Lopez-Novoa U, Sáenz J, Mendiburu A, Miguel-Alonso J. A*n efficient implementation of kernel density estimation for multi-core and many-core architectures.* **The International Journal of High Performance Computing Applications.** 2015;29(3):331-347.. Copyright © The Author(s) 2015 published by Sage. https://doi.org/10.1177/109434201557681

The International Journal of High Performance Computing Applications

An Efficient Implementation of Kernel Density Estimation for Multi-core & Many-core Architectures

Abstract

Kernel Density Estimation (KDE) is a statistical technique used to estimate the probability density function of a sample set with unknown density function. It is considered as a fundamental data smoothing problem used with large datasets, and it is widely applied in areas such as climatology or biometry. Due to the large volumes of data that these problems usually process, KDE is a computationally challenging problem. Current HPC platforms with built-in accelerators have an enormous computing power, but they have to be programmed efficiently in order to take advantage of that power. We have developed a novel strategy to compute KDE using bounded kernels, trying to minimize memory accesses, and implemented it as a parallel program targeting multi-core and many-core processors. The efficiency of our code has been tested with different datasets, obtaining impressive levels of acceleration when taking as reference alternative, state-of-the-art KDE implementations.

Keywords

Kernel Density Estimation, Bounded Kernel Functions, Parallel Computing, Many-core Processors

1. Introduction

Kernel Density Estimation (KDE) is a statistical technique used to estimate the probability density function of a sample set with unknown density function. It was first introduced in the 1960s for univariate data and, due its widespread adoption, multivariate estimators appeared in subsequent years. It is considered as a

fundamental data smoothing problem and it is widely used due to its properties, in contrast to other common density estimation techniques, such as histograms [1][2].

KDE is a common tool in many research areas, used for a variety of purposes. For example, in [3] authors use density estimates to forecast weather and other factors as part of a model for optimizing maize production. In the same field, it has been applied to evaluate the signature of climate change in the frequency of weather regimes [4]. In [5], the effectiveness of a particular medical treatment is determined by means of KDE. In computer vision [6], it is applied for image segmentation and tracking. In the field of evolutionary computation, density estimation has been used to estimate a distribution of the problem variables in estimation of distribution algorithms [7][8]. An extensive list of application fields of KDE can be found in [9].

A common characteristic of the mentioned research areas is the need to deal with large volumes of data, and also the need of performing several KDE runs (varying parameters) in order to select the best estimators. For the largest tests we conducted to evaluate this work, a single run with a state-of-the-art implementation took more than 30 hours to finish. Thus, some form of fast KDE computation is required to solve in a short time these challenging problems. The way to go is the common one in high-performance computing: parallel processing.

Current high-performance systems are becoming hybrid: a combination of several multi-core CPUs and one or more accelerator devices, which can be Graphics Processing Units (GPUs), or many-core coprocessors such as Intel's Xeon Phi [10]. GPUs have a huge raw performance (several TFLOPs), but require extensive fine-tuning in order to get close to their peak power. They are usually programmed using low-level APIs such as CUDA or OpenCL, but recently annotation-based frameworks such as OpenACC [11] have eased the development of accelerator-based parallel applications. The Xeon Phi, although less powerful (in terms of peak TFLOPs: around one), is easier to program: a diversity of paradigms and tools is available for it, including those developed for state-of-the-art multi-core CPUs. In this paper, we test our KDE code in an Intel Core i7 multi-core processor, and in a many-core Intel Xeon Phi coprocessor.

This paper presents two main contributions. The first one is an analysis of the KDE problem, that results in a novel algorithmic approach that offers important savings in terms of execution times, even when implemented as a sequential program. However, given the intrinsic data-parallelism of KDE, a second contribution comes naturally: a multi-threaded parallel implementation of our proposal using OpenMP,

suitable for both multi-core and many-core processors. The use of OpenMP has enabled us to develop a program not only portable, but also performance efficient in both platforms. We have compared the execution times of our code with those of two state-of-the-art implementations of KDE, for different datasets, in order to show the excellent performance achieved. Furthermore, we have analyzed our program to study its scalability, and to identify possible ways of further improving its efficiency.

The remainder of this paper is organized as follows. We provide a description of the KDE problem in Section 2, and present a novel approach to compute it in Section 3. We describe the development environment and our KDE implementation in Section 4. The experiments carried out to assess its performance are detailed in Section 5. Finally, Section 6 draws some final conclusions and describes our future lines of work on this topic.

2. Background on KDE

The probability distribution of a random variable X is described through its Probability Density Function (PDF) f. This function f gives a natural description of the distribution of X, and allows to determine the probabilities associated with X using the relationship:

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

Given several observed data points (samples) from a random variable X, with unknown density function f, density estimation is used to create an estimated density function $\hat{f}$ from the observed data. There are two approaches to density estimation, parametric and non-parametric. The former assumes that the data is drawn from a known parametric family of distributions, for example a normal distribution. The latter does not make this assumption. The main restriction of parametric methods is that they impose restrictions on the shapes that f can have. KDE is a non-parametric approach.

One of the most common techniques for density estimation of a continuous variable is the histogram, which is a representation of the frequencies of the data over discrete intervals (bins). It is widely used due to its simplicity, but it has several shortcomings, such as the lack of continuity. The kernel density estimation (KDE) technique relies on assigning a kernel function K to each sample (observation in the dataset), and then summing all the kernels to obtain the estimate. In contrast to the histogram, KDE constructs a smooth

probability density function, which may reflect more accurately the real distribution of the data. We now describe the KDE technique in more detail.

Given a multivariate sample set $(\boldsymbol{x}_1, \boldsymbol{x}_2 ... \boldsymbol{x}_n)$ where $\boldsymbol{x}_i$ is the i-th sample value from an unknown density f, KDE builds an estimation the following way:

$$\hat{f}(\boldsymbol{x}) = \frac{1}{n} \sum_{i=1}^{n} K_H \left(\boldsymbol{x} - \boldsymbol{x}_i\right) \tag{2}$$

where:

- n denotes the cardinality of the sample set.
- $K_H(\boldsymbol{x}) = |H|^{-1/2} K(H^{-1/2}\boldsymbol{x})$
- $K(\boldsymbol{x})$ refers to the kernel function used to define the weight or influence of each sample.
- H is a dxd dimensional diagonal matrix containing the bandwidth or smoothing parameter value for each dimension.

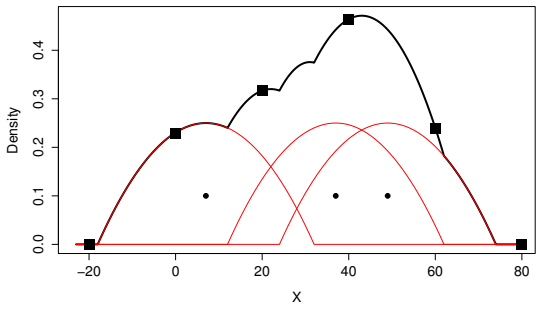


Fig. 1. Example of KDE for 1D data. Small dots represent the samples, and the red lines are the kernels around them. The estimated density is computed at the indicated points (square dots) in a discretized grid with a step of 20

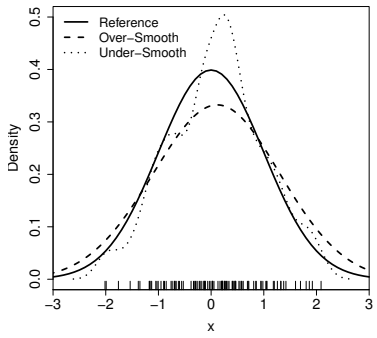


Fig. 2. Effect of the smoothing parameter when estimating data sampled from a normal distribution

Intuitively, the kernel estimator is a sum of "bumps" placed at the sample points. The kernel function K determines the shape of the bumps, while the smoothing factor h determines their width. As an example of KDE, Figure 1 depicts a simple estimator for a one dimensional space with three sample points. Each of the sample points is surrounded by a kernel depicted as a bell and the estimator is depicted as the thick line over

the kernels. Note that, even though a density estimation is a continuous function, KDE programs generate as output the values of the estimation in the discretized space requested by the user.

The bandwidth parameter h controls the influence area and smoothness of kernels. It is used to reduce the noise in the density estimation, and it must be carefully selected. To describe its effect, we depict in Figure 2 a simulated random sample from the Gaussian distribution in 1D, and different estimations of the density derived from those samples. The solid curve depicts the real density of the data, while the other lines depict (kernel-based) density estimations using different values of h. The dashed line corresponds to a large h, and results in an over-smoothed estimator. In contrast, the dotted line corresponds to a small h and results in an exceedingly sharp estimator. None of the cases reflect accurately the actual density of the samples. Several techniques to aid in the selection of the optimum bandwidth are detailed elsewhere [1] [12].

A kernel function K is a symmetric but not necessarily positive function that integrates to one. Kernels can be classified into two groups: bounded and unbounded, depending on their area of influence. Two widely used kernel functions are Epanechnikov (which is bounded) and Gaussian (which is unbounded). They are defined in Equations 3 and 4 respectively. In Equation 3, c_d is the volume of the unit d-dimensional sphere: $c_1 = 2, c_2 = \pi, c_3 = 4\pi/3, ...$ For the sake of clarity, we have depicted in Figures 3 and 4 the Epanechnikov and Gaussian kernels for 1D spaces. Note how bounded kernels only affect those points in the space at a limited distance (1 in the figure), while the influence area of unbounded kernels spans infinitely in both directions.

$$K(\boldsymbol{x}) = \begin{cases} 1/2c_d^{-1}(d+2)(1-\boldsymbol{x}^T\boldsymbol{x}) & \text{if } \boldsymbol{x}^T\boldsymbol{x} < 1 \\ 0 & otherwise \end{cases} \tag{3}$$

$$K(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^d}} exp\left(-\frac{1}{2}\boldsymbol{x}^T\boldsymbol{x}\right) \tag{4}$$

The choice of the particular kernel to apply is up to the KDE user, taking into account the problem at hand. According to Silverman [1], asymptotically, there are no differences between the different kernels at hand. Moreover, he states that it is desirable to base the choice of the kernel on other considerations, such as the degree of differentiability required or the computational effort involved. In particular, he describes how,

when the right value of h is chosen, almost all the popular kernels produce a relatively small mean integrated square error in the estimation.

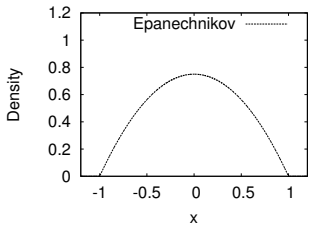


Fig. 3. Shape of a Epanechnikov kernel in 1D

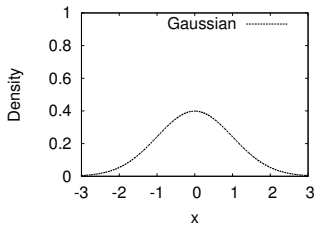


Fig. 4. Shape of a Gaussian Kernel in 1D

When dealing with multivariate scenarios, considering all dimensions as if they were on the same scale could lead to poor estimates, particularly when the data has very different variability in each dimension [13]. To tackle this issue, Fukunaga [14] suggested to (1) transform the data to have unit covariance matrix, (2) smooth it using a radially symmetric kernel and (3) transform it back. The idea is to adapt the shape of the kernel to the distribution of the data, as depicted in Figure 5 for a 2D space, where the dashed circle represents the original kernel and the ellipse is the transformed one. The distribution of the data is represented by the sample covariance matrix $\mathbf{\Sigma}$, which contains the variances and covariances between the d sample vectors of the dataset. If the original $\mathbf{\Sigma}$ matrix is unknown, an estimated $\hat{\mathbf{\Sigma}}$ must be created. Then, following [14], Equation 2 is adapted this way:

$$\hat{f}(\boldsymbol{x}) = \frac{|\hat{\mathbf{\Sigma}}|^{-1/2}}{nh^d} \sum_{i=1}^{n} K\left(h^{-2}(\boldsymbol{x}-\boldsymbol{x}_i)^T \hat{\mathbf{\Sigma}}^{-1}(\boldsymbol{x}-\boldsymbol{x}_i)\right) \tag{5}$$

This transformation is called whitening the data and the resulting effect is that the obtained estimator is more accurate, as the kernel is adapted to the distribution of the data. This approach also allows us to use a single, scalar h value for all the dimensions.

3. An efficient algorithm to compute KDE

KDE generates an estimation of the probability density function of a sample set. This estimation is computed in a user defined evaluation space (or evaluation grid) that should include all the observation points in the

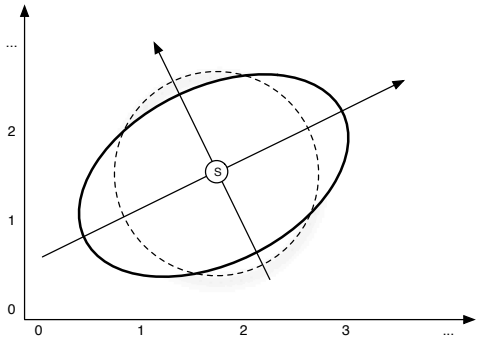


Fig. 5. Example of an original and its corresponding whitened radial symmetric kernel in 2D, taken from [14]

sample set. In the example of Figure 1, the evaluation space spans from -20 to 80, covering the three samples plus some “extra” space at the boundaries.

KDE implementations discretize this space, computing the density in equally separated grid points. This way, the evaluation space can be represented as a multi-dimensional matrix. The separation between grid points is defined by a user-provided evaluation step or grid step, that can be different in each dimension. In Figure 1, the density of the space will be computed for six evaluation points (-20, 0, 20, 40, 60 and 80), which are equally separated at distance 20.

The accuracy of the estimation generated by a KDE program, thus, depends on several parameters that must be provided by the user:

1. The choice of kernel function
2. The bandwidth parameter
3. The evaluation space, determined by a vector of bounds
4. The grid step vector

Most programs offer default values for these parameters, or heuristics to compute them. In the following sections we assume that these parameters are known, that the input of the KDE program is a dataset containing the samples, and that the output is a multi-dimensional matrix with the density estimation at all the evaluation points within the discretized evaluation space defined by the user.

3.1. Methods to compute KDE

KDE is commonly computed as described in Algorithm 1. For each evaluation point, the combined density that the samples generate on it is computed, which depends on the kernel of choice. This computation requires measuring the distance between the evaluation point and each sample. The combined density at the evaluation point is the sum of all the partial densities. We call this approach evaluation point-wise KDE or EP-KDE for short.

Algorithm 1 Evaluation point-wise KDE (EP-KDE)

```
for each Evaluation Point e do
  value(e) = 0
  for each Sample s do
    dist = computeDistance(e,s)
    value(e) += computeDensity(dist)
  end for
end for
```

The computational complexity of EP-KDE is $O(k_d mn)$, where k_d is a constant related to the dimensionality of the dataset, m is the number of evaluation points, and n the number of samples. Note that m is proportional to the size of the evaluation space and the grid step. For large, multi-dimensional spaces or/and tight grid steps, m can be huge.

EP-KDE is directly parallelizable in a data-parallel fashion: the computations affecting a given evaluation point are independent from those affecting other points. If a thread is in charge of computing the density in an evaluation point, all the threads can progress concurrently without any sort of memory-write contention.

The EP-KDE approach is valid for both unbounded (e.g. Gaussian) and bounded (e.g. Epanechnikov) kernels. In the first case, all the samples affect all the evaluation points. In contrast, with bounded kernels, samples only contribute to the density in those evaluation points within its influence area. Therefore, using EP-KDE, in most cases the computeDistance function of Algorithm 1 will return a value outside the bounds of the kernel and, therefore, function computeDensity will return 0.

As EP-KDE with bounded kernels can lead to a huge amount of worthless computations (that would depend on the size of evaluation space and on the dispersion of the samples), we have developed a more efficient algorithm for this group of bounded kernels. It is described in Algorithm 2, and we call it sample-wise KDE or S-KDE for short. Instead of focusing one by one on each evaluation point and the influences of all samples over it, S-KDE focuses one by one on each sample, computing its influence on the evaluation points surrounding it. As the kernel is bounded, the area of influence is confined within a bounding box, which is computed in advance. This bounding box is a hyperrectangle whose dimensions are determined by the maximum per-dimension distances of influence of the kernel. Note that this is kernel-dependent, but not sample-dependent. Thus, the size and shape of the bounding box $\boldsymbol{b}$ is computed just once. Then, for each sample the bounding box must be aligned to the evaluation grid, defining the per-sample bounding box $\boldsymbol{b_s}$.

Algorithm 2 Sample-wise KDE (S-KDE)

```
for each Evaluation Point e do
  value(e) = 0
end for
b = computeBoundingBox
for each Sample s do
  b_s = adjustBoundingBox(b,s)
  for each Evaluation Point e in b_s do
    dist = computeDistance(e,s)
    value(e) += computeDensity(dist)
  end for
end for
```

With S-KDE, computations of distance and summations of influences are greatly reduced: for each sample most evaluation points fall outside the influence area of the kernel, and are not considered in subsequent computations. The complexity of S-KDE is $O(k_d np)$, where k_d is a constant related to the dimensionality of the dataset, n is the number of samples and p is the size of the bounding box, which is expected to be much smaller than the total number of evaluation points m.

S-KDE is also naturally data-parallel, using a per-sample approach. However, two concurrent threads computing the partial densities of two different samples with overlapping influence areas can incur into memory-write contention. Therefore, a mechanism to coordinate memory accesses must be put in place.

3.2. Computing the bounding box

An efficient implementation of S-KDE requires the computation of the bounding boxes, which surround the area of influence of each sample. In this section we describe how b can be computed, assuming that data is transformed using the Fukunaga approach described in Section 2. Our proposal is based on a method described in [14]. The area of influence of a sample using the (transformed) kernel has an elliptic shape in 2D spaces as depicted in Figure 5, and a d-dimensional ellipsoid shape for spaces of higher dimensions. The challenge is to find a box bounding the ellipsoid, and to align it to the discretized grid that defines the evaluation space. In Figure 6 we depict an example: a sample s in position (6.75, 4.5), its area of influence (the ellipse), and the bounding, aligned rectangle (dashed line).

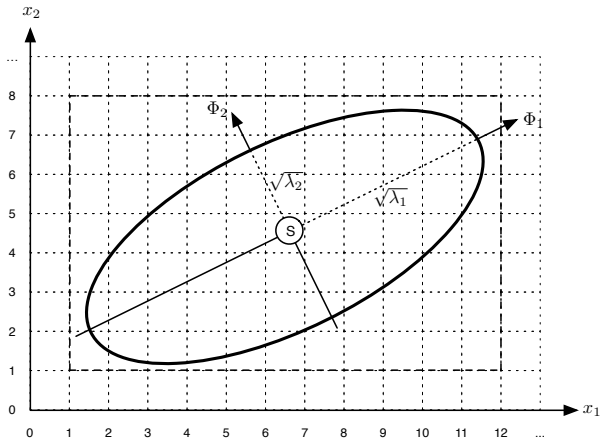


Fig. 6. Area of influence of a sample and the corresponding bounding box

The starting point of the calculation is the covariance matrix of the sample set. It contains the information to determine the shape and size of the ellipsoid. In addition, the smoothing parameter h, which modifies the size of the kernel, must be taken into account. We first compute $\mathbf{\Sigma}' = \mathbf{\Sigma}h^2$, where $\mathbf{\Sigma}'$ is the modified covariance matrix with the applied bandwidth value. Then, the semi-axes of the ellipsoid are computed as the eigenvalues of $\mathbf{\Sigma}'$, and its orientations as the eigenvectors of that matrix. After this, the next step is to compute:

$$\Theta = \sqrt{\lambda} I \tag{6}$$

where Θ is a diagonal matrix containing the square root values of the $\boldsymbol{\lambda}$ eigenvalues vector. I represents the identity matrix. The square root of each eigenvalue gives the length of each semi-axis from the center of the ellipsoid, but aligned to the coordinates axis. Thus, the matrix Θ of eigenvalues must be multiplied by the Φ eigenvectors matrix to transform the axes of the ellipsoid (to "rotate" them):

$$\boldsymbol{Y} = \Phi\Theta \tag{7}$$

The result is a matrix $\boldsymbol{Y} = \left[\boldsymbol{y}_1^T \boldsymbol{y}_2^T ... \boldsymbol{y}_d^T\right]$ where each $\boldsymbol{y}_i^T$ column contains the coordinates of the semi-axes of the ellipsoid.

From this, a vector of distances $\boldsymbol{b}$ is obtained which determines, per direction, the absolute value of the maximum distance at which evaluation points are influenced by the kernel:

$$b_i = ||\boldsymbol{y}_i|| \tag{8}$$

where $||\cdot||$ is the Euclidean norm operator. In other words, each element of vector $\boldsymbol{b}$ defines half the length of each edge of the hyperrectangle surrounding the kernel. Once $\boldsymbol{b}$ is known (note that this computation is done just once), the actual, aligned per-sample bounding box $\boldsymbol{b_s}$ can be computed, adding and subtracting $\boldsymbol{b}$ from/to the absolute coordinates of the sample, and rounding up the resulting values.

3.3. From d-dimensional bounding boxes to 2D bounding rectangles

Using the process described in the previous section, we fit a d-dimensional hyperrectangle-shaped box around the ellipsoid-shaped influence area of a sample, as determined by the kernel function. The evaluation points influenced by the sample are only those within the ellipsoid, which means that the box contains evaluation points not belonging to the influence area of the sample. In fact, depending on the shape and angle of the ellipsoid, the evaluation points outside the ellipsoid could be more numerous than those inside it.

Motivated by this fact, we define a way to make a tighter delimitation of bounding boxes. The first idea is to reduce the dimensionality of the problem, splitting (chopping) the original d-dimensional box into several non-overlapping $d-1$-dimensional boxes. Each of them is again chopped into $d-2$-dimensional boxes, repeating the process until a collection of 2D rectangles (slices) is obtained. Then, each rectangle is cropped

until reducing it to a smaller one minimizing the number of evaluation points not influenced by the sample. This two-step procedure is represented in Figures 7 and 8.

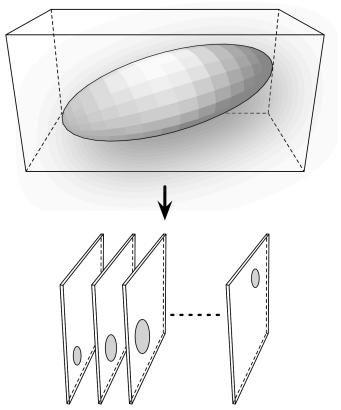


Fig. 7. Chopping a 3D bounding box into 2D slices

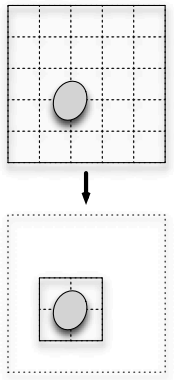


Fig. 8. Cropping a 2D slice to obtain a minimum-size bounding rectangle

The number of slices per bounding box will be determined by the size of the evaluation space and the grid step. Without loss of generality, we will use a three dimensional space as example. The chopping is performed along the third dimension of the space, which corresponds to the z axis in the cartesian coordinate space. E.g. if a bounding box is located between coordinates 8 and 12 of z axis, and the step in the z axis is 0.5, there would be 9 slices. Once 2D slices are obtained, the cropping is performed to reduce the 2D bounding box to make it surround tightly the ellipse. This process involves several steps that are described below and in Figure 9. A detailed description of this technique is carried out in Appendix A.

1. Calculate the rotation angle of the ellipse in the slice.
2. Calculate the length of the principal axes of the ellipse.
3. Using the rotation and the length of the principal axes, calculate the coordinates of the edge vertices of the ellipse.
4. Using the coordinates of the edge vertices, calculate the boundaries of the ellipse. Then, align boundaries to the evaluation grid and calculate coordinates for the minimal bounding box.

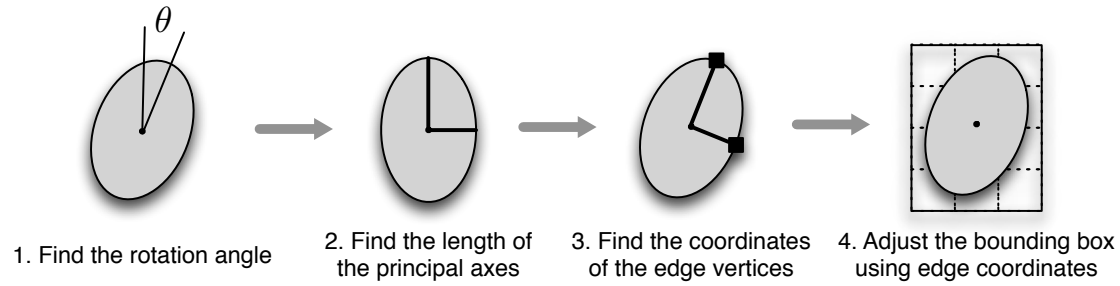


Fig. 9. Steps to perform cropping of a 2D bounding box

4. Parallel implementation of S-KDE

The S-KDE algorithm described in Section 3 has a much lower computational complexity than the commonly used EP-KDE algorithm. However, it can be computationally challenging when dealing with large datasets and/or large, dense evaluation spaces. Thus, we have implemented it from the beginning as a data-parallel program, targeting multi-core and many-core processors.

4.1. Target hardware

The current landscape of HPC systems presents a wide variety of compute devices for the execution of massively parallel codes, from multi-core processors to accelerators with hundreds of processing units, such as GPUs. For our work we have chosen multi-core processors, due to their widespread presence, and many-core coprocessors, due to their growing popularity in the HPC community and their architectural similarity with multi-core processors (which simplifies code portability).

In particular, we have used an Intel Sandy Bridge-E multi-core processor and an Intel Xeon Phi coprocessor in the tests reported in this work. Both share the Intel 64-bit x86 architecture and Intel development tools (compiler suites, VTune Amplifier, ...), but hold strong architectural differences: e.g. Sandy Bridge-E processors hold at most 8 physical cores with 256-bit wide vector units supporting SSE and AVX instruction sets. Xeon Phi coprocessors hold up to 61 computing cores with 512-bit wide vector units and a different vector instruction set, AVX-512. Furthermore, Sandy Bridge-E processors have out-of-order execution capable cores and hold up to 20 MB of shared L3 cache, while Xeon Phi coprocessors are designed for in-order execution, and share up to 30 MB of L2 cache among all their cores. More detailed descriptions of

the hardware architecture of these devices can be found in [10], and a discussion about the programming difficulties of these devices in [15].

4.2. Implementation details

We have developed our code in ANSI-C, and parallelization is achieved making use of OpenMP and Intel compiler directives. In particular, we use OpenMP directives to define how tasks are mapped to threads, and the #pragma simd directive to vectorize parts of the code. In addition, we use the Meschach math library [16] for some matrix operations, such as the computation of eigenvalues and eigenvectors. All computations are performed in double precision.

The workflow of our S-KDE code is depicted in Figure 10. Each thread is in charge of computing the influence of a set of samples over the evaluation space. For each sample, the thread first adjusts its bounding box (as defined in Section 3.2). If the evaluation space is of dimensionality three or higher, the chop-and-crop procedure described in Section 3.3 is recursively applied to reduce the computation to 2D slices. Then, for each 2D bounding rectangle the density that the sample creates in each evaluation point is computed.

One of the drawbacks of performing sample-wise computations is the memory contention that may appear when two or more different threads, managing samples whose influence area overlap, have to add partial density values into the memory position that represent the same evaluation point. To reduce this harmful effects, each thread calculates every row of the slice in its private memory, and adds it into main memory using the atomic OpenMP pragma. This way, we ensure data write consistency. There is a cost to pay, though: atomic operations causes overheads due to the serialization of memory write operations. We analyze this overhead in detail in Section 5.

Regarding the Xeon Phi, one of our target platforms, it is worth mentioning that this device has two working modes: offload and native. In the former, all the code runs on a host processor and just some parts are offloaded to the coprocessor. In the latter, the whole code runs in the Xeon Phi, which behaves as a separate computer. In our tests we have used the native mode, which requires cross-compilation, in the host computer, of the code and the companion libraries.

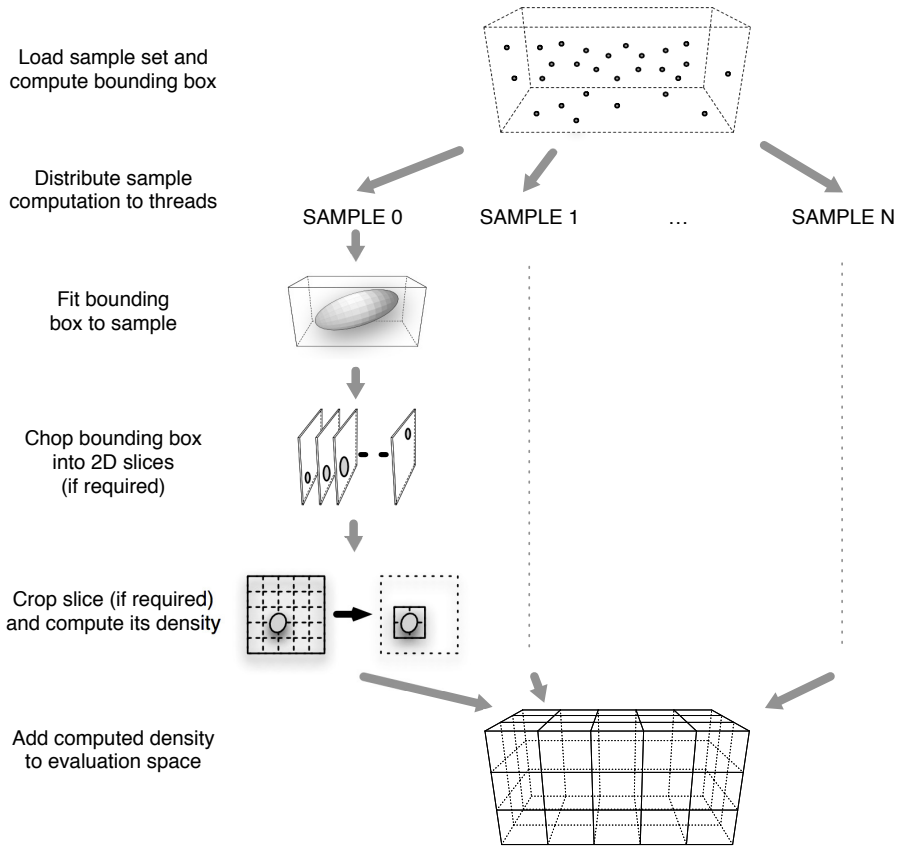


Fig. 10. Workflow of our S-KDE implementation

4.3. Other KDE implementations

We have found an extensive body of literature on KDE implementations, general or targeting specific computing platforms. We review this literature and describe it classified according to the algorithmic approach used to compute KDE, with focus on parallel implementations.

4.3.1. Evaluation-point wise implementations

Initial parallel KDE implementations were based on MPI, distributing the workload among several computing nodes of a multicomputer. One of these implementations was presented in 2002 by Racine [17]. A more complete version was introduced in 2007 by Lukasik [18], introducing approaches for load balancing and parallel techniques to compute some of the KDE parameters.

More recent GPU-based implementations of KDE are also available. We have identified two CUDA implementations of KDE, one by Srinivasan et al. [19] in 2010 and another one by Michailidis et al. [20] in 2013. As the code of the latter was not publicly available, we will focus on the former, called GPUML. It offers an interface for C/C++ and Matlab that provides CUDA implementations of several weighted kernel summation and matrix construction algorithms. It provides support for several kernel functions (including Gaussian and Epanechnikov). To speed-up the computations, the program implements data-reuse mechanisms on the shared memory of the GPU. In Section 5 we will compare the execution times of S-KDE and GPUML using in both cases Epanechnikov kernels.

We also tried to compare our code with some KDE implementations for the widely used Python environment, but they were not capable of running our experimental tests. Function Gaussian_KDE from SciPy [21] package seemed the most popular choice among Python practitioners, but it was not capable of loading our sample datasets, raising memory overflow errors. Function KDEMultivariate from StatsModel [22] successfully loaded our datasets, but its performance was exceedingly poor. We linked our Python environment with the highly optimized Math Kernel Library [1](MKL) by Intel. MKL provides vectorized and multi-threaded versions of math routines, such as those in BLAS or LAPACK. Unfortunately, StatsModel seemed to take no benefit from these optimized math functions, and worked in a single-core fashion. In addition, due to its EP-KDE approach, its computational complexity is much higher than that of our S-KDE code. Thus, our experiments with KDEMultivariate were unfeasible in computation time, and we left them out of this work.

4.3.2. Sample wise implementations

Within the popular R software environment, package ks [23] includes a kde function whose implementation holds similarities with our S-KDE: it performs computations in a sample-wise fashion, but using a Gaussian kernel. A kind-of bounding box around the Gaussian kernel is defined, using a parameter to limit its influence area. The default value for the threshold preserves at least the 99% of the Gaussian kernel influence.

This approach makes feasible to apply the S-KDE approach but, at the expense of loosing precision in the influence area of the kernel. Our choice was to use the Epanechnikov kernel to make the computations

[1] http://software.intel.com/intel-mkl

avoiding a potential loose of precision. In addition, our chop-and-crop technique allows us to work with large, multi-dimensional datasets, whose processing would be unfeasible otherwise.

As an additional feature, kde in ks includes a mode to compute the density estimation in a list of points provided by the user instead of a complete evaluation grid. This mode allows to perform KDE in spaces of dimensionality greater than three without excessive execution times, if the list not too long.

As we did with Python, we linked the R environment with Intel MKL. In this set-up, we verified that ks-kde effectively took advantage of multi-core CPUs for matrix operations. We include in Section 5 a comparison of S-KDE against R's ks-kde. Regarding the kernel influence threshold, we have used the default value.

5. Evaluation of S-KDE

In this section, we describe the experiments we made to analyze the performance of our S-KDE implementation under different configurations, comparing the execution times with those of an evaluation-point wise version of KDE (EP-KDE), R's ks-kde and GPUML. We also identify possible ways of improving S-KDE performance even further.

5.1. Testing environment

As detailed in Section 4.1, we have tested our S-KDE code in two different hardware platforms, a multi-core processor and a many-core coprocessor. In particular have used an Intel Core i7 3820 CPU and an Intel Xeon Phi 3120A coprocessor. We have also used a NVIDIA GTX 650 GPU for some comparison tests involving GPUML. The main features of these processors are summarized in Table 1.

	Core i7 3820	Xeon Phi 3120A	GTX 650
Architecture	Sandy Bridge-E	MIC	Kepler
Number of cores	4	57	384
Number of threads	8	228	384
Clock speed	3.6 Ghz	1.1 Ghz	1.05 Ghz
Vector width	256 bit	512 bit	1024 bit
Main memory	8 GB DDR3	6 GB GDDR5	1 GB GDDR5

Table 1: Hardware features of the processors used in to run KDE. Note that the RAM in the i7 is the one installed in the computer powered by that processor, while the other reported values are those of the accelerator built-in memory

As explained before, the complexity (or problem size, which determines the execution time) of S-KDE depends on two factors: the number of samples in the dataset and the number of evaluation points in each per-sample bounding box. The latter depends on three parameters: the dimensionality of the problem, the distance between the evaluation points (or per-dimension evaluation step) and the bandwidth.

In this work, we have performed several tests of KDE codes varying the size of the evaluation space for four different datasets (two 2-dimensional and two 3-dimensional). We have fixed the boundaries of the evaluation space, but the per-dimension evaluation step has been modified in order to increase or reduce the problem size.

For 2D tests we have used datasets from an actual, real word problem: DNA sequencing. The datasets were generated by IonTorrent sequencing machines [24], which monitor millions of simultaneous sequencing reactions, and additionally produce reports with the collected information. The IonTorrent community makes publicly available some sample datasets corresponding to their AmpliSeq Cancer sequencing machines, along with their corresponding sample reports. In particular, a part of the generated reports is the density map of the chip, and in the tests reported in this paper we have reproduced that computation. More information about the Ion chips and the available datasets can be found at the Ion Community web site [2]. We have used the datasets reported for the Ion 316 and Ion 318 chips, which contain 3473932 and 5885326 samples each. The size of the evaluation spaces used in the tests (not in terms of actual, physical sizes but in terms of the number of points on which density is estimated) are listed in Tables 2 and 3. Note that more points means tighter grids.

Dim X	Dim Y	Total
560	540	302400
560	1350	756000
1400	1350	1890000
1400	2700	3780000
2800	2700	7560000

Table 2: Size of the evaluation spaces (number of grid points) used with the Ion 316 chip dataset

Dim X	Dim Y	Total
760	680	516800
760	1700	1292000
1900	1700	3230000
1900	3400	6460000
3800	3400	12920000

Table 3: Size of the evaluation spaces (number of grid points) used with the Ion 318 chip dataset

[2] http://ioncommunity.lifetechnologies.com

For the 3D tests we created two synthetic datasets using mvrnorm function from MASS library within the R framework [25]. These datasets have 500000 and 1000000 samples each. As we did with the 2D tests, we fixed the boundaries but modified the steps in order to compute problems of different sizes, working with lighter or denser evaluation spaces. The actual numbers of evaluation points per dimension for each test are listed in Table 4.

Dim X	Dim Y	Dim Z	Total
110	220	322	7792400
110	440	322	15584800
220	440	322	31169600
220	440	805	77924000
220	1100	805	194810000

Table 4: Size of the evaluation spaces (number of grid points) used with the 3D datasets

In all these tests, the bandwidth value is a relevant parameter to be taken into account. It modifies the shape and spread of the kernel, that is its influence area, which impacts the number of computations to be performed per sample. In this work we use a heuristic detailed in [1] to compute an appropriate bandwidth for a given dataset:

$$h = A(K)n^{-1/(d+4)} \tag{9}$$

where n is the number of observed samples and d the number of dimensions. $A(K)$ is a constant calculated as:

$$A(K) = \left\{8c_d^{-1}(d+4)(2\sqrt{\pi})^d\right\}^{1/(d+4)} \tag{10}$$

where c_d is the volume of the unit d-dimensional sphere (e.g. π for 2D or $4\pi/3$ for 3D).

Regarding compilation details, S-KDE was compiled for both the Core i7 CPU and the Xeon Phi coprocessor using the Intel C Compiler version 14.0.1 with -O2 optimization. The cross-compilation for the Xeon Phi required the -mmic flag.

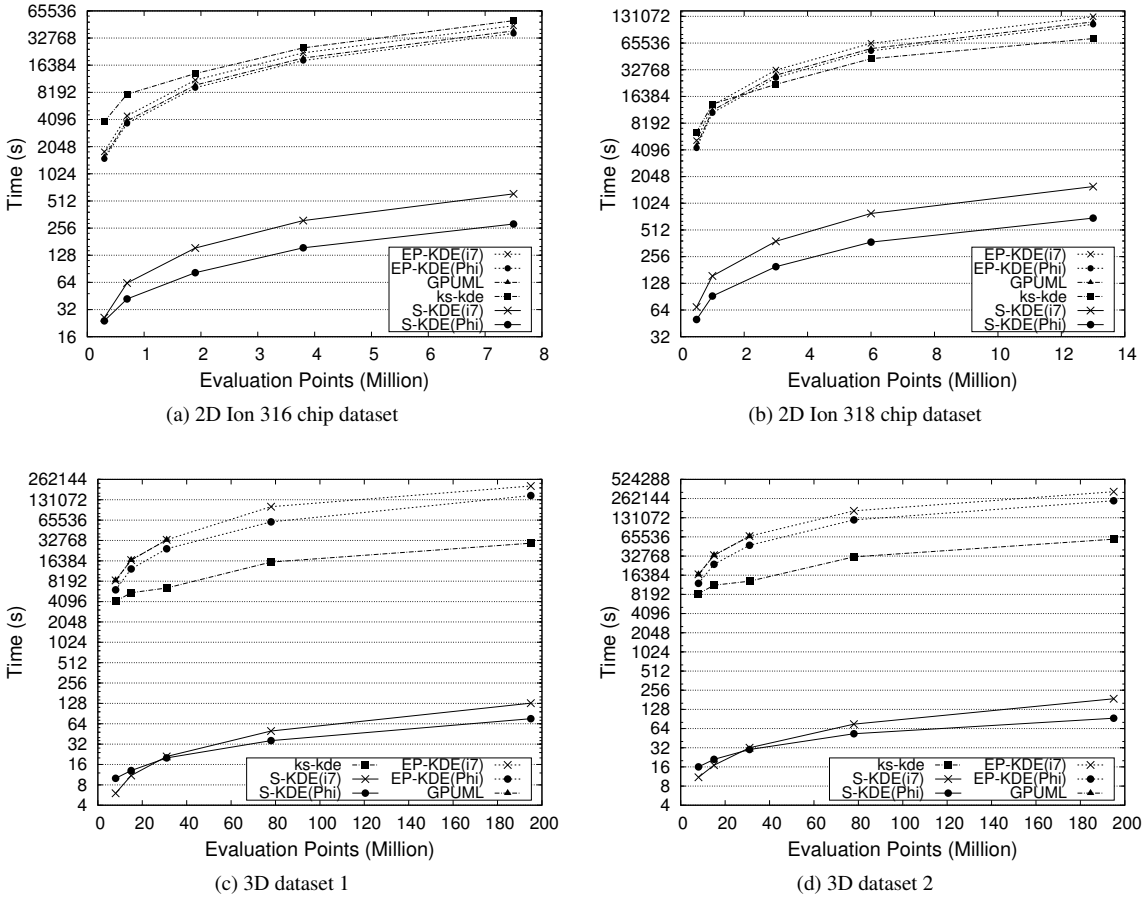
5.2. Measuring the performance of S-KDE

In this section we report the results of some tests carried out to evaluate the performance of our S-KDE parallel implementation. In order to provide a meaningful comparison, we developed a parallel implementation of an evaluation point-wise approach (EP-KDE onwards), also using OpenMP. We will compare S-KDE with R's ks-kde (supported by MKL) and GPUML too. Our purpose is to confirm that our approach represents a significant improvement over state-of-the-art KDE implementations. Results confirm this fact: our code is, by far, the fastest one.

As we are testing different programs, that compute KDE using different approaches and with different kernel functions, we need to set a "fair" comparison basis. We have tried to run the programs in such a way that they produce the same or very similar results. To measure this similarity we have used the score function described by Perkins et al. [26], which returns the similitude value between two different density functions. Note that two different algorithm-kernel combinations may require different values of bandwidth in order to generate the same density estimation.

Our comparison procedure is as follows. We first compute the density estimation (DE) for a given dataset and evaluation space with our code, using the bandwidth value provided by the heuristic described above (Section 5.1), and measure the execution time. Then we run a competitor program (for example, GPUML) with the same dataset and evaluation space, but varying the bandwidth value until the resulting DE reaches a similitude score over 98%. The execution time of the run passing this similitude threshold is the one assigned to the competitor program. In the actual tests, the bandwidth values used with ks-kde resulted in a 98.2% similitude on average; for GPUML, similitude was 99.1% on average.

In the test, S-KDE and EP-KDE were executed in a multi-core i7 processor and in a Xeon Phi coprocessor. The ks-kde was executed in the multi-core processor (taking advantage of the parallel processing capabilities of MKL), and GPUML in a GTX 650 GPU. Each program-platform combination was used with the four datasets, varying the problem size (using different steps in the evaluation space). All the measured execution times are summarized in Figures 11a and 11b for the 2D IonTorrent datasets, and in Figures 11c and 11d for the 3D synthetic datasets. Note that GPUML data for the 3D experiments with large evaluation spaces are not reported because they could not be executed, due to limitations in the memory size of our GTX 650 GPU.



(a) 2D Ion 316 chip dataset

(b) 2D Ion 318 chip dataset

(c) 3D dataset 1

(d) 3D dataset 2

Fig. 11. Execution times of the KDE implementations under test, for different datasets

The first conclusion is that, in all the tested cases, S-KDE is significantly faster than the competitors. We show in Table 5 the speed-up of S-KDE running in both the Core i7 and the Xeon Phi compared to ks-kde, GPUML and EP-KDE.

The long execution times of ks-kde can be easily explained considering that (1) R is an interpreted environment and, therefore, R programs have higher runtime overheads than compiled programs; (2) the bounding boxes used in ks-kde to prune Gaussian kernels are larger than those defined by Epanechnikov kernels and, thus, require higher computational effort; and (3) ks-kde does not implement the chop-and-crop technique to reduce computations in problems of dimensionality three and higher. The performance improvement derived from this feature of S-KDE will be analyzed in detail later, in Section 5.3.

	2D S-KDE (i7)	2D S-KDE (Phi)	3D S-KDE (i7)	3D S-KDE (Phi)
EP-KDE (i7)	76.4	138.5	1759.1	1797.9
EP-KDE (Phi)	62.9	114.0	1241.9	1257.5
GPUML	66.9	121.4	1628.3	1397.1
ks-kde	85.8	145.5	443.5	463.8

Table 5: Average speed-up of S-KDE (in i7 and Xeon Phi) compared to state-of-the-art KDE implementations

Regarding GPUML and EP-KDE, the large execution times are a consequence of the evaluation-point-wise algorithm implemented: the whole evaluation space is swept, point by point. For each evaluation point, the influence that each and every sample has on it is computed—even when, in many cases, this influence is zero because the used kernel is bounded and the point falls outside the influence area of the sample. In addition, GPUML implementation suffers from overheads derived from the use of a GPU-based discrete accelerator, being the main one the cost of transferring input data and results from/to main memory to/from GPU memory (through a PCIe connection). It is also worth mentioning that the size of the GPU memory imposes limits on the size of the problems: note that our GPU has only 1 GB of memory to store the d-dimensional matrix representing the evaluation space plus other data structures, while the i7 has 8 GB and the Xeon Phi has 6 GB.

5.3. Assessing the benefits of parallel computing and the chop-and-crop technique

The analysis in the previous section presents a high-level, black box evaluation of the performance of S-KDE. In this section we explore further into the code, in order to better understand where the performance gains are coming from. There are three basic elements contributing to a fast KDE program: (1) the sample-wise density estimation approach, (2) the chop-and-crop technique and (3) the use of parallel processors. When needed, we compare our code with ks-kde which also uses bounding boxes and, through MKL, can take advantage of parallel hardware—but does not implement chop-and-crop. Also, we focus on a particular dataset: the 3D dataset 1. This is just to simplify tables and graphs. Results for other datasets show the same trends.

5.3.1. Benefits of parallel processing

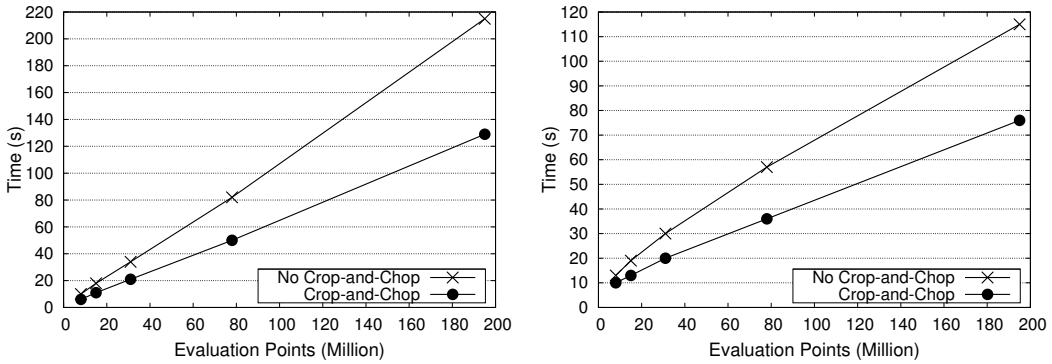
S-KDE can run as sequential code, thus allowing us to evaluate the "algorithmic" performance of our sample-wise approach combined with chop-and-crop. We did some tests running S-KDE with a single thread (that is, in a single core) in the 4-core i7 platform, and found that execution time is, on average 4.28 times longer. This is for 3D dataset 1, but the experiments for the remaining datasets report similar slow-down values. Combining this results with those in Table 5, we find that the sequential S-KDE is still between 20 and 100 times faster than ks-kde running in the i7. However, as both S-KDE and ks-kde have been measured in the same i7 and, through the MKL libraries, ks-kde can take advantage of the underlying parallel hardware, we consider that the reported comparisons yielding 85.8 and 443.5 speedups for S-KDE are fair. The Xeon Phi accelerator can add some extra performance when using all its cores simultaneously.

5.3.2. Benefits of chop-and-crop

As explained before, when dealing with datasets of dimensionality three or higher, our code uses a crop-and-chop process to avoid computations involving evaluation points contained in a bounding box but not affected by a sample. In order to assess the performance gains derived from using this technique, we ran tests with the 3D dataset 1, with crop-and-chop disabled (meaning that the per-sample bounding box is a rectangular prism) or enabled (meaning that, for each sample, a collection of bounding rectangles of different sizes is processed). Results are reported in Figure 12a (S-KDE in the i7) and 12b (S-KDE in the Xeon Phi). These tests clearly show the performance gains obtained with crop-and-chop: S-KDE runs 60.4% faster in the Core i7, and 49.2% faster in the Xeon Phi, due to the removal of useless computations (and the corresponding memory accesses). For example, in the tests with 194.81 millions of evaluation points, each 3D rectangular bounding box contains 102461 points on average; crop-and-chop reduces the number of processed evaluation points by half: 53511.

5.4. Identifying performance bottlenecks

Although the execution speed achieved by S-KDE in the two tested platforms is very competitive, we have performed additional tests and analyses in order to better understand the behavior of our code, and to identify possible bottlenecks and opportunities to improve its performance. To that extent we have used Intel's VTune Amplifier, which provides detailed information about the time spent by the program running



(a) Core i7 (b) Xeon Phi

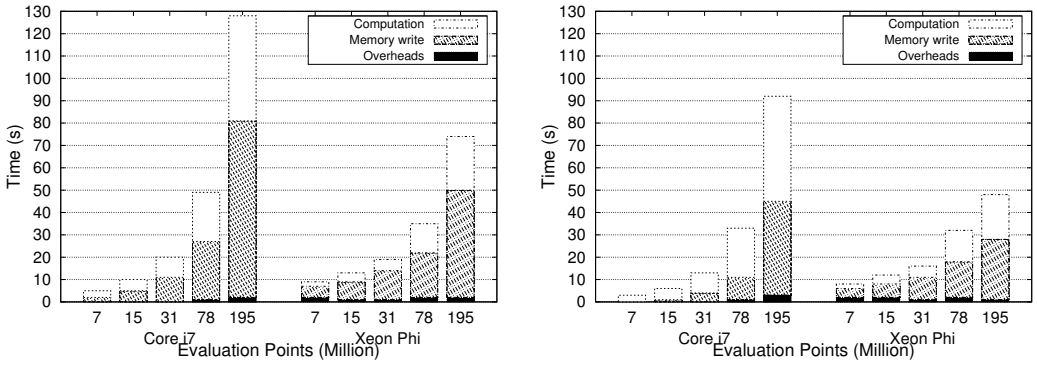
Fig. 12. Execution times for 3D dataset 1 with crop-and-chop enabled / disabled

different portions of the code. We have been able to dissect the execution times into three portions: (1) computation: time devoted to calculations of KDE, without taking into account memory writes; (2) memory writes: time spent in synchronized memory writes of partial results; and (3) overheads: initialization, ending and additional operations, including I/O.

As this dissection is very similar for all the tested scenarios, we focus again on the 3D dataset 1, depicting the results in Figure 13a. It shows that, on average, the code spends half the time computing KDE, and the other half performing atomic memory writes. Overheads are almost negligible for the datasets and evaluation spaces that we tested.

Time spent in writing operations may appear excessive, but we have to consider that any algorithm computing kernel density estimation requires extensive traversals of large datasets: it is a memory bounded problem. In our particular case, S-KDE needs to consolidate into main memory the per-sample results of the partial densities for each evaluation point inside the bounding boxes. Additionally, bounding boxes can overlap: the influence of several samples on the same evaluation point has to be aggregated. As different threads take care of different samples, writes must be synchronized using the atomic OpenMP pragma to avoid race conditions, and atomicity involves processing overheads. In order to understand the volumes of data we are talking about, we can provide these figures: working with the 3D dataset 1 in the largest evaluation space, each bounding box contains 53511 evaluation points (with crop-and-chop enabled); the sum of all

the bounding boxes would use a total of 199,34 GB but, as they overlap, they have to be consolidated into a matrix (representing the whole evaluation space) of 1,46 GB.



(a) Atomic writes

(b) Normal writes

Fig. 13. Dissection of execution times of S-KDE for 3D dataset 1

Considering these facts, there are two possible ways of accelerating S-KDE: reducing memory operations and reducing inter-thread coordination when performing writes. We have already discussed the chop-and-crop technique for reducing the set of evaluation points computed per sample. Before dealing with the second, we were interested in measuring the potential benefits of avoiding write synchronization. To do so, we run the same experiments removing the atomicity constraint, reporting the results in Figure 13b. The resulting output values of those S-KDE runs are not valid, but the execution times allowed us to estimate the room for improvement. The removal of the atomic directive makes our code run, on average, 42.7% faster in the i7, and 14.5% faster in the Xeon Phi.

The first conclusion we can draw from these figures is that the effect of synchronized writing is more important in the i7 than in the Xeon Phi. This is due to the differences between these two architectures. The Core i7 CPU has four memory channels, with on-chip memory controllers, 256 KB of per-core L2 cache and 10 MB of shared L3 cache. In comparison, the 57 cores in the Xeon Phi are connected through a bidirectional ring, and share 28.5 MB of L2 cache. We presume that cost of atomic writes is relatively higher in the i7 than in the Xeon Phi due to the smaller size of i7's caches, and also because memory operations are already partially serialized in the Xeon Phi by its interconnection network.

The second conclusion is that there is room for performance improvements, particularly in the multi-core CPU, if we guarantee the correct operation of memory writes while avoiding the overheads derived from the use of atomic directives. A first idea is to coordinate the way threads process samples, in such a way that those samples being processed simultaneously have non-overlapping bounding boxes. This approach requires a previous ordering on samples in non-overlapping groups, with the risk of suffering from an ordering overhead that surpasses the achieved benefit. Another approach could be exploiting even further the chop-and-crop mechanism, making a different use of the available parallelism: samples could be processed sequentially, but per-sample cropped slices could be processed in parallel, because they never overlap. Currently, this approach is not applicable to 2D problems. Additionally, the number of slices should be large enough to efficiently use all the available processors, and distribution of slices to threads must be correctly load balanced, because they are of different sizes. We have left the implementation of these techniques as future work.

6. Conclusions and Future Work

Experiments with the current implementation of S-KDE have shown that, compared with available, state-of-the-art implementations of KDE, it provides, by far, the best performance, even when running in a modest i7 processor. This performance can be boosted if a many-core coprocessor is available. However, we do not consider S-KDE as a finished product: we have analyzed its behavior, and identified different mechanisms to accelerate it, specially for multi-core processors.

The speed achieved by S-KDE come from three sources. The main one is algorithmic: the sample-wise approach to estimate densities has lower complexity than the evaluation point-wise approach, requiring fewer memory accesses. These accesses are further reduced by implementing the chop-and-crop technique for problems of dimensionality three and higher. Finally, S-KDE benefits from the exploitation of parallel architectures, in particular multi-core CPUs and many-core coprocessors.

Currently, S-KDE is only applicable to bounded kernels, and its scalability is limited by the available memory. Learning from R's ks-kde, we could include support for unbounded kernels if limits to its influence area are imposed (effectively implementing bounding boxes); we could also work with extremely large evaluation spaces if the output of the program is not a complete matrix with all the evaluation points, but a list of user-defined evaluation points in which the estimation of the density is required. Another approach

to deal with large-scale problems in reduced-memory situations could be to adopt a divide-and-conquer approach, dividing the evaluation space in zones, consolidating them as a final step. These modifications, together with the exploration of mechanisms to reduce memory contention, are part of our lines of future work.

From the point of view of parallel processing, we are also working on making the multi-core and the attached coprocessor run concurrently on the same problem. Going further, several accelerators and CPUs on a single system could be used simultaneously to greatly accelerate KDE computations.

The S-KDE code is available under request, for any researcher performing density estimation tasks.

A. Detailed description of the cropping technique

This appendix complements Section 3.3, providing a detailed explanation of the cropping process. We describe here equations used to adjust an optimal bounding box to each ellipse in a slice.

From a d dimensional bounding box, a nested loop traverses every $d-1$ box until 3 dimensional boxes are obtained. The cropping calculations begin in three dimensional spaces. First, the equation that represents the shape of the kernel, i.e., the equation of a 3D ellipsoid, must be completed:

$$Ax^2 + Bxy + Cy^2 + Dxz + Eyz + Fz^2 = 1 \tag{11}$$

where terms A to F are filled with values from the inverse of the covariance matrix of the dataset Σ^{-1}. At this point the bandwidth parameter must be applied as $\Sigma'^{-1} = \Sigma^{-1}h^{-2}$ to modify the kernel accordingly. Then, terms A, C and F of the equation are completed with $\Sigma'^{-1}_{1,1}$, $\Sigma'^{-1}_{2,2}$ and $\Sigma'^{-1}_{3,3}$ respectively, and terms B,D and E with $2 * \Sigma'^{-1}_{1,2}$, $2 * \Sigma'^{-1}_{1,3}$ and $2 * \Sigma'^{-1}_{2,3}$.

Once the equation of the kernel is completed, the 2D slices that form the 3D bounding box are traversed along the z axis. Given a slice at distance z from the center of the ellipsoid, we must calculate several parameters from its ellipse to make the cropping. The first step is to substitute the distance of the slice in the equation of the 3D ellipsoid, to obtain the equation that represents the ellipse in the slice :

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0 \tag{12}$$

From this equation the series of steps described in Section 3.3 are applied to crop the optimal bounding box to the 2D slice.

Step1– Calculate the rotation angle of the ellipse in the slice: The rotation angle θ is calculated as:

$$\theta = \frac{\arctan(\frac{B}{A-C})}{2} \tag{13}$$

Step2– Calculate the length of the principal axes of the ellipse: We first calculate the equation of the ellipse in a unrotated manner as:

$$\begin{aligned} A' &= Acos^2\theta + Bcos\theta sin\theta + Csin^2\theta \\ B' &= 0 \\ C' &= Asin^2\theta - Bcos\theta sin\theta + Ccos^2\theta \\ D' &= Dcos\theta + Esin\theta \\ E' &= -Dsin\theta + Ecos\theta \\ F' &= F \end{aligned} \tag{14}$$

where terms A to F belong to Equation 12 and terms A' to F' represent the terms of equation of the same ellipse, but in an unrotated manner. From this new equation we can derive the lengh of the principal axes a and b of the ellipse as:

$$a = \sqrt{\frac{-4F'A'C' + C'D'^2 + A'E'^2}{4A'C'^2}} \qquad b = \sqrt{\frac{-4F'A'C' + C'D'^2 + A'E'^2}{4A'^2C'}} \tag{15}$$

Step3– Calculate the coordinates of the edge vertices: To do so, we first calculate the coordinates c'_x, c'_y of the center of the unrotated ellipse as:

$$c'_x = \frac{-D'}{2A'} \qquad c'_y = \frac{-E'}{2C'} \tag{16}$$

to then calculate the edge vertices vx' and vy' as:

$$\begin{aligned} vx'_x &= c'_x + a & vy'_x &= c'_x \\ vx'_y &= c'_y & vy'_y &= c'_y + b \end{aligned} \tag{17}$$

$\mathbf{Step4}-$ Crop the bounding box: First, we must apply the rotation to edge vertices vx and vy as:

$$\begin{aligned} vx_x &= vx'_x cos\theta + vx'_y sin\theta & vy_x &= vy'_x cos\theta + vy'_y sin\theta \\ vx_y &= -vx'_x sin\theta + vx'_y cos\theta & vy_y &= -vy'_x sin\theta + vy'_y cos\theta \end{aligned} \tag{18}$$

to then find the boundaries of the ellipse applying the Euclidean norm as:

$$bound_x = \sqrt{vx_x^2 + vy_x^2} \qquad bound_y = \sqrt{vx_y^2 + vy_y^2} \tag{19}$$

The last step is to calculate the coordinates of the bounding box aligned to the evaluation grid, rounding $bound_x$ and $bound_y$ to the evaluation step per dimension.

References

[1] Silverman BW. Density estimation for statistics and data analysis. vol. 26. Chapman & Hall/CRC; 1986.

[2] Ahamada I, Flachaire E. Non-Parametric Econometrics. No. 9780199578009 in OUP Catalogue. Oxford University Press; 2010. Available from: http://ideas.repec.org/b/oxp/obooks/9780199578009.html.

[3] Andrés Ferreyra R, Podestá GP, Messina CD, Letson D, Dardanelli J, Guevara E, et al. A linked-modeling framework to estimate maize production risk associated with ENSO-related climate variability in Argentina. Agricultural and Forest Meteorology. 2001;107(3):177–192.

[4] Corti S, Molteni F, Palmer T. Signature of recent climate change in frequencies of natural atmospheric circulation regimes. Nature. 1999;398(6730):799–802.

[5] Weissbach R. A general kernel functional estimator with general bandwidth-strong consistency and applications. Journal

of Nonparametric Statistics. 2006;18(1):1–12.

[6] Elgammal A, Duraiswami R, Davis LS. Efficient kernel density estimation using the fast gauss transform with applications to color modeling and tracking. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2003;25(11):1499–1504.

[7] Bosman PA, Thierens D. IDEAs based on the normal kernels probability density function. Department of Computer Science, Utrecht University; 2000. 11.

[8] Luo N, Qian F. Evolutionary algorithm using kernel density estimation model in continuous domain. In: Asian Control Conference, 2009. ASCC 2009. 7th. IEEE; 2009. p. 1526–1531.

[9] Sheather SJ. Density estimation. Statistical Science. 2004;p. 588–597.

[10] Jeffers J, Reinders J. Intel Xeon Phi Coprocessor High Performance Programming. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2013.

[11] The OpenACC Application Programming Interface; 2013. 2.0a. Available from: http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.

[12] Scott DW. Multivariate density estimation: theory, practice, and visualization. vol. 383. Wiley; 2009.

[13] Givens GH, Hoeting JA. Computational statistics. Wiley series in probability and statistics. Wiley-Interscience; 2005.

[14] Fukunaga K. Introduction to statistical pattern recognition (2nd ed.). San Diego, CA, USA: Academic Press Professional, Inc.; 1990.

[15] Lopez-Novoa U, Mendiburu A, Miguel-Alonso J. A Survey of Performance Modeling and Simulation Techniques for Accelerator-Based Computing. Parallel and Distributed Systems, IEEE Transactions on. 2015 Jan;26(1):272–281.

[16] Leyk Z, Stewart D. Meschach: matrix computation in C. Proceedings of the CMA. 1994;32:1–240.

[17] Racine J. Parallel distributed kernel estimation. Computational statistics & data analysis. 2002;40(2):293–302.

[18] Łukasik S. Parallel computing of kernel density estimates with MPI. Computational Science–ICCS 2007. 2007;p. 726–733.

[19] Srinivasan BV, Hu Q, Duraiswami R. GPUML: Graphical processors for speeding up kernel machines. In: Workshop on High Performance Analytics-Algorithms, Implementations, and Applications; 2010. .

[20] Michailidis PD, Margaritis KG. Accelerating Kernel Density Estimation on the GPU Using the CUDA Framework. Applied Mathematical Sciences. 2013;7(30):1447–1476.

[21] Millman KJ, Aivazis M. Python for Scientists and Engineers. Computing in Science & Engineering. 2011;13(2):9–12.

[22] Seabold S, Perktold J. Statsmodels: Econometric and Statistical Modeling with Python. In: Proceedings of the 9th Python in Science Conference; 2010. p. 57–61.

[23] Duong T. ks: Kernel density estimation and kernel discriminant analysis for multivariate data in R. Journal of Statistical Software. 2007;21(7):1–16.

[24] Rothberg JM, Hinz W, Rearick TM, Schultz J, Mileski W, Davey M, et al. An integrated semiconductor device enabling non-optical genome sequencing. Nature. 2011;475(7356):348–352.

[25] Venables WN, Ripley BD. Modern Applied Statistics with S. 4th ed. New York: Springer; 2002.

[26] Perkins S, Pitman A, Holbrook N, McAneney J. Evaluation of the AR4 Climate Models' Simulated Daily Maximum Temperature, Minimum Temperature, and Precipitation over Australia Using Probability Density Functions. Journal of climate. 2007;20(17):4356–4376.