2^2 \end{bmatrix}. \tag{8}$$

Finally, one must remember that nonparametric estimators (to be effective) need more and more data as dimensionality increases. Here, a negative phenomenon called *curse of dimensionality* becomes a real problem. As a consequence, values $\hat{f}(x)$ calculated from (3) or (6) becomes inaccurate. The problem of determining the required sample size needed to achieve a given level of accuracy is studied by some authors. See for example [31,33].

As an example of how KDE works consider a toy dataset of 8 data points $x = \{0,\ 1,\ 1.1,\ 1.5,\ 1.9,\ 2.8,\ 2.9,\ 3.5\}$. Three different PDFs based on these data are depicted in figure 1. It is easy to notice how the bandwidth h influences the shape of the PDF curve. Choosing the best value of h is not a trivial task and this problem was and still is extensively studied in literature [31,33,35]. In Figure 1 lines in bold show the estimated PDFs, while normal lines show the shapes of individual kernel functions $K(x)$ (Gaussian normal kernels). Dots represent the data points x_i.

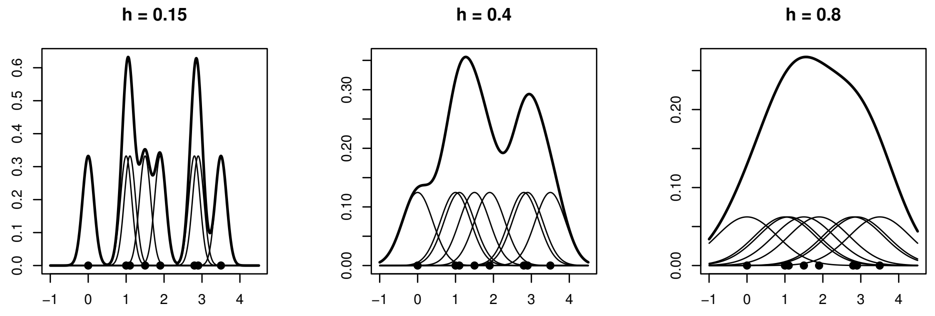


Fig. 1: An example of using kernel density estimators for determining the probability density function

4.3 Probability density functions in databases

A commonly used logical structure in DW area is the multidimensional cube [14]. The particular dimensions can store both *nominal* (also called *categorical*) as well as *numerical* data (both discrete and continuous). In the case of nominal ones, determining all the possible aggregates is not trivial but feasible task. Of course, in practice usually not all possible aggregates are calculated, as the time and storage requirements would be prohibitive. In contrast, if a dimension stores numerical data one must arbitrary discretize them (by creating separate bins). Calculating an optimal number of bins and their width is not a trivial task. After discretization, the total number of aggregates which are (or will be in the future) possible to create is fixed. To avoid the need for preliminary discretization of data, one can consider using PDFs. As a matter of fact, to calculate them we must read all the data stored (or eventually, only a representative sample of the data). But, what is very important, we need to read them only once. Usually we can assume that *statistical properties* of the data remain constant in a period of time (days, months, sometimes even years). So, there is no need to recalculate PDFs after every SQL DML statement executed by a user (INSERT, UPDATE, DELETE). The obvious benefits of PDFs comparing to materialized aggregates is that permanent storing of the former is extremely easy and cheap. Everything what we need to store are either parameters of formula-based PDFs (for example mean value μ and standard deviation σ in the case of Gaussian normal PDF) or output points $\hat{f}(x)$ for nonparametric-based PDFs.

Now let us consider a relational table where an attribute *column*1 is numerical in nature. If we want to calculate the number of records (*count* aggregate operator is SQL) in the range $a \leq column1 \leq b$ one can use formulas given in section 4. If n is the total number of records, it is trivial to conclude that *approximate* count aggregate can be calculates as

$$Count = n \cdot \int_a^b f(x)dx. \tag{9}$$

Similarly, the sum of all record (*sum* aggregate operator is SQL) in the given range can be calculates as

$$Sum = n \cdot \int_a^b xf(x)dx. \tag{10}$$

The similar reasoning leads to conclusion that the product $Sum/Count$ is an equivalent to aggregate SQL's *average* operator. The aforementioned three formulas can be immediately generalised into multidimensional case, that is for 2D case we have

$$\begin{aligned} Count &= n \cdot \int_c^d \int_a^b f(x,y)dxdy, \\ Sum &= n \cdot \int_c^d \int_a^b h(x,y)f(x,y)dxdy. \end{aligned} \tag{11}$$

Here $h(x, y)$ is in general a function of two random variables. To calculate for example *sum* of the variable x, one have to set $h(x, y) = x$. Above we calculate the sum and the number of records within logical range ($a \leq column1 \leq b$) AND ($c \leq column2 \leq d$).

The values of the integrals can be done analytically (if the analytical solution exists) or by applying a proper numerical integration method (some of them were mentioned above in section 4.1). In the case of nonparametric approach, only numerical integration is practicable. In [12] we give a few practical examples based on real datasets which proof the practical usefulness of using kernel based PDFs for calculating database aggregates.

4.4 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 literature as *univariate* problems. Contrary to the PLUGIN method, the LSCV method is designed for both 1D as well as nD problems (known in 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 equations (3) and (8)). From the other side, the most computational complex version assumes that the bandwidth matrix H is used, instead of the scalar h (see equation (6)). This matrix is positive definite and symmetric.

Below, in the next 3 subsections, we give a very condensed recipes for calculating optimal bandwidth h and H using PLUGIN and LSCV approaches described briefly above in section 2.3. The LSCV one is presented in two variants: the simplified one for evaluating optimal h (that is if the density estimate is calculated from (3); this variant will be called LSCV_h) and the general multivariate variant of the method (that is if the density estimate is calculated from (6); this variant will be called LSCV_H). We comment the individual mathematical formulas very briefly as this is beyond the scope of the paper. Every subsection is prefaced with very short overview of the main ideas of the methods. All the necessary details on the methods as well as details on deriving of particular formulas can be found in many source materials, see for example probably the most often cited books [35,31,33].

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 *mean integrated squared error* (MISE) and its asymptotic approximation (AMISE).

PLUGIN This method is used for calculating an optimal bandwidh h for univariate problems, that is applicable to formula (3) where d is set to 1. First we calculate the variance and the standard variation estimators of the input data (equations (12) and (13)). Then we calculate some more complex formulas (equations from (14) to (18)). The explanations of the essence of them is beyond the scope of the paper and can be found in many books on kernel estimators, for example see the three above-mentioned books. Finally, after completing all the necessary components we can substitute them into equation (19) to get the searched optimal bandwidth h.

1. Calculate value of variance estimator:

$$\hat{V} = \frac{1}{n-1}\sum_{i=1}^{n} X_i^2 - \frac{1}{n(n-1)}\left(\sum_{i=1}^{n} X_i\right)^2 . \tag{12}$$

2. Calculate value of standard deviation estimator:

$$\hat{\sigma} = \sqrt{\hat{V}}. \tag{13}$$

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

$$\hat{\Psi}_8^{NS} = \frac{105}{32\sqrt{\pi}\hat{\sigma}^9}. \tag{14}$$

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

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

$$K^6(0) = -\frac{15}{\sqrt{2\pi}},$$

$$\mu_2(K) = 1.$$

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

$$\hat{\Psi}_6(g_1) = \frac{2}{n^2 g_1^7} \sum_{i=1}^{n} \sum_{j=1,i<j}^{n} K^{(6)} \left(\frac{X_i - X_j}{g_1} \right) + nK^{(6)}(0), \tag{16}$$

$$K^6(x) = \frac{1}{\sqrt{2\pi}} \left(x^6 - 15x^4 + 45x^2 - 15 \right) e^{-\frac{1}{2}x^2}.$$

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

$$g_2 = \left(\frac{-2K^4(0)}{\mu_2(K)\hat{\Psi}_6(g_1) n} \right)^{1/7}, \tag{17}$$

$$K^4(0) = \frac{3}{\sqrt{2\pi}},$$

$$\mu_2(K) = 1.$$

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

$$\hat{\Psi}_4(g_2) = \frac{2}{n^2 g_2^5} \sum_{i=1}^{n} \sum_{j=1,i<j}^{n} K^{(4)} \left(\frac{X_i - X_j}{g_2} \right) + nK^{(4)}(0), \tag{18}$$

$$K^4(x) = \frac{1}{\sqrt{2\pi}} \left(x^4 - 6x^2 + 3 \right) e^{-\frac{1}{2}x^2}.$$

8. Calculate the final value of bandwidth h:

$$h = \left(\frac{R(K)}{\mu_2(K)^2 \hat{\Psi}_4(g_2) n} \right)^{1/5}, \tag{19}$$

$$R(K) = \frac{1}{2\sqrt{\pi}},$$

$$\mu_2(K) = 1.$$

LSCV_h This method is used for calculating an optimal bandwidh h for both univariate and multivariate problems, that is applicable to formula (3) where d

is set to any integer value qreter or equal to 1. First we calculate the covariance matrix (equation (21)), its determinant and its inversion. Then we form an objective function (24) which will be minimized according to the searched bandwidth h. After determining the search range of the bandwidth h (equation (29)) as well as the starting bandwidth h (equation (28)), using the "brute force" strategy we search for such h which minimizes the objective function (24). The performance of the brute force method seems to be acceptable in almost all practical applications. If, however it will turn out to be too slow in practice, a more smart (and faster) method can be used, for example the Golden ratio criterion.

1. Calculate covariance matrix, given the input data. In columns one can find consecutive d-dimensinal vectors of our experimental input data. There are n such vectors:

$$X = [X_1, X_2, \cdots X_n] = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d,1} & x_{d,2} & \cdots & x_{d,n} \end{bmatrix} . \tag{20}$$

   Covariance matrix is equal to:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{1,2} & \cdots & \sigma_{1,d} \\ \sigma_{2,1} & \sigma_2^2 & \cdots & \sigma_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{d,1} & \sigma_{d,2} & \cdots & \sigma_d^2 \end{bmatrix} . \tag{21}$$

   where:
   - σ_i^2 - is a variance of each dimension of the random variable,
   - σ_{i_1,i_2} - is a covariance between random variables i_1 and i_2,

$$\sigma_i^2 = \frac{1}{n-1} \sum_{j=1}^{n} x_{i,j}^2 - \frac{1}{n(n-1)} \left(\sum_{j=1}^{n} x_{i,j} \right)^2 , \tag{22}$$

$$\sigma_{i_1,i_2} = \frac{1}{n-1} \sum_{j=1}^{n} x_{i_1,j} x_{i_2,j} - \frac{1}{n(n-1)} \sum_{j=1}^{n} x_{i_1,j} \sum_{j=1}^{n} x_{i_2,j} . \tag{23}$$

2. Calculate determinant of the covariance matrix Σ: $det(\Sigma)$.
3. Calculate inverse of the covariance matrix Σ: Σ^{-1}.
4. Let X_i be the i-th column of the input matrix X. The LSCV_h method requires to minimize the function $g(h)$:

$$g(h) = h^{-d}\left[2n^{-2}\sum_{i=1}^{n}\sum_{j=1,i<j}^{n} T\left(\frac{X_i - X_j}{h}\right) + n^{-1}R(K)\right], \tag{24}$$

where

$$T(u) = (K * K)(u) - 2K(u), \tag{25}$$

$$K(u) = (2\pi)^{-d/2}det(\Sigma)^{-1/2}exp\left(-\frac{1}{2}u^T\Sigma^{-1}u\right). \tag{26}$$

$$(K * K)(u) = (4\pi)^{-d/2}det(\Sigma)^{-1/2}exp\left(-\frac{1}{4}u^T\Sigma^{-1}u\right). \tag{27}$$

5. Calculate the approximate value of the bandwidth h:

$$\begin{aligned} h_0 &= \left(\frac{R(K)}{\mu_2(K)^2R(f'')n}\right)^{1/(d+4)}, \\ \frac{R(K)}{\mu_2(K)^2} &= \frac{1}{2^d\pi^{d/2}d^2}, \\ R(f'') &= \frac{d(d+2)}{2^{d+2}\pi^{d/2}}. \end{aligned} \tag{28}$$

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

$$Z(h_0) = [h_0/4, 4h_0]. \tag{29}$$

The optimal bandwidth h is equal to:

$$argmin_{h\in Z(h_0)}g(h). \tag{30}$$

LSCV_H This method is used for calculating the optimal bandwidh matrix H for both univariate and multivariate problems, that is applicable to formula (6). In this variant of the LSCV method the objective function is defined by equation (32). Now our goal is to find such the bandwich matrix H which will minimize this objective function. This is the classical nonlinear optimization problem and can be solved by using for example the well known Nelder-Mead method [27]. This method needs a starting matrix which can be calculated from the *rule-of-thumb* heuristic equation (37) taken from [35].

Two detail notes on evaluating the objective function (32) are needed here. First, as we said above, Nelder-Mead method can be used for finding the optimal H bandwidth matrix. This method does not guarantee that in every step the current H is positive–definite. This requirement, however is necessary as the bandwidth H matrix components must be by definition positive scalars. So, the searching for the optimal H must be done over the space of all positive–definite matrices only. Moreover, while calculating the current value of (32) the inversion of H is needed, hence positive–definite requirement is a must.

Second, we know that bandwidth matrix H is always a symmetric one (see (8)) 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 $vech(H)$, where $vech$ (vector half) operator takes a symmetric $d \times d$ matrix and stacks the lower triangular half into a single vector of length $d(d+1)/2$. That is for an example matrix we have

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}, \ vech(A) = \begin{bmatrix} 1 & 2 & 3 & 5 & 6 & 9 \end{bmatrix}. \tag{31}$$

1. Let

$$g(H) = 2n^{-2} \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} T_H(X_i - X_j) + n^{-1} R(K) \tag{32}$$

where

$$T_H(X_i - X_j) = (K * K)_H(X_i - X_j) - 2K_H(X_i - X_j), \tag{33}$$

$$K_H(X_i - X_j) = \\ (2\pi)^{-d/2} |H|^{-1/2} exp\left(-\frac{1}{2}(X_i - X_j)^T H^{-1} (X_i - X_j), \right) \tag{34}$$

$$(K * K)_H(X_i - X_j) = \\ (4\pi)^{-d/2} |H|^{-1/2} exp\left(-\frac{1}{4}(X_i - X_j)^T H^{-1} (X_i - X_j) \right), \tag{35}$$

$$R(K) = 2^{-d} \pi^{-d/2} |H|^{-1/2}. \tag{36}$$

2. Find H which minimizes $g(H)$. Start from (Σ is defined by (21)):

$$H_{start} = (4/(d+2))^{1/(d+4)} n^{-1/(d+4)} \Sigma^{1/2}. \tag{37}$$

4.5 Some formula modifications

The equations for LSCV_h algorithm may be reformulated to require less operations. Let us consider equation (26):

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

As can be determined from equation (24), the u is always equal to $\frac{X_i-X_j}{h}$. Let us therefore reformulate the equation (26):

$$\begin{aligned}\tilde{K}(v) &= K(v/h) = \\ &= (2\pi)^{-d/2}|\Sigma|^{-1/2}exp\left(-\frac{1}{2}\frac{v}{h}^T\Sigma^{-1}\frac{v}{h}\right) = \\ &= (2\pi)^{-d/2}|\Sigma|^{-1/2}exp\left(-\frac{1}{2}\frac{1}{h^2}v^T\Sigma^{-1}v\right).\end{aligned} \tag{38}$$

Let

$$S(v) = v^T\Sigma^{-1}v. \tag{39}$$

If we substitute this into equation (38) we obtain:

$$\tilde{K}(v) = (2\pi)^{-d/2}|\Sigma|^{-1/2}exp\left(-\frac{1}{2}\frac{1}{h^2}S(v)\right). \tag{40}$$

Analogous changes can be made to equation (27):

$$(\tilde{K}*\tilde{K})(v) = (4\pi)^{-d/2}|\Sigma|^{-1/2}exp\left(-\frac{1}{4}\frac{1}{h^2}S(v)\right). \tag{41}$$

These changes can be next propagated to equations (24) and (25):

$$\tilde{T}(v) = T\left(\frac{v}{h}\right) = (\tilde{K}*\tilde{K})(v) - 2\tilde{K}(v) \tag{42}$$

$$g(h) = h^{-d}\left[2n^{-2}\sum_{i=1}^{n}\sum_{j=1,i<j}^{n}\tilde{T}\left(X_i - X_j\right) + n^{-1}R(K)\right] \tag{43}$$

It is easy to notice that $S(v)$ values are scalars, and moreover they are constant, independent on the value of 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 minimum of

$g(h)$. Let us determine the complexity of calculating the $g(h)$ function before and after modifications. Calculating of a single exponent value in either $K(u)$ or $(K*K)(u)$ function requires $O(d^2)$ operations (see section 5.5). 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 $g(h)$ function needs to be calculated during searching for its minimum. Total complexity of unmodified LSCV_h algorithm is therefore $O(n_hn^2d^2)$

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

The solutions described above unfortunately cannot be used for optimizing of the LSCV_H algorithm. This is due to the fact that the expression $(X_i - X_j)^TH^{-1}(X_i - X_j)$ found in equations (34) and (35) (which is an equivalent to $S(v)$) depends on the $g(H)$ (equation (32)) function argument. Consequently, this expression must be recomputed each time the $g(H)$ function is computed.

5 Optimization of bandwith selection on SIMD architectures

In this section we describe parallel algorithms for bandwidth selection and implementation details for two out of three implementations compared in this paper: SSE implementation and GPU implementation. Sequential implementation will be described in section 8.

5.1 Identification of some compute-intensive parts in mathematical formulas

Let us take a closer look at equations (12), (16), (18), (32) and (43). All of these equations (among other operations) compute sums of large number of values. As such sums are performed multiple times and constitute a large part of the number of basic matematical operations computed in these equations. Accelerating them would significantly increase algorithm performance. Consequently, in general we need an algorithm which given a single row matrix A, would compute:

$$R(A) = \sum_{i=1}^{n} A_i. \tag{44}$$

The process of using the same operation multiple times on an array of values to obtain a single value (sum, multiplication, mininimum, maximum, variance, count, average etc.) is called *reduction of an array.* Parallel reduction of large arrays is a known problem [36,15]. Basic algorithm as well as GPU and SSE implementations, are described in section 5.2.

Let us consider the equation (12). This equation 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:

$$R_{fun}(A) = \sum_{i=1}^{n} fun(A_i). \tag{45}$$

Parallel computation of such sums can be performed by using a simple modification of parallel reduction algorithms. For details refer to section 5.3.

Let us now consider equations (16) and (18). 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:

$$RR_{fun}(A) = \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} fun(A_i - A_j). \tag{46}$$

As can be easily noticed, the function $fun(x)$ is computed for a difference between every combination of values from row matrix A. Parallel algorithms for computing such sums are given in section 5.4.

Similar sums can also be found in equations (32) and (43). These sums, given a two dimensional A matrix, and function $fun(x)$ are equivalent to:

$$RR^{v}_{fun}(A) = \sum_{i=1}^{n} \sum_{j=1, i<j}^{n} fun(A_{:,i} - A_{:,j}).^{1} \tag{47}$$

In these equations however, each argument of the function $fun(x)$ is a vector and computation of this function is much more complex. Moreover, in both cases function $fun(x)$ can be expressed as: $fun(x) = fun1(fun2(x))$ where

$$fun2(x) = x^T M x, \tag{48}$$

M is any matrix and $fun1(y)$ is any scalar function. Let us now consider equation (43). Here, the function $fun(x)$ is an equivalent of the function $\tilde{T}(v)$ presented in equation (42). Function $\tilde{T}(v)$ is computed using functions $\tilde{K}(v)$ (equation (40)) and $(\tilde{K} * \tilde{K})(v)$ (equation (41)). These functions in turn can be computed based on a value of the function $S(v)$ (equation (39)) which is an equivalent of $fun2(x)$. As was mentioned earlier in section 4.5, the $S(v)$ values can be precomputed and used each time equation (43) is computed. We can therefore split the problem of computing the sums in equation (43) into two problems: (a) computing $fun2(x)$ ($S(v)$) values (see section 5.5) and (b) finding a sum of values of a scalar function introduced earlier. Section 6.2 presents details on how to use precomputed $S(v)$ values in LSCV_h algorithm. Similar observations can be also made for equation (32). Here, the function $fun(x)$ is an equivalent of the function

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

$T_H(X_i - X_j)$ presented in equation (33). Function $T_H(X_i - X_j)$ is computed using functions $K_H(X_i - X_j)$ (equation (34)) and $(K * K)_H(X_i - X_j)$ (equation (35)). 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 values of $fun2$ cannot be precomputed as matrix H^{-1} is different every time equation (32) is computed. Nonetheless solutions presented in section 5.5 can also be used in this case. Section 6.3 presents how to efficiently compute equation (32).

5.2 Parallel reduction of array values $R(A)$

Basic algorithm In this section we briefly introduce basic ideas behind the well known problem of parallel reduction of array values [36], i.e. given a single row matrix A, we present an algorithm which computes

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

Basic idea behind the parallel reduction of arrays is presented in Figure 2. At the beginning the array contains 8 different values. As a first step, pairs of neighbor values from the input array should be added in parallel. Results are stored in the first half of the array. This process is repeated but each time it runs only on the first half of the array which was the input to the previous step. The algorithm stops when the input part of an array is composed on only a single value, which is the result of the parallel reduction. This approach may be easily generalized to larger arrays of values, even of non power-of-two sizes. In such a case, values whose “pairs” land outside of array bounds, are left unchanged. The above generic method for value reduction will be adapted in the following sections to accelerate several specific parts of the equations.

GPU implementation GPU based algorithm for reduction of a large number of values was proposed in [15] where the author introduces several low-level optimizations which include:

- utilizing a small but very fast shared memory available on GPUs (see section 3.2),
- using alternative reduction scheme (pairs of added values are not neighbours in the array),
- unrolling of loops and utilizing templates to achieve very efficient gpu-kernel for reduction of arrays.

Paper [15] presents a gpu-kernel in CUDA C code, which performs reduction of up to 1024 values (using 512 threads) stored in the shared memory array. Graphics cards with CC$\geq$2.0 may start up to 1024 threads per block, so this code may also be appropriately extended to support arrays of size up to 2048 values. The same paper also introduces a schema which utilizes this gpu-kernel to support arbitrary sized arrays:

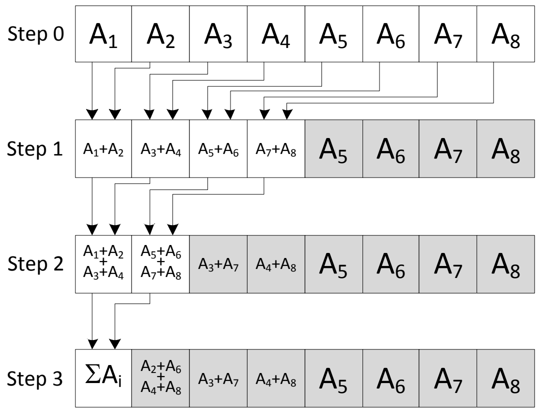


Fig. 2: Parallel reduction schema

1. Divide the input dataset into subsets of consecutive values of size equal to some power of 2 (1024 or 2048 depending on compute capability).
2. Start a one dimensional block of threads for each of these subsets (in a one dimensional grid). Each block should be composed of the number of threads equal to the half of the size of the subset.
3. Each thread in each block should retrieve two values from the input array (which is initially stored in the slow global memory), add them and store the result in the fast, shared memory. If for some reason grid of threads is not big enough to process all data, each thread should add more then two values.
4. Next, the threads should perform the parallel reduction algorithm on the values stored in the shared memory.
5. The resulting value (for each block) should be stored in the output array in the global memory.
6. The above steps should be repeated on the output array of the previous iteration until only a single value is obtained.

SSE implementation Our SSE implementation uses a horizontal addition instrinsic instruction `__m128 _mm_hadd_ps(__m128 x,__m128 y)` where `x` and `y` are two 128 bit vectors of four floats. Given two vectors `a=[a1,a2,a3,a4]` and `b=[a5,a6,a7,a8]` the instruction computes one vector containing `[a1+a2,a3+a4,a5+a6,a7+a8]`, i.e. it adds neighbour values. As may be easily noticed, the `__m128 _mm_hadd_ps(__m128 x,__m128 y)` instruction performs first iteration of reduction schema presented in Figure 2. For bigger arrays, each iteration, except the last two (reducing four values into one), may be performed by executing the instruction on consecutive elements of the array appropriate number of times.

The algorithm described above requires many memory accesses and may therefore suffer delays stemming from memory access latency. The problem would be largely reduced if the whole temporary array would fit within the CPU's cache. To increase probability of such situation, the processed array is split into many chunks of arbitrary small (but power-of-two) sizes, such that would fit in the cache. Each such chunk is processed independently and the results are then reduced using the same algorithm (similarly as in the GPU approach presented above).

Notice that the described algorithm has a very low parallelism when compared with the GPU approach (only 4 values are added at the same time). To work around this, we start a number of independent threads, each of which processes a subset of chunks of the processed array. The number of threads dependents on the number of CPU cores and core capability (such as HyperThreading [26]).

Careful reader might notice that SSE implementation could be implemented in a much simpler way by using a simple sequential reduction and standard vector addition intrinsic instruction `__m128 _mm_add_ps(__m128 x,__m128 y)` where `x` and `y` are two 128 bit vectors of four floats. Moreover, by using this method memory access latency could be largely reduced. However, as with all of floating point mathematical computations, reduction algorithms are a subject to numerical rounding errors. Sequential reduction has an error constant of $O(n)$, whereas the pairwise reduction algorithm presented above has an error constant $O(log_2 n)$ [17] and consequently yields smaller errors. One could also argue, that there exists another sequential reduction algorithm introduced by Kahan in [22], which has an error constant of $O(1)$ [17]. This algorithm however requires several times more floating operations and is therefore much slower then simple sequential reduction and pairwise reduction.

5.3 Parallel reduction of scalar function values $R_{fun}(A)$

Basic algorithm In this section we present an algorithm for computing sums equivalent to:

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

where A is any single row matrix and fun is any scalar function. Notice, that the function fun for each iteration is computed independently. Therefore, values of this function may be easily computed in parallel and the obtained values can be reduced (summed up) later using the previously introduced reduction algorithms (MapReduce approach [8]). However, to remove the need for storing function values in memory we suggest to slightly modify the basic array reduction schema described in section 5.2 in such a way that fun function values are computed and added on-the-fly.

GPU implementation GPU implementation is a straightforward modification of the scheme presented in section 5.2. In step 3, after the values are retrieved from the global memory, fun function values are computed. Afterwards, the computed values are added and stored in the shared memory. This is implemented as a separate gpu-kernel, as subsequent reduction (see step 6) is performed without computing of fun function values.

SSE implementation Presence of fun function in SSE implementation of reduction algorithm requires minor adjustments in the basic reduction function. First, a SIMD version of the fun function needs to be implemented. This new version should process four values passed in a single `__m128` vector, and return four values also as a single `__m128` vector. The only modification to the SSE reduction implementation requires that in the first iteration of the algorithm, the SIMD version of the fun function is used on every four value vector retrieved from memory and obtained result is subsequently used in the reduction algorithm.

5.4 Parallel reduction of scalar function in nested sums $RR_{fun}(A)$

Basic algorithm In this section we present an algorithm for computing sums equivalent to:

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

where A is any single row matrix and fun is any scalar function. Based on the solutions presented in [4], we propose the following parallel schema for computing function values, which utilizes cache memory. First, let us notice that fun function values are computed only for indexes i and j such that $i < j$ and might therefore be stored in an upper triangular matrix of size $n \times n$. Lets take a look at Figure 3. The A matrix is divided into chunks of small (power-of-two) size k (recall that here A matrix contains only a single row). The triangular matrix of fun function values is divided into tiles of size $k \times k$. Notice, that tiles on the diagonal contain redundant positions which should not be computed. Each tile corresponds to some combination of two A matrix chunks denoted E and F. For each tile, a group of $k \times k$ threads is started. First, a subset of threads in a group copies the corresponding chunks into the cache memory. Next, each thread in the tile computes the function value based on two A matrix values retrieved from the cache. Redundant threads (below the main diagonal) return function value 0. Next, threads in each tile cooperate to reduce the computed function values into a single value using parallel reduction algorithm introduced earlier. Consequently, each tile yields one reduced value. These values are then formed into a single result, by once again using a parallel reduction algorithm.

The remaining problem is: what to do if the size of the A matrix cannot be expressed as a multiple of the chosen k value? In such a case, the matrix should be extended to the size equal to the nearest multiple of k. Redundant threads allocated to process tiles corresponding to last chunk of the array storing A matrix should output zero as a function value. No additional changes to the above schema are required.

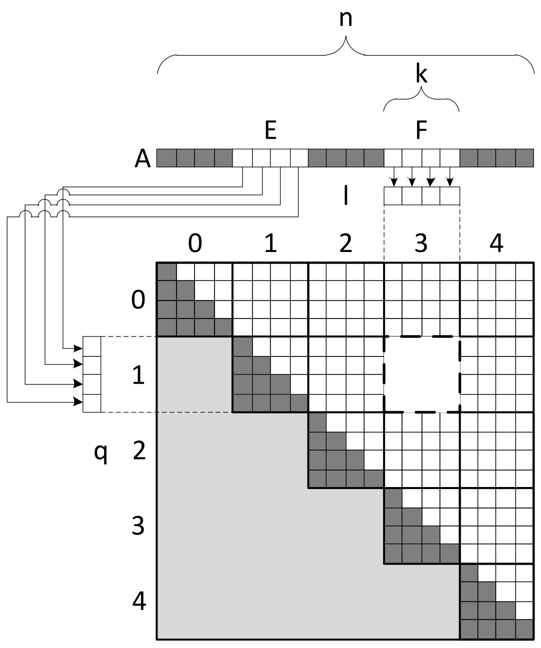


Fig. 3: Parallel cache aware computation of values of two variable functions

GPU implementation The proposed parallel schema seems to fit the GPU architecture and CUDA API pretty well. Each tile can be processed by a square block of threads and the shared memory can be used as a cache. Unfortunately, the computation grids in CUDA API can only be linear or rectangular (no triangular grids). To solve this problem we propose to run a one dimensional grid of blocks, and based on the block number, compute its position in the triangular matrix (see Figure 4). To find the position of the tile in the triangle matrix based on its number on the one dimensional grid, we propose to use the equations (49) and (50) (see appendix A for derivation of these equations), where bx is the block number, l is the corresponding tile column and q is the corresponding tile row:

$$l = \left\lceil \frac{\sqrt{8bx+9}-3}{2} \right\rceil , \tag{49}$$

$$q = bx - \frac{l(l+1)}{2} . \tag{50}$$

Notice, that as the number of blocks in one dimensional grid cannot exceed 65535 (current GPU limitation), the number of tiles in one dimension of the triangle matrix cannot exceed 360. This in turn limits the number of values in the A matrix to only $360k$ columns. To solve this problem we allocate two dimensional grids which are composed of at least the appropriate number of blocks and then we find the "one dimensional" number of the block based on its position in the two dimensional grid. This new position is used in the subsequent computations in equations (49) and (50). The number of blocks started this way may be greater than required. Therefore each thread must detect whether it belongs to one of such redundant blocks and abort computations if necessary.

Given NVIDIA graphics card limitations, $k = 16$ (256 thread blocks) could be used for graphics cards with CC$\leq$1.3 and $k = 32$ (1024 thread blocks) for graphics cards with CC$\geq$2.0. However, as was observed in [15], given an array of n values, only $n/2$ threads are used during reduction. Therefore, after computing of function values in a tile, only half of the threads in each block would be used in subsequent tile values reduction. To solve this problem we propose to use $k = 32$ regardless of graphics cards compute capability, but use blocks of only 256 threads to process tiles. Each thread should compute four fun function values and add them. This way we allow our code to be run on all graphics cards and achieve better thread utilization at the same time.

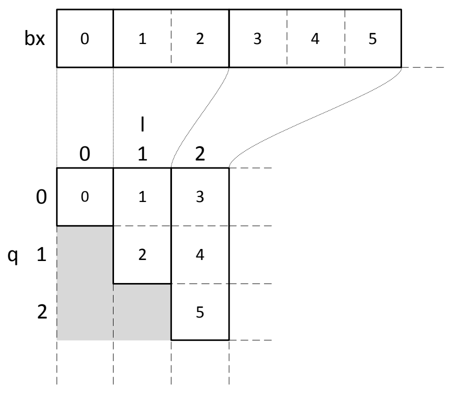


Fig. 4: Conversion of block number to its position in upper triangular matrix

SSE implementation In our SSE implementation each tile is processed by a single thread, though some processing parallelism is achieved due to the usage of SSE SIMD instructions (similarly as in section 5.3 SIMD version of the function fun is needed). Here, cache is implemented as two local arrays E and F (corresponding to chunks E and F) which are read multiple times during computing of tile values and therefore are probably cached. A matrix is divided into 16 value chunks ($k = 16$) and consequently each tile is composed of 256 values. However, each such chunk is divided into 4 four value vectors stored in SSE $\_\_m128$ variables. Processing of each tile is illustrated in Figure 5 and described below:

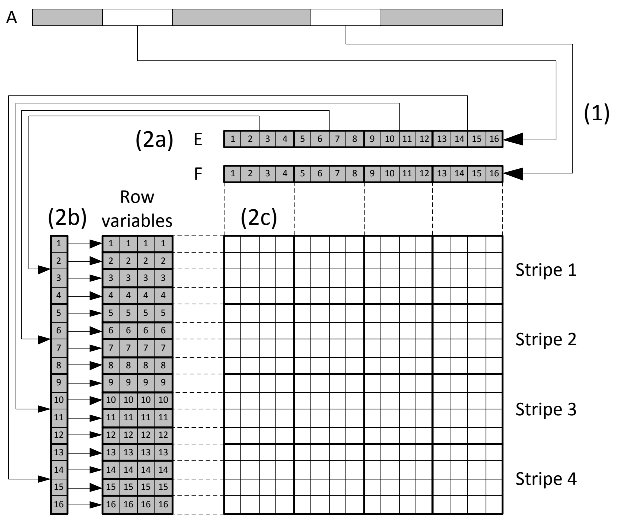


Fig. 5: Processing of a tile in SSE implementation

1. First, the two chunks corresponding to a tile are copied to local arrays E and F.
2. Next, the tile is processed in four stripes. Each stripe is processed as follows:
   (a) A four value vector is retrieved from the E local array which contains "row values".
   (b) Next, these four values are copied into four `__m128` variables (subsequently called row variables).
   (c) Each row of a stripe is processed in four steps, each of which includes the following operations: (1) retrieve a four value vector from F array, (2) compute a difference between appropriate row variable and the retrieved vector using `__m128 _mm_sub_ps(__m128 x, __m128 y)` instruction, (3) compute four fun function values using SIMD version of the fun function and (4) add the obtained results into a single accumulator `__m128` type variable using `__m128 _mm_add_ps(__m128 x, __m128 y)` instruction.
3. After all stripes are processed, the accumulator contains four values. These values are added to obtain final result of tile processing.

Processing of a tile that lies on main diagonal is similar, but several modifications are made. First, only one chunk is retrieved into local array (obviously). Moreover, loops are altered to not process parts of stripes below the main diagonal. Notice however, that some "below diagonal" values in small 4×4 tiles are still computed. In such cases, each vector computed by the reduced function is multiplied by appropriate vectors composed of zeroes and ones to reset the unwanted function results.

The tiles are processed concurrently on a number of threads which are processed most efficiently on a CPU (depending on the number of cores and whether the HyperThreading [26] capability is available or not).

5.5 Computing of $fun2$ function values in $RR^v_{fun}(A)$

Basic algorithm Let us recall that in equations (32) and (43) the sums equivalent to:

$$RR^v_{fun}(A) = \sum_{i=1}^{n} \sum_{j=1,i<j}^{n} fun(A_{:,i} - A_{:,j}) = \sum_{i=1}^{n} \sum_{j=1,i<j}^{n} fun1(fun2(A_{:,i} - A_{:,j}))$$

can be found, where A is any matrix, $fun1$ is any scalar function and $fun2(x) = x^T M x$ where M is also any matrix. As was suggested in section 5.1 parallel processing of such sums can be split into two problems: computing $fun2$ function values and then reducing them using previously introduced algorithm. Consequently, we need an algorithm which given an A matrix would find a triangular matrix $B = [b_{i,j}]$, $i = 1, \ldots, n$, $j = i+1, \ldots, n$, such that:

$$b_{i,j} = fun2(A_{:,i} - A_{:,j}).$$

As each $fun2$ function value may be computed independently, parallel computation algorithm seems obvious. To achieve better performance a cache aware solution similar to the one presented in section 5.4 may be used. Consider the schema presented in Figure 6. 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 A matrix chunks. Row of a tile within the triangular matrix is denoted as q and column is denoted as 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 the function value based on two vectors retrieved from the cached chunks. Each thread detects whether it is over or on the main diagonal or not. If it is below the diagonal it stops further computations. If it is over the main diagonal it computes $fun2$ function value and stores it in the output array. Linear position in the output array may be computed using equation for sum of arithmetic progression, based on the threads 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).

We shall now derive an efficient order of performing operations needed to compute $fun2$ function values within a tile for an array storing A matrix that is row-major aligned in the computers memory. For simplicity let us assume that the considered tile does not lie on the main diagonal. We shall denote the tile of the matrix containing $fun2$ function values as a $k \times k$ submatrix Y. Each such tile corresponds to two $d \times k$ (d is the number of rows in A matrix) chunks E

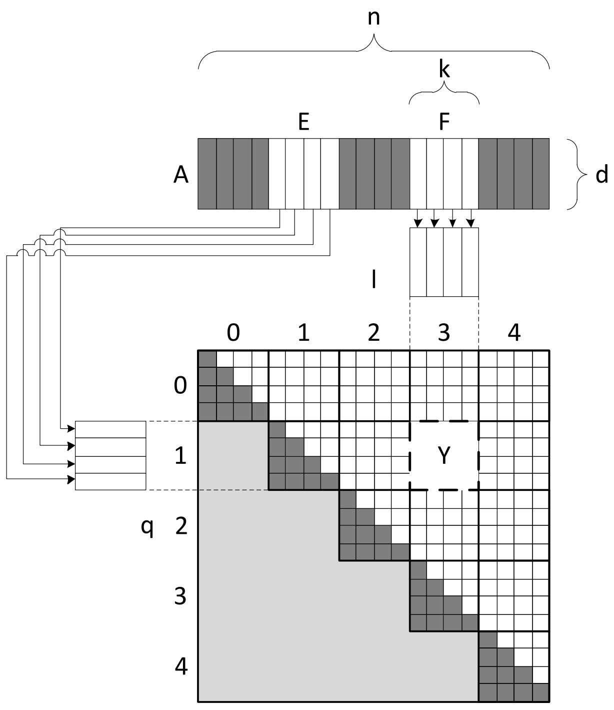


Fig. 6: Parallel computation of $fun2$ function values

and F of the A matrix. Let us assume, that chunks E and F start at columns $qk + 1$ and $lk + 1$ respectively. Let:

$$\begin{aligned} i &= qk + r, \\ j &= lk + p \end{aligned} \tag{51}$$

where $r, p = 1, \dots, k$. 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 function $fun2$, and the new notations introduced above we know that:

$$y_{r,p} = fun2(v^{r,p}) = (v^{r,p})^T M v^{r,p}. \tag{52}$$

Let us extract the first matrix multiplication operation from the equation (52):

$$z^{r,p} = (v^{r,p})^T M. \tag{53}$$

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

$$z^{r,p} = [z_a^{r,p}]_{a=1,\dots,d} \tag{54}$$

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

$$z_a^{r,p} = (v^{r,p})^T M_{:,a} = \sum_{c=1}^{d} v_c^{r,p} m_{c,a}. \tag{55}$$

Let us now substitute the equation (54) into equation (52):

$$y_{r,p} = z^{r,p} v^{r,p} = [z_a^{r,p}]_{a=1,\dots,d} \, v^{r,p}. \tag{56}$$

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

$$y_{r,p} = [z_a^{r,p}]_{a=1,\dots,d} \, [v_a^{r,p}]_{a=1,\dots,d} = \sum_{a=1}^{d} z_a^{r,p} v_a^{r,p}. \tag{57}$$

If we substitute equation (55) into equation (57) we obtain:

$$y_{r,p} = \sum_{a=1}^{d} \left(\sum_{c=1}^{d} v_c^{r,p} m_{c,a} \right) v_a^{r,p}. \tag{58}$$

Recall that $v_x^{r,p} = e_{x,r} - f_{x,p}$. If we substitute this into equation (58) we obtain:

$$y_{r,p} = \sum_{a=1}^{d} \left(\sum_{c=1}^{d} (e_{c,r} - f_{c,p}) m_{c,a} \right) (e_{a,r} - f_{a,p}). \tag{59}$$

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 Y matrix. This is accomplished by replacing each occurence of the column number p with the colon which means "all available values". To retain 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:

$$Y_{r,:} = \sum_{a=1}^{d} \left(\sum_{c=1}^{d} ([e_{c,r}]_k - F_{c,:}) m_{c,a} \right) ([e_{a,r}]_k - F_{a,:}) \tag{60}$$

Notice, that equation (60) 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 $r = 1, \dots, k$ perform the following steps:
  1. $Y_{r,:} \leftarrow [0]_k$
  2. for each $a = 1, \dots, d$ 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) $Y_{r,:} \leftarrow Y_{r,:} + part * ([e_{a,r}]_k - F_{a,:})$
3. Output $Y_{r,:}$
```

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

GPU implementation Our GPU implementation is a straightforward implementation of the parallel schema presented in section 5.5. Each tile is processed by a group of 256 threads ($k = 16$). We have also limited lengths of vectors to up to 16 values ($d \leq 16$) to simplify copying of data to shared memory (acting as cache) and subsequent processing (256 threads can copy 16 vectors of 16 values in parallel). As CUDA threads are not processed independently (they are run in 32 thread SIMD groups) and are a subject to memory access restrictions, loops computing $fun2$ function values, as well as global memory to shared memory copying code, have to be carefully designed and data has to be properly layed out in memory in order to avoid serialization of memory accesses.

Let us start with copying of data. A matrix is stored in an array in GPUs global memory in row major order. Consequently, subsequent values in the memory belong to subsequent matrix columns. Recall that CUDA threads are grouped in warps. Consider a single memory access instruction. Depending on the compute capability of the graphics card one memory transaction is performed per halfwarp (compute capability ≤ 1.3) or one per warp (compute capability ≥ 2.0), as long as each accessed address lies within the same 128B segment within the global memory. On graphics cards with compute capability (≤ 1.1) additional restrictions on accessed addresses are imposed. Given the fact that $k = 16$, 16 columns should be copied to the shared memory. To adhere to the memory access requirements, threads access consecutive memory addresses (i.e. they copy "rows" of the processed chunk of matrix). Consequently, a single thread warp will copy two rows of data. For GPUs with CC$\leq$1.3, a memory transaction per halfwarp is perfomed and consequently, each row will be retrieved in a single 64B memory transaction. For GPUs with CC$\geq$2.0, even though one 128B transaction can be performed per warp, here two 128B transactions will be made, as each row is in different memory segment. Consequently for graphics cards with CC$\geq$2.0, memory retrieval is not as efficient as for the graphics cards with CC$\leq$1.3 (unless we increase k to 32). However, the excessive retrieved rows will be cached and might benefit other thread blocks. Moreover, this step is short, when compared to next computation steps and the delay should be negligible. Copying of data from global memory to shared memory is depicted in Figure 7.

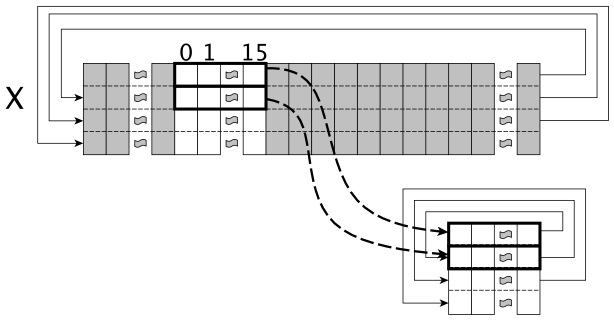


Fig. 7: Copying data to shared memory

To compute $y^{r,p}$ values M matrix is needed as well. Array storing M matrix is stored in row major order in constant memory. Let us assume that the shared memory array storing columns from the chunk corresponding to the "row" coordinate of the triangle matrix tile is denoted E whereas the shared memory array storing columns from the chunk corresponding to the "column" coordinate of the triangle matrix tile is denoted F. By tx and ty we denote thread coordinates within a block. The main loop of a single thread computing $fun2$ function value is as follows:

```
1. $p \leftarrow tx$
2. $r \leftarrow ty$
3. $yrp \leftarrow 0$
4. For each consecutive row $a$
   (a) $part \leftarrow 0$
   (b) For each consecutive row $c$
           $part \leftarrow part + M[c * d + a] * (E[c * k + r] - F[c * k + p])$
   (c) $yrp \leftarrow yrp+ = part * (E[a * k + r] - F[a * k + p])$
```

It is easy to notice that this algorithm is a straightforward implementation of the equation (59), where the r and p coordinates of a computed value within a tile correspond to threads location within a block. Notice, that each thread in a warp will access the same value in the M array at each iteration of the loop. This is an optimal access pattern for the constant memory where this matrix is stored.

Let us now analyse the access patterns to the shared memory arrays E and F. Data in these arrays is stored in row major order. This is due to the fact that A matrix is stored in row major order in the global memory and code that retrieves data from A matrix into arrays E and F retains this order. Consequently, first k values in both of these arrays are the values of the first row of the corresponding chunk of the A matrix. Next k values correspond to the second row, and so on. Consequently, consecutive values of each column are stored every k values. Such storage allows our gpu-kernel to access data in the arrays E and F optimally.

We shall start with an explanation for graphics cards with compute capabilities ≤ 1.3. Let us assume that $k = 16$. Consequently, each half-warp processes one row of a block. As conflicts may only appear between threads in a single half-warp, our subsequent analysis will consider only one row of a block. In Table 1 we present for each thread in a halfwarp its corresponding p and r (tx and ty), array address which will be accessed in both E and F arrays as well as corresponding shared memory banks. Each consecutive thread is assigned consecutive tx values [4]. Given the fact, that there are 16 shared memory banks, each thread in the half-warp accesses array F through a different bank, which in turn guarantees that there are no conflicts. Let us now consider the E array. It would seem, that in each iteration every thread accesses the same bank. However, it not only accesses the same bank, but also the same address. Consequently, the broadcast mechanism of shared memory may be used and all such reads are performed in a single memory transaction within a half warp.

For graphics cards with compute capability ≥ 2.0 explanations are similar. The graphics cards with $\geq$2.0 have 32 shared memory banks, and each memory transaction is performed per warp not per half warp. Given this, all of the above explanations are correct if $k = 32$. It is interesting however to notice that even if $k = 16$, our gpu-kernel is also efficient. Please take a look at Table 2. Similarly as in Table 1 we present for each thread in a warp its corresponding p and r (tx and ty), array address which will be accessed in both E and F arrays as well as corresponding shared memory banks. If $k = 16$ than each warp processes two rows of a block. This causes 16 2-way conflicts between threads accessing the same values in the F array (16 pairs of threads access the same bank), and two 16way conflicts between threads accessing the E array (two groups of 16 threads access the same bank). However, it may also be noticed that in all of these conflicts the same values are accessed and may therefore be broadcasted. The graphics cards with CC$\geq$ 2.0, have an improved broadcast mechanism in which all broadcasts are processed in a single shared memory transaction. Consequently, the described conflicts will not cause memory access serialization.

Table 1: Breakdown of memory accesses within a single halfwarp on GPUs with compute capability ≤ 1.3 ($k = 16$)

thread in halfwarp	0	1	2	...	14	15
$tx = p$	0	1	2	...	14	15
$ty = r$	g	g	g	...	g	g
F address $c * k + p$	$16c + 0$	$16c + 1$	$16c + 2$	...	$16c + 14$	$16c + 15$
F bank	0	1	2	...	14	15
E address $c * k + g$	$16c + g$	$16c + g$	$16c + g$	...	$16c + g$	$16c + g$
E bank	g	g	g	...	g	g

Table 2: Breakdown of memory accesses within a single halfwarp on GPUs with compute capability ≥ 2.0 ($k = 16$)

thread in warp	0	1	2	...	14	15
$tx = p$	0	1	2	...	14	15
$ty = r$	g	g	g	...	g	g
F address $c * k + p$	$16c + 0$	$16c + 1$	$16c + 2$	...	$16c + 14$	$16c + 15$
F bank	0	1	2	...	14	15
E address $c * k + r$	$16c + g$	$16c + g$	$16c + g$	...	$16c + g$	$16c + g$
E bank	g	g	g	...	g	g
thread in warp	16	17	18	...	30	31
$tx = p$	0	1	2	...	14	15
$ty = r$	$g + 1$	$g + 1$	$g + 1$	...	$g + 1$	$g + 1$
F address $c * k + p$	$16c + 0$	$16c + 1$	$16c + 2$	...	$16c + 14$	$16c + 15$
F bank	0	1	2	...	14	15
E address $c * k + r$	$16c + g + 1$	$16c + g + 1$	$16c + g + 1$	...	$16c + g + 1$	$16c + g + 1$
E bank	$g + 1$	$g + 1$	$g + 1$	...	$g + 1$	$g + 1$

SSE implementation SSE implementation is a straightforward implementation of the generic algorithm presented in section 5.5. Let us start with description of processing of the off-main diagonal tiles. Here we also use $k = 16$. First, data is copied into local arrays (which represent E and F matrices) which will hopefully be cached. Subsequent code is just an implementation of the solution from section 5.5. One important thing that should be noted is that we use four variables of type `__m128` to implement vector $Y_{r,:}$ and another four variables to implement vector *part*. Main diagonal tiles are processed similarly by a specialized variant of the algorithm. The main differences are that code part that outputs results, ignores the excessive values and processing of parts of $Y_{r,:}$ and *part* vectors is omitted, when applicable.

Utilizing multiple cores, as previously involves running the two variants of the tile processing algorithm in multiple threads.

6 Algorithm implementations

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

6.1 PLUGIN

In our PLUGIN algorithm implementation we utilize parallel reduction algorithms presented in sections 5.2 and 5.3 to compute variance estimator (step 1, section 6.1, equation (12)) and algorithm presented in section 5.4 for computing sums in steps 5 and 7 (section 6.1, equations (16) and (18)). Remaining steps

(2,3,4,6 and 8) are all simple equations that are inherently sequential and therefore cannot be further optimized. Nonetheless they require very small number of operations and can therefore be performed on CPU in negligible time.

6.2 LSCV_h

Our implementations of the LSCV_h algorithm use the modified equations presented in section 4.5. Consequently, the algorithm is performed using the following steps:

1. Compute covariance matrix of matrix X: Σ (see equations (21), (22) and (23)).
2. Compute determinant of the covariance matrix Σ: $det(\Sigma)$.
3. Compute inverse of the covariance matrix Σ: Σ^{-1}.
4. Compute the approximate value of the bandwidth (see equation (28)).
5. Determine the range in which we search for a minimum of the objective function $g(h)$ (see equation (29)).
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 equation (39)).
7. Search for minimum of $g(h)$ objective function within the range computed previously (step 5). Each time objective is computed, its modified version (see equation (43)) should be used, which can be computed based on precomputed values of the $S(v)$ function.

Steps 1 to 5 of the algorithm in all implementations are performed sequentially on CPU without using SSE instructions. Though computing of covariance matrix could be performed easily in parallel by using an algorithm similar to the one presented in section 5.5 we have not implemented this as this step takes very little time when compared to the last steps of the LSCV_h algorithm presented above. Values of $S(v)$ function are precomputed using the algorithm presented in section 5.5. The last step (minimization of $g(h)$ function) is performed by a "brute force" search for a minimum on a 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 4.4 other approaches to minimization of $g(h)$ objective function are also possible. In this paper however we present implementations of the "brute force" method.

GPU implementation In GPU implementation, searching for minimum of $g(h)$ is performed as follows. First, based on the grid density and the width of the range $Z(h_0)$ (equation (29)) the number of different h values to be tested (denoted n_h) are determined. Next, the number of one dimensional blocks needed to perform reduction of the nested sums in equation (43) (denoted n_b) are determined. Given the block size bs this may be computed as: $n_b = \lceil (n(n-1)/2)/(2bs) \rceil$. Given these values, a two dimensional computing grid is started, where each row corresponds to a single tested h value (consequently there are n_h rows). Each row is composed of n_b one dimensional blocks.

Parallel computation of $g(h)$ values for all tested arguments is performed similarly as in algorithm presented in section 5.3. The main differences here are as follows:

- Each thread based on the y coordinate of its block within a computation grid computes tested argument h (its not retrieved from the memory).
- Reduction is performed on the precomputed $S(v)$ values. For each retrieved $S(v)$ value and based on the grid row dependent h argument value, the function $\tilde{T}(v)$ 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 value.

Similarly as in section 5.3 reduction is performed in each block, so each started block yields one reduced value. An $n_h \times n_b$ matrix of values obtained in this way is stored in global memory. As we expect a single value per row, not a single value per block, subsequent reduction is performed in each row independently. This process is repeated until only a single value per grid row is obtained. The obtained values represent nested sums from the equation (43). To find the final $g(h)$ values, for each value obtained during reduction step, a single thread is started, which performs remaining operations of the equation (43). Computed $g(h)$ values are copied to the computers memory and the argument for which the function $g(h)$ has minimal value is found. Notice that the last operation could also be perfomed in parallel on GPU. However, this step can be performed in negligible time so there is no need to accelerate it.

Now we would like to address a problem that comes from GPU limitations. The computation grid can have no more than 65535 rows and columns. Consequently, if n_h or n_b cross this boundary, the corresponding computation grid cannot be started. Solving the problem with too big n_b value is pretty simple. If n_b is bigger than 65535, only 65535 blocks per grid row are started. The algorithm for reduction of values presented in [15] adds more than two values per thread if there are more values to be reduced, than two times the available number of threads. If n_h is bigger than 65535 the set of h values to be tested is divided into portions of size 65535 and the computations described above are performed sequentially for each such portion.

SSE implementation As SSE implementation has much less threads to utilize, we have used a simpler approach. Similarly, as in GPU implementation we first determine the value n_h. In contrast to GPU approach however, we do not compute $g(h)$ function for each h argument in parallel. This is performed sequentially in a loop, and only computation of a single $g(h)$ function is parallelized. To perform nested sums from the equation (43), we use the algorithm presented in section 5.3. This algorithm is started on the precomputed set of $S(v)$ values and for each such value the function $\tilde{T}(v)$ is computed and subsequently reduced into a single value. Based on this value, the final value of $g(h)$ function is computed.

The loop keeps track of the lowest $g(h)$ value found up-to-date (and corresponding h value) and after the loop is finished the result of the LSCV_h algorithm is known.

6.3 LSCV_H

The LSCV_H algorithm can use any numerical function minimization algorithm capable of minimizing $g(H)$ function (see equation (32)). Such algorithms are often inherently sequential as they are based on iterative improvement of results of 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 hope of finding 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 that 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 objective function at least once per iteration to assess currently found solution. Our objective function $g(H)$ can take a long time to compute, and while it is computed, other optimization algorithm steps cannot be processed. Consequently, the time of finding the optimal H matrix can be improved if we optimize computing of $g(H)$ function. Unfortunately, we cannot use a set of precomputed values like the ones used to speed up the LSCV_h algorithm, but we can adapt the algorithm used to compute those values. Compare exponents in equations (40) and (41) (algorithm LSCV_h) with exponents in equations (34) and (35) (algorithm LSCV_H). Both exponents are computed similarly. The main difference is that in LSCV_h algorithm the matrix Σ^{-1} is constant and exponents may be precomputed, while the corresponding H^{-1} matrix in LSCV_H algorithm is not constant. Consequently, to compute a single value of $g(H)$ both steps: computing exponents and reducing T function value have to be performed. To make this solution a little bit more cache friendly, we combine both: exponent finding algorithm described in section 5.5 and function value reduction algorithm presented in section 5.3 into one algorithm.

GPU implementation Let us start with describing the modification of the GPU algorithm. Recall the algorithm description from section 5.5. The processing in this algorithm is done in tiles: 256 threads process 256 combinations of two vectors from the matrix X. Each tile is represented as a single thread block. We adapt this algorithm by adding additional steps for each thread, after it finishes computing of its corresponding value. Based on this value, $T(H)$ function (see equation (33)) is computed and the result is stored in the additional buffer in the shared memory. Next, all threads within a block are synchronized and then perform parallel reduction algorithm on the obtained values. Consequently, after all threads within a block finish processing a tile, a single value, which constitutes partially performed nested sums from the equation (32) is obtained. This value is then stored in global memory for further reduction by using the algorithm from section 5.2. The computed sum is then used to finalize computation of the $g(H)$ function.

SSE implementation Modifications of the SSE algorithm are pretty similar. Recall that here each 16 value row processed is implemented by four `__m128` variables. Consequently, each computed row of a processed tile is returned in this form. After such row is computed, it is not stored in the resulting array, but SIMD implementation of the $T(H)$ function is computed on each of these four variables. Next, all of these variables are added to a single `__m128` accumulator. This process is repeated for all rows computed during processing of a tile. Finally, the four values stored in the accumulator are added to obtain a single value which constitutes partially performed nested sums from the equation (32). These values are then stored into an array for further reduction by using the algorithm from section 5.2. Computed sum is then used to finalize computing of the $g(H)$ function.

7 Experiments

7.1 Environment and implementation versions

For the purpose of experiments we have implemented PLUGIN, LSCV_h and LSCV_H algorithms (see section 6), each in 3 versions: Sequential implementation, SSE implementation and GPU implementation. LSCV_H implementations only implemented computing of the $g(H)$ objective function, as this is the only element of this algorithm that has influence on its performance. We have also made a small change in LSCV_h algorithm. To make performance of this algorithm completely data independent, we have slightly modified the last part of the algorithm where we search for a minimum of the objective function on a grid. The objective function is computed for a fixed number of points (150) within the computed interval as opposed to section 4.4 where only distance between the tested points is specified, and their number is dependent on the width of the computed interval. In all versions we have used ALGLIB library [5] to perform matrix square root and Horner's method for accelerating calculations of polynomial values (in portions of equations (16) and (18)). Each implementation uses single precision arithmetic. As was stated in section 3.2 single precision arithmetic is much more efficient on GPUs than double precision. Notheless, performance of double precision computations on GPUs grow with each new graphics card and should be adequate in the near future. Only simple modifications to the presented solutions, which take into account mainly efficient access to shared memory, need to be made in order to accomodate double precision. To make the results of performance tests of CPU and GPU comparable we have decided that the Sequential and SSE implementations should also utilize single precision. Still one should notice that both of these implementations could utilize double precision as well. For sequential implementations changes are trivial.

SSE implementation could be changed in two ways to accomodate double precision: either operate on 128bit vectors holding two double precision instead of four single precision values (utilize SSE2 instructions) or operate on 256bit vectors containing four double precision values (utilize AVX instructions and new registers). Below we give a short description of each of the versions.

– **Sequential implementation** - a “pure C” straightforward sequential implementation of formulas presented in sections 4.4 (PLUGIN algorithm), 4.4 and 4.5 (LSCV_h algorithm) algorithm, and 4.4 (LSCV_H algorithm). 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 awarness in this implementation is reflected in loops construction and memory storage which avoids non linear memory accesses (up to transposing matrices if necessary). This implementation does not use any SSE instructions and is single threaded. Reductions are performed only using sequential implementation of the hierarchichal algorithm presented in section 5.3.
– **GPU implementation** - implementation that accelerates computations using CUDA API to execute computations on a GPU. 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. For details see section 6.1 (PLUGIN algorithm), 6.2 (LSCV_h algorithm) and 6.3 (LSCV_H algorithm).
– **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 [6] is used for thread management. Implementation also uses C++ templates to automatically create several versions of each function with unrolled loops. For details see section 6.1 (PLUGIN algorithm), 6.2 (LSCV_h algorithm) and 6.3 (LSCV_H algorithm).

Experiments were performed on a computer with Intel Core i7 CPU working at 2.8GHz, NVIDIA GeForce 480GTX graphics card and 8GB 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 procesors. All implementations were run under Linux operating system (Arch Linux distribution, 3.3.7 kernel version).

All of the tested implementations may be downloaded from `http://www.cs.put.poznan.pl/wandrzejewski/_sources/hcode.zip`.

7.2 Experiments and datasets

We have 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:

- PLUGIN algorithm: $n = 1024, 2048, \ldots, 32768$, $d = 1$ (PLUGIN algorithm only supports one dimensional data).
- LSCV_h algorithm: $n = 64, 128, \ldots, 1024$, $d = 1, \ldots, 16$.
- LSCV_H algorithm: $n = 1024, 2048, \ldots, 16384$, $d = 1, \ldots, 16$.

All processing times measured for GPU implementations include data transfer times between GPU and CPU.

From the obtained processing times we have 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.

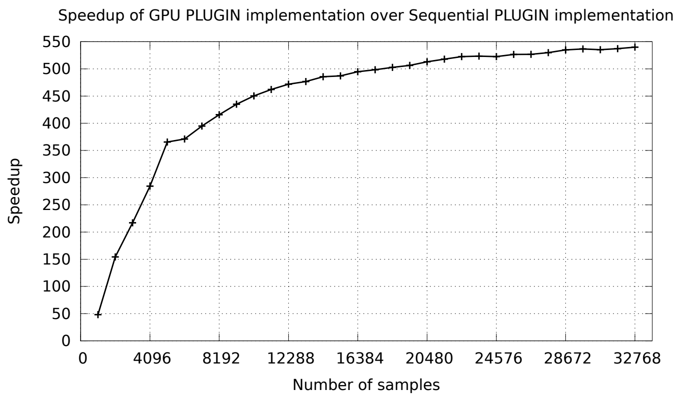
As the input data does not influence processing times of tested implementations (except maybe LSCV_H method where the performance of a minimization algorithm depends on the dataset, but in our implementation we skip this part), we supply random dataset for each algorithm/sample size/dimensionality size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

7.3 Experimental results

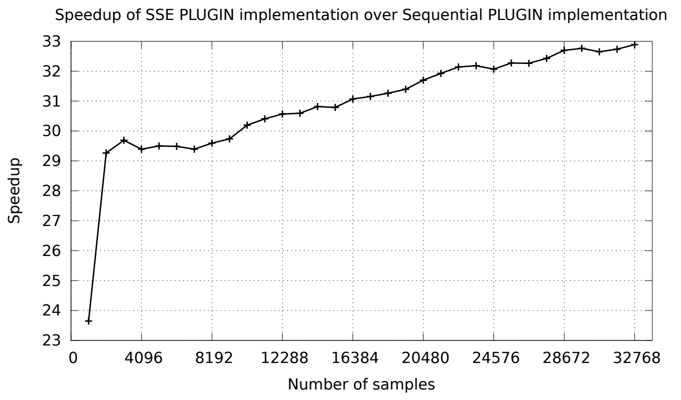
Let take a look at Figure 8a. It presents speedups of GPU PLUGIN algorithm implementation over Sequential implementation. As can be noticed, GPU implementation is about 500 times faster than sequential implementations. Moreover, notice that the bigger the instance is the greater speedup is obtained. Figure 8b presents speedups of SSE PLUGIN algorithm implementation over Sequential implementation. The obtained speedups are about 32. Notice, that similarly as with GPU implementation, the obtained speedup grows as the size of the instance increases. Figure 8c presents speedups of SSE PLUGIN algorithm implementation over Sequential implementation. The maximum obtained speedup is about 16 and the speedup grows as the size of the instance increases.

First, let us explain the observed increase in obtained speedups. PLUGIN algorithm has the complexity of $O(n^2)$ (the most complex part is computing double sums in equations (16) and (18)). Consequently, for each implementation there exists a second order polynomial which can be used to determine the time needed to process an instance of size n. Speedup is calculated by dividing the processing times of the slower implementation by the processing times achieved by the faster implementation. Let $F(n) = a_f n^2 + b_f n + c_f$ be the polynomial that determines the processing time of the faster implementation and $S(n) = a_s n^2 + b_s n + c_s$ be the polynomial that determines the processing time of the slower implementation. Speedup is calculated as follows:

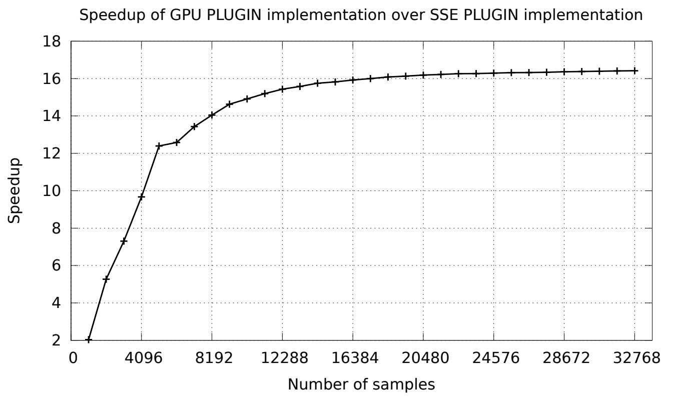
$$Speedup(n) = \frac{S(n)}{F(n)} = \frac{a_s n^2 + b_s n + c_s}{a_f n^2 + b_f n + c_f} \tag{61}$$



(a) GPU implementation speedup over Sequential implementation



(b) SSE implementation speedup over Sequential implementation



(c) GPU implementation speedup over SSE implementation

Fig. 8: Total speedups of all PLUGIN algorithm implementations

Notice that:

$$lim_{n\to 0} Speedup(n) = \frac{c_s}{c_f}, \tag{62}$$

whereas:

$$lim_{n\to\infty} Speedup(n) = \frac{a_s}{a_f}. \tag{63}$$

It can be therefore determined that speedup starts from some value $\frac{c_s}{c_f}$ and asymptotically approaches $\frac{a_s}{a_f}$ for sufficiently big instances. Here $\frac{c_s}{c_f}$ value is smaller than $\frac{a_s}{a_f}$ and consequently the speedup increases with the number of samples n. Using this framework we can also determine the maximum speedup in the given hardware and software conditions, by first fitting a second order polynomial to the obtained processing times using least-squares measurement and then calculating limit using equation (63).

The speedup values shown in Figure 8b can be explained as follows. The SSE algorithm uses SIMD instructions, performing the same operation on four different values. Moreover, a quad core processor with hyperthreading capabilities is used (sequential implementation uses only a single thread and hence only one core). Four cores times four times faster processing of data should lead to about 16 times speedup. Notice however, that due to HyperThreading capabilities, each core can process multiple threads more efficiently (we have used 8 threads - two per core). Moreover, additional cache aware optimizations make the SSE implementation even faster as little time is wasted on data retrieval from RAM.

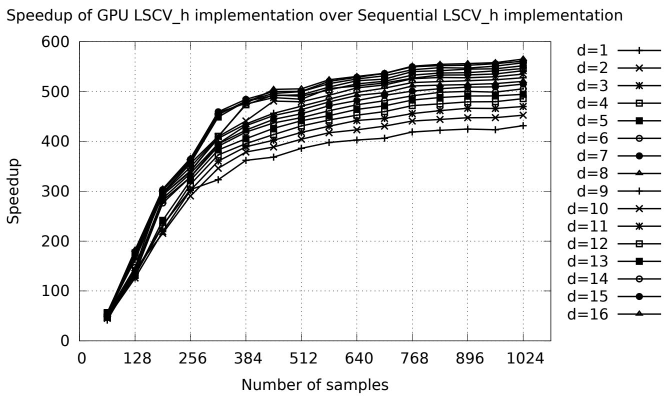
The speedup of about 16 of GPU implementation over SSE implementation and 500 over sequential is hard to explain similarly as was done for the SSE over Sequential implementation speedup as both architectures are very different.

Lets us now take a look at Figure 9. It presents speedups of GPU LSCV_h implementation and SSE implementation over Sequential implementation (Figures 9a and 9b respectively) and Speedup of GPU implementation over SSE implementation (Figure 9c). Each curve represents a different data dimensionality. Speedups achieved here are as follows:

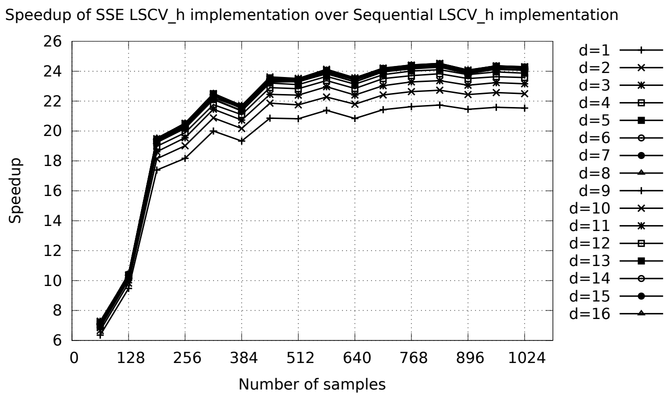
– GPU implementation is about 550 times faster than Sequential implementation,
– SSE implementation is about 20 times faster than Sequential implementation,
– GPU implementation is about 20 times faster than SSE implementation.

Observations that can be made here are very similar to the ones made for PLUGIN algorithm, as:

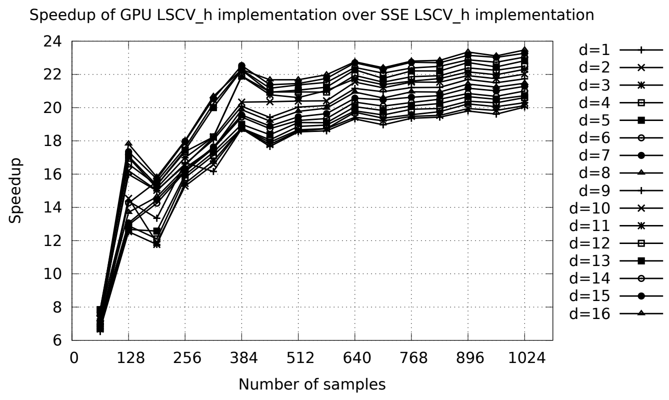
– Algorithm's complexity is $O(n^2(d^2 + n_h))$ (see section 4.5). Given the fact that d and n_h are constant, the complexity is reduced to $O(n^2)$, i.e. the same as complexity of the PLUGIN algorithm.



(a) GPU implementation speedup over Sequential implementation



(b) SSE implementation speedup over Sequential implementation



(c) GPU implementation speedup over SSE implementation

Fig. 9: Total speedups of all LSCV_h algorithm implementations

– Most of the algorithm processing time constitutes of computing of $g(h)$ function (see equation (43)) which is performed by using data reduction algorithms (the same algorithms are used in PLUGIN implementations).

One can also make several interesting observations.

First, for each dimensionality the speedup limit is slightly different. This is due to the compiler optimizations 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 bigger the speedup is, but for bigger dimensionalities the difference between them is smaller. This may be explained similarly as we have explained a similar fenomenon earlier (the one in which the speedup was increasing with respect to the number of samples n). Let us recall that LSCV_h algorithms complexity is $O(n^2(d^2 + n_h))$. For a specified constant values of n and n_h (arbitrarily set to 150), the complexity is reduced to $O(d^2)$. Consequently, processing times may be determined by a second order polynomial where dimensionality is the independent variable. As 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 at $d \to \infty$.

Second observation is that the curves in Figures 9b and 9c are not smooth. These disturbances are caused by SSE implementation which achieved for $n = 128$ and to a lesser extent $n = 384$ a little bit worse performance then expected.

Finally, let us analize Figure 10 which presents speedups of GPU LSCV_H implementation and SSE implementation over Sequential implementation (Figures 10a and 10b respectively) and speedup of GPU implementation over SSE implementation (Figure 10c). Similarly, as in LSCV_h algorithm we have tested processing times for data dimensionalities equal to $d = 1, \ldots, 16$ and computed speedups based on the obtained values. Speedups achieved here are as follows:

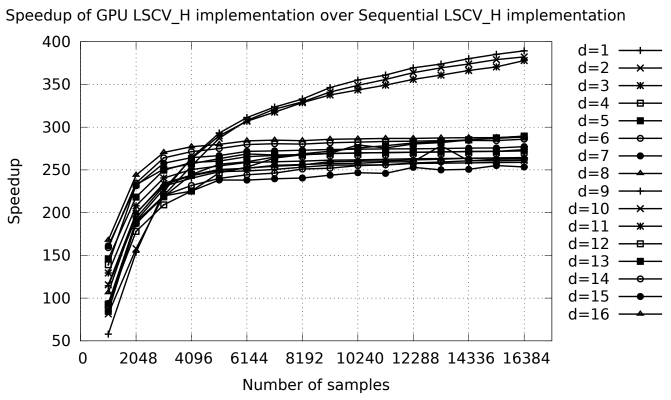
– GPU implementation is 290 times faster than Sequential implementation,
– SSE implementation is 20 times faster than Sequential implementation,
– GPU implementation is 10 times faster than SSE implementation.

Most of the discussion for PLUGIN and LSCV_h algorithms presented earlier can also be used to explain results obtained for LSCV_H algorithm implementations. Recall that our LSCV_H implementation is not complete. We have only tested processing time needed to compute $g(H)$ function (equation (32)). Let us determine the complexity of computing $g(H)$ function. The $g(H)$ function contains double sums which add $O(n^2)$ $T_H(X_i - X_j)$ function values. Computation of the $T_H(X_i - X_j)$ function requires $O(d^2)$ operations. Consequently, computation order of $g(H)$ function is $O(n^2d^2)$. This in turn leads to the earlier discussion of representing function processing times with second order polynomials with either n or d as independent variable, and the second variable constant. As was stated earlier, if processing times are expressed by second order polynomials, then speedup function (defined as ratio of two polynomials) is a function that has a limit at 0 and at ∞. All of the observed values approach some asymptotic value which is equal to the limit of the speedup function at the infinity boundary. Starting value (for low n or d values) is close to the speedup function limit at n (or d) tending to 0. Several other observations can be made. First, take a look at Figure 10a. 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 the increase in the number of samples (n). These observations however, though look differently than previously analysed on other figures, also fit our ealier discussion regarding limits of the speedup function defined as a ratio between two second order polynomials. Let us consider a situation where $n = 16384$. As n is constant, processing times can be determined by a second order polynomials dependent on d. For low values of d ($d = 1, \ldots, 3$) we observe high speedups which descrease as d increases. Such observation means that probably the constant part of the polynomial is smaller for GPU implementation than for Sequential Implementation. However, in such a case it would mean that GPU implementation requires less initialization time than Sequential version. This is certainly not true, as 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 sequential implementations outer loops are dependant 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 of those loops (branch prediction etc.). Low values of d mean that the inner loops, which compute equation (33), require less time and therefore constitute a lesser percentage of the whole algorithm processing time. The same situation does not influence GPU implementation as much, as outer loop iterations are performed in parallel. Consequently less time is wasted on n^2 dependant loop processing costs. This in turn leads to conclusion that for high n and low d 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 Figure 10a. One could ask why the same phenomenon was not observed for LSCV_h algorithm. This stems from the fact that in LSCV_h algorithm the values of equation (42) (which is equation's (33) counterpart) are computed only once, and other algorithm's tasks dominate.

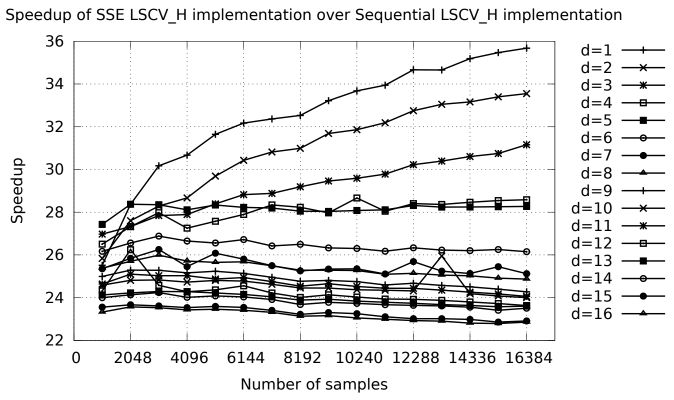
Finally, in Table 3, for illustrative purposes only, we present comparison of processing times obtained for largest instances in each of the experiments.

Table 3: Processing times for largest instances in experiments

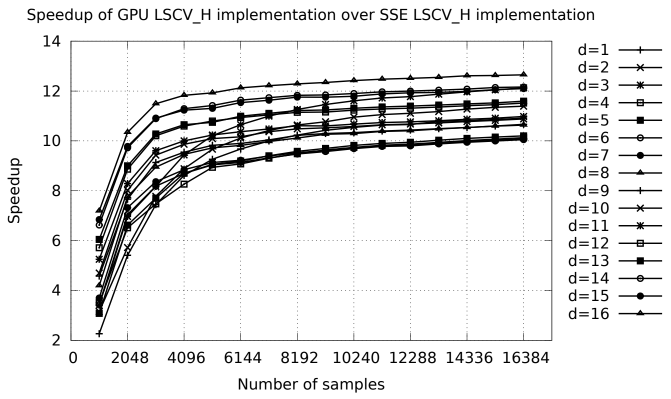
	Implementation [ms]		
Algorithm (instance size)	GPU	SSE	Sequential
PLUGIN ($n = 32768$)	87.9	1442.3	47435.3
LSCV_h ($n = 1024$, $d = 16$)	14.7	344.1	8283.6
LSCV_H ($n = 16384$, $d = 16$)	184.2	2320	53258.8



(a) GPU implementation speedup over Sequential implementation



(b) SSE implementation speedup over Sequential implementation



(c) GPU implementation speedup over SSE implementation

Fig. 10: Total speedups of all LSCV_H algorithm implementations

8 Conclusion

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 (KPDF). 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, the serious drawback of the KPDF approach is that computations of the bandwidth parameter – a crucial parameter in KPDF – are very time consuming. To solve this problem we have investigated several methods of optimizing these computations. We utilized two SIMD architectures: *SSE CPU* architecture and *NVIDIA GPU* architecture to accelerate computations needed to find the optimal value of the bandwidth parameter. We have 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 4.4. As for LSCV_h algorithm we have 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 4.5. The fast SSE and CUDA implementations are also compared with a simple sequential one. All the necessary details on using SIMD architectures for fast computation of the bandwidth parameter are presented in section 5 and the final notes on how to utilize the algorithms are presented 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. The above results confirm the great usability of modern processing units. All the codes developed have been made public available and can be downloaded from `http://www.cs.put.poznan.pl/wandrzejewski/_sources/hcode.zip`.

References

1. AMD64 architecture programmer's manual volume 6: 128-bit and 256-bit XOP, FMA4 and CVT16 instructions (2009). `http://support.amd.com/us/Embedded_TechDocs/43479.pdf`
2. AMD64 architecture programmer's manual volume 1: Application programming (2012). `http://support.amd.com/us/Processor_TechDocs/24592_APM_v1.pdf`
3. Intel® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C (2012). `http://download.intel.com/products/processor/manual/325462.pdf`
4. NVIDIA CUDA Programming Guide (2012). `http://developer.download.nvidia.com/compute/DevZone/doc`
5. Bochkanov, S., Bystritsky, V.: ALGLIB. `http://www.alglib.net`
6. Chapman, B., Jost, G., Pas, R.v.d.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press (2007)
7. Cunningham, P.: Dimension reduction. Tech. Rep. UCD-CSI-2007-7, University College Dublin (2007)

8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). DOI 10.1145/1327452.1327492. URL `http://doi.acm.org/10.1145/1327452.1327492`
9. Duong, T.: Bandwidth selectors for multivariate kernel density estimation. Ph.D. thesis, University of Western Australia, School of Mathematics and Statistics (2004)
10. Fodor, I.: A survey of dimension reduction techniques. Tech. rep., University of California, Lawrence Livermore National Laboratory (2002)
11. Garofalakis, M., Gibbons, P.: Approximate query processing: Taming the terabytes. 2001 International Conference on Very Large Databases (VLDB'2001) (2001). A Tutorial
12. Gramacki, A., Gramacki, J., Andrzejewski, W.: Probability density functions for calculating approximate aggregates. Foundations of Computing and Decision Sciences **35**(4), 223–240 (2010)
13. Greengard, L., Strain, J.: The fast gauss transform. SIAM Journal of Scientic and Statistical Computing **12**(1), 79–94 (1991)
14. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. The Morgan Kaufmann Series in Data Management Systems (2006)
15. Harris, M.: Optimizing parallel reduction in CUDA. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction`
16. Hendriks, H., Kim, P.: Consistent and efficient density estimation. In: Proceedings of the 2003 international conference on Computational Science and Its Applications- ICCSA 2003: Part I, vol. LNCS 2667, pp. 388–397. Springer (2003)
17. Higham, N.J.: The accuracy of floating point summation. SIAM J. Sci. Comput **14**, 783–799 (1993)
18. Ioannidis, Y., Poosala, V.: Histogram-based approximation of set-valued query answers. In: Proceedings of the 25th International Conference on Very Large Data Bases, pp. 174–185 (1999)
19. Jagadish, H.V., Koudas, N., Muthukrishnan, S., Poosala, V., Sercik, K.C., Suel, T.: Optimal histograms with quality guarantees. In: VLDB, pp. 275–286 (1998)
20. Johnson, N., Kotz, S., Balakrishnan, N.: Continuous Univariate Distributions, Volume 1. Probability and Statistics. John Wiley & Sons, Inc. (1994)
21. Johnson, N., Kotz, S., Balakrishnan, N.: Continuous Univariate Distributions, Volume 2. Probability and Statistics. John Wiley & Sons, Inc. (1995)
22. Kahan, W.: Pracniques: further remarks on reducing truncation errors. Commun. ACM **8**(1), 40– (1965). DOI 10.1145/363707.363723. URL `http://doi.acm.org/10.1145/363707.363723`
23. Kozielski, S., Wrembel, R. (eds.): New Trends in Data Warehousing and Data Analysis. *The Annals of Information Systems*, vol. 3. Springer Verlag (2009)
24. Li, Q., Racine, J.: Nonparametric Econometrics: Theory and Practice. Princeton University Press (2007)
25. Łukasik, S.: Parallel computing of kernel density estimates with mpi. Lecture Notes in Computer Science **4489**, 726–734 (2007)
26. Marr, D.T., Binns, F., L, D., Hill, Hinton, G., Koufaty, D.A., Miller, J.A., Upton, M.: Hyper-Threading technology architecture and microarchitecture. Intel Technology Journal **6**(1), 4–15 (2002)
27. Nelder, J.A., Mead, R.: A simplex method for function minimization. Computer Journal **7**, 308–313 (1965)
28. Raykar, V., Duraiswami, R.: Very fast optimal bandwidth selection for univariate kernel density estimation. Tech. Rep. CS-TR-4774/UMIACS-TR-2005-73, Dept. of Computer Science, University of Maryland, College Park (2006)

29. Shanmugasundaram, J., Fayyad, U., Bradley, P.: Compressed data cubes for olap aggregate query approximation on continuous dimensions. In: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 223–232 (1999)
30. Sheather, S.: Density estimation. Statistical Science **19**(4), 588–597 (2004)
31. Silverman, B.: Density Estimation For Statistics And Data Analysis. Chapman & Hall/CRC Monographs on Statistics & Applied Probability, London (1986)
32. Silverman, B.W.: Algorithm AS 176: Kernel density estimation using the fast fourier transform. Journal of Royal Statistical Society Series C: Applied statistics **31**(1), 93–99 (1982)
33. Simonoff, J.: Smoothing Methods in Statistics. Springer Series in Statistics (1996)
34. Vitter, J., Wang, M.: Approximate computation of multidimensional aggregates of sparse data using wavelets. In: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, pp. 193–204 (1999)
35. Wand, M., Jones, M.: Kernel Smoothing. Chapman & Hall/CRC Monographs on Statistics & Applied Probability, London (1995)
36. Xavier, C., Iyengar, S.: Introduction to Parallel Algorithms. Wiley Series on Parallel and Distributed Computing. Wiley (1998). URL `http://books.google.pl/books?id=W3Ld65MnwgkC`
37. Yang, C., Duraiswami, R., Gumerov, N.: Improved fast gauss transform. Tech. Rep. CS-TR-4495, Dept. of Computer Science, University of Maryland, College Park (2003)

A Derivation of equations (49) and (50)

In this appendix we provide detailed derivation of equations (49) and (50) used for calculating thread block's position within an upper triangular matrix. We assume that the thread blocks are started within a one dimensional grid and are assigned unique non negative consecutive numbers bx (see Figure 4). Based on bx we want to compute block's column (denoted l) and block's row (denoted q) within an upper triangular matrix.

We shall start with deriving the number of blocks n_b contained in triangular matrix. As each consecutive column has one more block than the previous one (i.e. column number l contains $l+1$ blocks) the number of blocks up to a column number l (i.e. $n = l + 1$ columns) may be calculated as a sum of arithmetic progression:

$$a_1 = 1, \tag{64}$$

$$a_n = l + 1, \tag{65}$$

$$n_b = n(a_1 + a_n)/2 = (l + 1)(l + 2)/2. \tag{66}$$

Let us now assume that we have a given bx number of a block and we know that this block is on the main diagonal of the triangular matrix. We will now find the column number l for this block. Let us consider a submatrix of the triangular matrix such that the bx-th block is in the lower right corner of this submatrix. As bx numbers are zero-based we know that there must be $n_b = bx + 1$ blocks in this sumbatrix. Let us substitute this to the equation (66):

$$(l+1)(l+2)/2 = bx + 1, \tag{67}$$
$$l^2 + 3l - 2bx = 0. \tag{68}$$

We solve the quadratic equation (68):

$$\Delta = \sqrt{8bx+9}, \tag{69}$$
$$l_1 = (-\sqrt{8bx+9} - 3)/2, \tag{70}$$
$$l_2 = (\sqrt{8bx+9} - 3)/2. \tag{71}$$

As the result of equation (70) is always negative the column number we are searching for is given by equation (71).

Let us now assume that bx block is not on the main diagonal. Let bx_1 be the biggest block number such that $bx_1 < bx$ and let bx_2 be the smallest block number such that $bx < bx_2$. Let l_{bx_1} be the column of bx_1 block and l_{bx_1} be the column of bx_2 block. It is easy to notic, that $l_{bx_1} + 1 = l_{bx_2}$. Consequently, the l_2 column calculated for bx block using equation (71) must be a non integer value $l_{bx_1} < l_2 < l_{bx_2}$. Moreover, as bx_1 is the number of the last block in its column, bx block must be in the same column as bx_2 block, i.e. its column number is the smallest integer value greater or equal to l_2 calculated using equation (71). This leads to the equation (49):

$$l = \left\lceil \frac{\sqrt{8bx+9} - 3}{2} \right\rceil .$$

To find block's row number q, we need to subtract from bx the number of blocks in previous columns. This can be easily found by substituting $l - 1$ into equation (66), which leads to the final equation (50):

$$q = bx - \frac{l(l+1)}{2}.$$