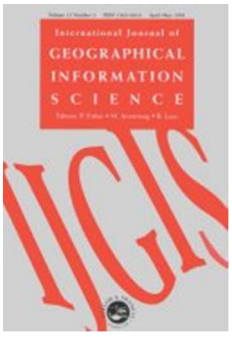

A GPU-accelerated adaptive kernel density estimation approach for efficient point pattern analysis on spatial big data

Journal:	*International Journal of Geographical Information Science*
Manuscript ID	IJGIS-2016-0209.R4
Manuscript Type:	Research Article
Keywords:	Adaptive kernel density estimation, Optimization, GPU/CUDA, OpenMP, Spatial big data

International Journal of Geographical Information Science

RESEARCH ARTICLE

A GPU-accelerated adaptive kernel density estimation approach for efficient point pattern analysis on spatial big data

Kernel density estimation (KDE) is a classic approach for spatial point pattern analysis. In many applications, KDE with spatially adaptive bandwidths (*adaptive* KDE) is preferred over KDE with an invariant bandwidth (*fixed* KDE). However, bandwidths determination for adaptive KDE is extremely computationally intensive, particularly for point pattern analysis tasks of large problem sizes. This computational challenge impedes the application of *adaptive* KDE to analyze large point data sets, which are common in this big data era. This article presents a graphics processing units (GPU)-accelerated *adaptive* KDE algorithm for efficient spatial point pattern analysis on spatial big data. First, optimizations were designed to reduce the algorithmic complexity of the bandwidth determination algorithm for *adaptive* KDE. The massively parallel computing resources on GPU were then exploited to further speed up the optimized algorithm. Experimental results demonstrated that the proposed optimizations effectively improved the performance by a factor of tens. Compared to the sequential algorithm and an Open Multiprocessing (OpenMP)-based algorithm leveraging multiple CPU cores for *adaptive* KDE, the GPU-enabled algorithm accelerated point pattern analysis tasks by a factor of *hundreds* and *tens* respectively. Additionally, the GPU-accelerated *adaptive* KDE algorithm scales reasonably well while increasing the size of data sets. Given the significant acceleration brought by the GPU-enabled *adaptive* KDE algorithm, point pattern analysis with the *adaptive* KDE approach on large point data sets can be performed efficiently. Point pattern analysis on spatial big data, computationally prohibitive with the sequential algorithm, can be conducted routinely with the GPU-accelerated algorithm. The GPU-accelerated *adaptive* KDE approach contributes to the geospatial computational toolbox that facilitates geographic knowledge discovery from spatial big data.

Keywords: *Adaptive* kernel density estimation, optimization, GPU/CUDA, OpenMP, spatial big data

1. Introduction

Kernel density estimation (KDE) is a classical approach to spatial point pattern analysis by identifying interesting 'hotspots' of point events (Diggle 1985; Burt et al. 2009; Fotheringham et al. 2000). It has been widely used in many application domains such as spatial epidemiology, crime pattern analysis, ecology, biology, traffic accidents analysis, etc. (Worton 1989; Gatrell et al. 1996; Xie & Yan 2008; Law et al. 2009). The basic assumption of KDE is that an event occurred at a given location (i.e., $\boldsymbol{X}_i$) could occur at another location (i.e., $\boldsymbol{x}$) at a lower probability and the probability decreases as the distance from $\boldsymbol{X}_i$ to $\boldsymbol{x}$ (i.e., d_i) increases. This probability can be described by a function, referred to as the kernel function $K(\cdot)$, whose value (the probability) is tied to d_i. A typical function for $K(\cdot)$ is the Gaussian kernel (adopted in this article) (Silverman 1986):

$$K\left(\frac{|\boldsymbol{x}-\boldsymbol{X}_i|}{h_i}\right) = \frac{1}{2\pi} e^{-\frac{|\boldsymbol{x}-\boldsymbol{X}_i|^2}{2{h_i}^2}} \quad (1)$$

where $|\boldsymbol{x}-\boldsymbol{X}_i|$ is d_i (the distance between the two points), h_i is a smoothing parameter called *bandwidth* that determines how quickly the probability decreases as d_i increases. KDE then estimates the probability density at any location x by summing up density contribution from all sample points (Diggle 1985; Fotheringham et al. 2000):

$$\hat{f}(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{h_i^2} K\left(\frac{|\boldsymbol{x}-\boldsymbol{X}_i|}{h_i}\right) e_i \quad (2)$$

where $\hat{f}(\boldsymbol{x})$ is the estimated density at location $\boldsymbol{x}$, h_i is the bandwidth for sample point $\boldsymbol{X_i}$, n is the total number of sample points, and e_i is an edge correction factor for $\boldsymbol{X_i}$ which is necessary for obtaining unbiased density estimates (Diggle 1985; Baddeley et al. 2015). The choice of the bandwidths is critical to KDE (Epanechnikov 1969; Silverman 1986). The bandwidths can either be invariant at sample points (*fixed* KDE) (Silverman 1986; Miecznikowski et al. 2010; Jones et al. 1996; Brunsdon 1995), or vary spatially across sample points (*adaptive* KDE) (Breiman et al. 1977; Worton 1989; Jones 1990; Brunsdon 1995; Sain et al. 1996; Shi 2010; Silverman 1986).

In many applications, *adaptive* KDE is preferred over *fixed* KDE. Adaptive KDE is particularly useful for analyzing human geographical data (e.g., retail store locations) where most events (i.e., locations) occur in the most densely populated areas and it can discern probability density variations in these areas (Breiman et al. 1977; Worton 1989; Brunsdon 1995; Fotheringham et al. 2000). It is also widely used for analyzing population related events such as disease, crime, and healthcare clinics because it can produce more meaningful density estimates by considering varying population distribution across space (Gatrell et al. 1996; Carlos et al. 2010; Shi 2010).

Determining the spatially adaptive bandwidths is the key step while applying *adaptive* KDE in spatial point pattern analysis. Following the principle that sample points in areas of low density should have larger bandwidth values and sample points in areas of high density should have smaller bandwidth values, Breiman et al. (1977) proposed to set bandwidths proportional to nearest neighbor distances. This implies the dependence $h_i \propto f(\boldsymbol{X}_i)^{-1}$ where $f(\boldsymbol{X}_i)$ is the true density at sample location $\boldsymbol{X}_i$ (Abramson 1982). Abramson (1982) showed that the dependence was too strong and suggested $h_i \propto f(\boldsymbol{X}_i)^{-0.5}$ instead. Silverman (1986) proposed to represent the bandwidths as

$$h_i = h\lambda_i \tag{3}$$

where h is a global bandwidth, and λ_i is a local smoothing parameter given by

$$\lambda_i = \left\{\frac{\tilde{f}(\boldsymbol{X}_i)}{g}\right\}^{-\alpha} \tag{4}$$

where α is a non-negative constant, $\tilde{f}(\boldsymbol{X}_i)$ is a pilot estimate of the density at sample point $\boldsymbol{X}_i$, and g is the geometric mean of the density values at the n sample points and hence $\log(g) = \frac{1}{n}\sum_{i=1}^{n}\log\tilde{f}(\boldsymbol{X}_i)$. $\tilde{f}(\boldsymbol{X}_i)$ is estimated using *fixed* KDE with a *prescribed* bandwidth value h_0 (Worton 1989). Three parameters need to be determined to compute spatially adaptive bandwidth for adaptive KDE using equation (3) and (4), including: the global bandwidth h, the bandwidth h_0

used in pilot density estimation, and α. No objective procedures have been developed to select h_0, although h can be determined using cross-validation on sample points if α is specified a priori (Silverman 1986). In many cases, one often simply subjectively specifies values for the two bandwidth parameters. As for α, Abramson (1982) and Silverman (1986) suggested that 0.5 is appropriate. However, setting $\alpha = 0.5$ is not an objective decision made based upon the data (i.e., sample points) either.

Brunsdon (1995) proposed an algorithm to objectively determine both α and h using sample points data. The 'optimal' values of α and h are calculated iteratively using cross-validation on the sample points based on the maximum likelihood criterion. In addition, the algorithm uses h in pilot density estimation in each iteration (i.e., $h_0 = h$) instead of choosing an arbitrary bandwidth for pilot density estimation. Note that although cross-validation can be used in bandwidth selection for both fixed KDE and adaptive KDE (Silverman 1986; Brunsdon 1995; Jones et al. 1996; Sain et al. 1996), using Brunsdon's algorithm to determine the optimal adaptive bandwidths involves searching for the optimal values of α and h simultaneously in a 2-dimensional search space while only a 1-dimensional search space is needed to determine the optimal fixed bandwidth h. Correspondingly, Brunsdon's algorithm for adaptive KDE is more computationally demanding than those for fixed KDE.

Determining the spatially adaptive optimal bandwidths for *adaptive* KDE using *Brunsdon's algorithm* is very computationally challenging, particularly on point events data of large size. Runtime of *Brunsdon's algorithm* increases rapidly as the number of sample points increases (see *Section* 2 and 3 for details). Spatial big data are now becoming ubiquitous as location-based services, social media, volunteered geographic information, citizen science projects, etc. are producing a huge amount of point data on a daily basis (Goodchild 2007; Shekhar et al. 2012; Wood et al. 2011; Huang & Wong 2015; Zhu et al. 2015; Longley & Adnan 2016). Whilst some time-critical patterns in these data are of interest (e.g., spatial pattern of real-time tweeting),

applying *adaptive* KDE for point pattern analysis on such spatial big data is prohibitively time-consuming. This gap needs to be filled in order to enable efficient point pattern analysis on spatial big data (Zhang et al. 2016).

Under the umbrella of high-performance computing, computing resources on distributed devices have been leveraged to address the computational challenges associated with geoprocessing and spatial analysis of big data through cluster computing, grid computing, heterogeneous computing, or cloud computing (Schadt et al. 2010; Wang 2010; Wright & Wang 2011; Yang et al. 2011; Shi et al. 2014). Multi-core central processing units (CPUs) and many-core graphics processing units (GPUs) on a single computer or on distributed computers are typical computing resources that have been exploited to accelerate geospatial applications (e.g., Guan & Clarke 2010; Qin et al. 2014; Tang et al. 2015). A modern GPU has hundreds of cores that can run thousands of threads in parallel, reaching a peak computing performance much higher than most advanced CPUs (Kirk & Hwu 2012; NVIDIA 2016). GPUs have been playing a dominant role in leveraging many-core computing architecture for general purpose computation, and have been introduced to the GIScience community to greatly accelerate various geospatial analysis algorithms such as viewshed analysis (Zhao et al. 2013; Osterman et al. 2014), quad-tree construction on rasters (Zhang & You 2013), cartogram construction (Tang 2013), point pattern analysis (Tang et al. 2015), cellular automata model (Guan et al. 2016), and polygon rasterization (Zhou et al. 2016).

In this article, we propose to leverage GPUs to address the computational challenge associated with determining the spatially adaptive optimal bandwidths for *adaptive* KDE. Although GPUs have been used for accelerating many geospatial algorithms, using GPUs to speed up spatial point pattern analysis in particular still remains at an early stage (Tang et al. 2015). A GPU-enabled Ripley's *K* function for massively parallel point pattern analysis was developed by Tang et al. (2015). Using the GPU-enabled *K* function for point pattern analysis, the acceleration factor can reach up to about 50 and 1, 501 for a single GPU and 50 GPUs respectively (Tang et al. 2015).

As for GPU-accelerated KDE in general, Michailidis & Margaritis (2013) implemented univariate and multivariate KDE with the support of GPUs using the Compute Unified Device Architecture (CUDA) programming model. However, these are *fixed* KDE implementations using only *prescribed* bandwidth that involve no bandwidth selection procedures. Andrzejewski et al. (2013) proposed efficient GPU versions of bandwidth selection algorithms for univariate and multivariate KDE. Yet the bandwidth selection algorithms are only for finding the optimal bandwidths for each individual multivariate dimension. While the optimal bandwidths vary across different variable dimensions, in each dimension sample points still have the same fixed bandwidth. The algorithms thus are useful only for determining optimal bandwidth for *fixed* KDE. They did not address the computation challenge to determine optimal bandwidths for *adaptive* KDE.

Edge correction is necessary to obtain unbiased density estimations for the KDE approach to spatial point pattern analysis (Diggle 1985; Baddeley et al. 2015). However, neither Michailidis & Margaritis (2013) nor Andrzejewski et al. (2013) implemented edge correction in their GPU-based KDE algorithms, as the algorithms were intended for density estimation in multivariate space rather than over geographic space. The original algorithm developed by Brunsdon (1995), though specifically designed for density estimation over geographic space, did not implement edge correction either. In this article, we supplement Brunsdon's algorithm with edge correction implementation.

This article presents a GPU-accelerated *adaptive* KDE approach for efficient spatial point pattern analysis on large data sets. *Brunsdon's algorithm* (edge correction enabled) for determining the spatially adaptive optimal bandwidths was accelerated by exploiting massively parallel computing resources on the GPU. Details of *Brunsdon's algorithm* are provided in Section 2. Optimizations for *Brunsdon's algorithm* are presented in Section 3. The design and implementation of the GPU-enabled *adaptive* KDE approach are described in Section 4. The

effectiveness of the optimizations and the correctness, efficiency, and scalability of the GPU-enabled *adaptive* KDE approach are then evaluated with experiments in Section 5. Conclusions are drawn in Section 6.

2. Determining spatially adaptive optimal bandwidths for adaptive KDE

Brunsdon's algorithm determines the spatially adaptive optimal bandwidths by finding the (α, h) values that maximize the likelihood (probability) of observing the n sample points. This likelihood is a product of the probabilities of observing each sample point, which is estimated from the other n-1 sample points using *adaptive* KDE with certain (α, h) values to compute bandwidths. The likelihood of observing the n sample points differs under different (α, h) values. *Brunsdon's algorithm* adopts a search routine to find the optimal (α, h) values that maximize this likelihood. The optimal (α, h) values are then used to compute spatially adaptive bandwidths for *adaptive* KDE to estimate a probability density surface over the study area (Brunsdon 1995; Hooke & Jeeves 1961).

The flow chart of *Brunsdon's algorithm* is shown in Figure 1. Starting with some initial (α, h) values, initial step values $(\Delta_\alpha, \Delta_h)$, and prescribed step threshold values $(\epsilon_\alpha, \epsilon_h)$, the algorithm goes through a series of iterations. In this article, the initial $\alpha = \alpha_0 = 0.5$, initial $h = h_0$ where h_0 is the 'rule-of-thumb' bandwidth (Fotheringham et al. 2000, p149), initial $\Delta_\alpha = 0.1$, initial $\Delta_h = h_0/10$, $\epsilon_\alpha = \Delta_\alpha/20 = 0.005$, and $\epsilon_h = \Delta_h/20$. In each iteration, (α, h) values are slightly adjusted towards the maximum likelihood direction. Within each iteration, the following steps and procedures are involved (Figure 1).

Step 1: compute edge correction factors for sample points under the current value of h. Edge correction factor for a sample point is the reciprocal of the kernel mass *inside* the study area (the kernel is centered at that sample point) (Diggle 1985; Baddeley et al. 2015), and depends on the bandwidth at that sample point. This is because the bandwidth determines the spread of the kernel

centered at that sample point. Given an h, edge correction factor e_i for sample point $\boldsymbol{X}_i$ is computed using the equation below (e_i is 1.0 if the kernel mass is completely within the study area) (Diggle 1985; Baddeley et al. 2015):

$$\frac{1}{e_i} = \int_{\boldsymbol{v}\in W} K\left(\frac{|\boldsymbol{X}_i - \boldsymbol{v}|}{h}\right) d\boldsymbol{v} \cong \sum_{j=1}^{m} K\left(\frac{|\boldsymbol{X}_i - \boldsymbol{c}_j|}{h}\right) \Delta A \quad (5)$$

where $\boldsymbol{v}$ is a point in study area W, $\boldsymbol{c_j}$ is the center point of the j^{th} raster cell in the study area, ΔA is the area of a raster cell, and m is the total number of cells. Note that here the bandwidth h is the same for all sample points.

Step 2: compute pilot density estimates at sample points under the current value of h. The pilot density at a sample point is estimated from sample points *including* the foci sample point itself (referred to as inclusive density) using *fixed* KDE. The pilot density estimation at sample point $\boldsymbol{X}_i$ is computed using the equation below:

$$\tilde{f}(\boldsymbol{X}_i) = \frac{1}{n}\sum_{j=1}^{n} \frac{1}{h^2} K\left(\frac{|\boldsymbol{X}_i - \boldsymbol{X}_j|}{h}\right) e_j \quad (6)$$

where $\tilde{f}(\boldsymbol{X}_i)$ is the pilot density estimate at sample point $\boldsymbol{X}_i$. Notice that the foci sample point itself is included in density estimation to avoid zero density at the foci sample point, and that the bandwidth h is the same for all sample points.

Step 3: compute spatially adaptive bandwidths for sample points following equation (3) and (4), under the current (α, h) values and the pilot density estimates computed in step 2.

Step 4: update (re-compute) edge correction factors for sample points using the equation below, given the spatially adaptive bandwidths computed in *step 3*:

$$\frac{1}{e_i} = \int_{\boldsymbol{v}\in W} K\left(\frac{|\boldsymbol{X}_i - \boldsymbol{v}|}{h_i}\right) d\boldsymbol{v} \cong \sum_{j=1}^{m} K\left(\frac{|\boldsymbol{X}_i - \boldsymbol{c}_j|}{h_i}\right) \Delta A \quad (7)$$

This equation differs from equation (5) as the bandwidths h_i vary across sample points.

Step 5: compute the probability of observing each sample point. This probability is estimated from sample points *excluding* the foci sample point (referred to as exclusive density) using *adaptive* KDE, which is computed using the equation below given the spatially adaptive bandwidths computed in *step 3* and the edge correction factors computed in *step 4*:

$$\tilde{f}^*(\boldsymbol{X}_i) = \frac{1}{n-1}\sum_{j=1,j\neq i}^{n} \frac{1}{{h_j}^2} K\left(\frac{|\boldsymbol{X}_i - \boldsymbol{X}_j|}{h_j}\right) e_j \quad (8)$$

where $\tilde{f}^*(\boldsymbol{X}_i)$ is the probability of observing sample point $\boldsymbol{X}_i$ given the other n-1 sample points. This equation differs from equation (6) in that the foci sample point is excluded in density estimation to prevent the algorithm from directly returning h=0 as the optimal bandwidth (Brunsdon 1995), and that the bandwidths h_j vary across sample points.

Step 6: compute the Log-likelihood of observing the n sample points under the current (α, h) values:

$$L(\alpha, h) = \sum_{i=1}^{n} \log \tilde{f}^*(\boldsymbol{X}_i) \quad (9)$$

in which $L(\alpha, h)$ is the Log-likelihood and $\tilde{f}^*(\boldsymbol{X}_i)$ is the probability of observing sample point $\boldsymbol{X}_i$ which was computed in *step 5*. Computing Log-likelihood instead of likelihood avoids potential numerical issues that might arise when the product of the probability values is very small.

Before proceeding to the next step, follow steps 1 through 6 to compute $L(\alpha+\Delta_\alpha, h)$, $L(\alpha-\Delta_\alpha, h)$, $L(\alpha+\Delta_\alpha, h+\Delta_h)$, and $L(\alpha-\Delta_\alpha, h-\Delta_h)$ — the Log-likelihood values corresponding to the neighbors of the (α, h) values in the 2-dimensional search space.

Step 7: Find the local maximum Log-likelihood L_{max} among $L(\alpha+\Delta_\alpha, h)$, $L(\alpha-\Delta_\alpha, h)$, $L(\alpha, h)$, $L(\alpha+\Delta_\alpha, h+\Delta_h)$, and $L(\alpha-\Delta_\alpha, h-\Delta_h)$. Suppose $L_{max} = L(\alpha', h')$ where (α', h') are the values corresponding to the local maximum (this ensures that the algorithm always proceeds towards the global maximum likelihood direction in the search space).

Then the algorithm evaluates whether to stop iteration. If $L_{max} \neq L(\alpha, h)$, update the current (α, h) with (α', h') and continue to the next iteration. Otherwise, the step values $(\Delta_\alpha, \Delta_h)$ are decreased by half, and checked against their thresholds $\epsilon_\alpha, \epsilon_h$. Iteration stops if $\Delta_\alpha < \epsilon_\alpha$ and $\Delta_h < \epsilon_h$, or if the number of iterations exceeds the predefined threshold (30 iterations in this article). Otherwise, it continues to the next iteration.

Once iteration stops (the algorithm converges or the number of iterations exceeds the predefined threshold), the optimal (α, h) values that maximize the likelihood of observing the *n* sample points are determined. The optimal (α, h) values are then used to compute the spatially adaptive bandwidths (equation 3 and 4) and the edge correction factors (equation 7) for *adaptive* KDE to estimate a probability density surface over the study area (equation 2).

[Figure 1 is about here]

Figure 1. Flow chart of the sequential *Brunsdon's algorithm* for *adaptive* KDE.

3. Optimizations

Spatial point pattern analysis using Brunsdon's algorithm for point pattern analysis on big point events data sets over large areas at fine spatial resolution would be extremely computationally demanding. The algorithmic complexity of a naïve implementation of *Brunsdon's algorithm* is $O(nm + n^2)$. The $O(nm)$ part is dominated by the computation involved in computing edge correction factors as it is repeated many time across the iterations (computing density surface over the study area is also $O(nm)$ but it is only performed once). The $O(n^2)$ part reflects the computation involved in estimating density values at the sample points across the iterations. Run time of the algorithm would increase quadratically as the number of sample points *n* increases, or linearly as the number of cells *m* in the study area increases. It is thus desirable to optimize the algorithm for performance. We present two optimizations for Brunsdon's algorithm to reduce its complexity before leveraging GPU to speed up the algorithm.

3.1 A heuristic to avoid re-computing edge correction factors

The first optimization is based on the heuristic that only edge correction factors of sample points close to the study area boundary need to be re-computed when updating bandwidths. Sample points far away from the study area boundary should have a constant edge correction factor very close to 1.0 (i.e., the kernel mass is almost completely within the study area). If such sample points can be identified, the cost of re-computing edge correction factors for them can be avoided.

This optimization was implemented as the following. For each sample point $\boldsymbol{X}_i$, its closest distance to the study area boundary d_i is computed. Later when computing edge correction factor for $\boldsymbol{X}_i$ (equation 5 or 7), the bandwidth h_i is compared against d_i. If d_i is larger than $C \times h_i$ (C is a constant), a value of 1.0 is assigned to e_i (edge correction factor of this sample point). Otherwise, e_i is computed but the computation is restricted to raster cells that are within $C \times h_i$ distance from the center sample point instead of all cells in the study area. A larger C will produce more accurate edge correction factors but with more computation. At a location that is 3×bandwidth distance from the center sample point, the density value (height of the Gaussian kernel surface) is only 1.1% of the height at the center point. The kernel mass (volume under the kernel surface; reciprocal of the edge correction factor of the center point) (equation 5 and 7) over the 3×bandwidth radius circular neighborhood of the center point is 98.9% of the total kernel mass. Since density value of the Gaussian kernel is very small and negligible at a location that is beyond 3×bandwidth away from the center sample point and edge correction factor computed over the 3×bandwidth radius circular neighborhood is very close to the true value that needs an infinite support to compute, we consider 3 is an appropriate value for C as it was done in Brunsdon (1995). This optimization reduces the $O(nm)$ part of the complexity of Brunsdon's algorithm to about $O(n\sqrt{m})$.

3.2 Spatial indexing to speed up density estimation at sample points

The second optimization assumes that a sample point $\boldsymbol{X}_i$ contributes only to density at locations that are within $C \times h_i$ distance from $\boldsymbol{X}_i$ (its contribution to locations out of the radius of $C \times h_i$ is ignored and set to zero). Correspondingly, when computing density at a location, we only need to integrate sample points that are within $C \times h_i$ distance from this location (note that h_i is the bandwidth of a sample point and different sample points have different bandwidths). Therefore, when computing density values at a given sample point (equation 6 or 8), the performance can be improved if we can quickly locate the sample points within the circular neighborhood of this point without checking all sample points.

A kd-tree spatial indexing data structure (Kakde 2005; Bentley & Friedman 1979) was utilized to implement this optimization. A 2-dimensional kd-tree (x, y coordinates are the two dimensions) was built on the sample points. It was then used to speed up range search to locate sample points within a certain distance from a foci sample point. Note that here the way to compute density at a sample point $\boldsymbol{X}_i$ is different from but still equivalent to equation (6) or (8). For each sample point $\boldsymbol{X}_i$, kd-tree is first used to find all sample points within its $C \times h_i$ radius circular neighborhood; then the contribution of $\boldsymbol{X}_i$ to the density at each of these neighboring sample points is computed (equation 1) and added to the accumulative density at the respective sample point. After performing the above two steps on all sample points, the density at each sample point is computed. The cost of building a kd-tree on n sample points is $O(n \log n)$ (Kakde 2005; Bentley & Friedman 1979), but it is amortized since the kd-tree is built once and used later in every range search. For a sample point, the complexity of range search with a kd-tree is $O(\sqrt{n} + K)$ where K is the number of points returned (Kakde 2005; Bentley & Friedman 1979). This optimization reduces the $O(n^2)$ part of the complexity of Brunsdon's algorithm to around $O(n\sqrt{n})$.

4. GPU-accelerated *adaptive* KDE for spatial point pattern analysis

4.1 GPU parallel programming model

Parallel computation on a GPU is achieved by invoking *kernels*. A kernel is a function executed concurrently by all threads running on a GPU. Thread is the basic computing unit on GPU. The total number of threads allowed on a modern GPU can be as many as millions or even billions. Threads are organized into *blocks*, which in turn are further organized into one, two or three dimensional *grids*. *Execution configuration* refers to *block size* (i.e., number of threads per block) and *grid size* (i.e., number of rows and columns of blocks per grid in two dimensional grids) and needs to be specified before launching a kernel. All threads on a GPU can read and write *global* GPU memory. Data processed on GPU need to be transferred from *host memory* (e.g., CPU-side memory) to global memory of the GPU first before they can be manipulated by GPU threads. At the same time, data in GPU global memory can be transferred back to the host memory for further processing (NVIDIA 2016).

Each thread has a unique serial thread identifier (i.e., *thread ID*). Thread ID is the key for assigning a partitioned computing task to a corresponding thread. Upon kernel invocation, all threads will concurrently execute the kernel. Each thread carries out computation specified in the kernel on different portions of the data (NVIDIA 2016). For example, when executing a kernel on GPU to compute edge correction factors, thread 0 computes the edge correction factor for sample point 0, thread 1 computes the edge correction factors for sample point 1, so on so forth. Threads are executed on physical GPU devices in 'warps' where each warp is a group of threads (e.g., 32 threads per warp for most current GPUs). In this way, a computing task can be carried out on a large number of GPU threads in a massively parallel fashion (Luebke 2008; Nickolls et al. 2008).

4.2 Design of GPU-accelerated *adaptive* KDE algorithm

The GPU-enabled *adaptive* KDE algorithm requires the cooperation of CPU-side and GPU-side routines. Kernels were implemented for the computation tasks involved in an iteration of *Brunsdon's algorithm*. A kernel was also implemented for computing *adaptive* KDE over the

study area (i.e., computing density surface). The CPU-side routine is responsible for the logical control of the algorithm and it invokes kernels (i.e., the GPU-side functions) to perform the computationally demanding tasks on GPU threads in a massively parallel fashion (Figure 2).

[Figure 2 is about here]

Figure 2. Flow chart of the GPU-enabled Brunsdon's algorithm for *adaptive* KDE.

Data for the GPU-enabled *adaptive* KDE algorithm is first loaded into the *host memory* of CPU environment and then transferred to the global memory of GPU (*device memory*) for parallel computation. The CPU-side then invokes kernels to perform computation tasks on the GPU using data in device memory. The GPU-enabled *adaptive* KDE algorithm is designed to minimize the data exchange between host memory and device memory as it is an expensive operation. The kernels perform computation tasks with input data in device memory and write output to device memory. A following launched kernel can take the output of a previously completed kernel as input (which is still on device memory). There is little data transfer between host and device across kernel invocations. Results on device memory are transferred back to host memory only once upon the completion of the last kernel.

4.3 Implementation

The GPU-enabled *adaptive* KDE algorithm was implemented using CUDA version 7.0 in the C/C++ programming language (NVIDIA 2016). The CPU-side control routine loads the data of sample points (i.e., coordinates) and study area (i.e., a raster indicating study area extent) from external files into host memory. It then computes the distance from each sample point to the study area boundary, sorts sample points on the distance to boundary, and builds a kd-tree spatial indexing data structure on the sample points. The control routine then allocates space in device memory using *cudaMalloc()* and transfers the sorted sample points, study area raster, and kd-tree

to device memory using *cudaMemCpy()*. It also allocates space in device memory for necessary ancillary data structures (e.g., an array to hold edge correction factors). The kernels running on GPU threads dynamically update these ancillary data structures while running the algorithm. The final densities computed at cells are copied back to host memory from device memory using *cudaMemCpy()*. The major kernels implemented for the GPU-enabled adaptive KDE algorithm are presented as follows.

4.3.1 Kernel for computing edge correction factors on GPU

Computing the edge correction factor for a sample point is independent from any other sample points. The computation for a sample point is dispatched to a specific GPU thread based on sample point ID and thread ID. A *kernel* was implemented to compute edge correction factors for sample points (Figure 3). Upon kernel invocation, edge correction factors of all sample points are computed 'concurrently' on GPU threads. The optimization described in Section 3.1 was implemented in this kernel, which was used for computing edge correction factors with both fixed and adaptive bandwidths.

[Figure 3 is about here]

Figure 3. Pseudo code of the kernel for computing edge correction factors for sample points.

It is worth noting that the execution time of a warp is determined by the slowest thread in that warp as GPU threads are executed in warps. If the workload of the threads in a warp varies wildly (i.e., thread divergence), warp execution will be slowed down. When computing edge correction factors, threads that correspond to sample points closer to the boundary likely have heavier load than threads processing sample points farther away from the boundary. This causes thread divergences if the distances to the boundary from sample points corresponding to threads in a warp vary significantly. In this article, the sample points are pre-sorted on distances to the

boundary. Threads in the same warp have approximately the same workload and therefore such potential thread divergences are mitigated.

Execution configuration for launching this kernel was determined using:

$$gridDim.x = gridDim.y = \lceil \sqrt{N\_blocks} \rceil \quad (10)$$

in which $gridDim.x$ and $gridDim.y$ are the number of rows and columns of blocks in a grid, and

$$N\_blocks = \lceil n/THREADS\_PER\_BLOCK \rceil \quad (11)$$

where *N_blocks* is the total number of blocks. n is the number of sample points, $THREADS\_PER\_BLOCK$ is the number of threads per block (usually it is a multiple of 32 and no larger than 1024), and $\lceil \cdot \rceil$ is a ceiling function. The calculated edge correction factors are saved in device memory and used by other kernels that are launched later (e.g., computing density estimates at sample points).

4.3.2 Kernel for computing density estimates at sample points on GPU

A kernel that implemented the optimization described in Section 3.2 was used to compute inclusive density estimates at sample points (Figure 4). The computation associated with a sample point (i.e., find sample points that are no more than *C* times its bandwidth using kd-tree, and then compute its contribution to the inclusive densities at these sample points) is dispatched to a corresponding GPU thread. Note that the atomic operation *atomicAdd* is used to add the density contribution of a sample point to the cumulative densities of other sample points. It serializes the updates of multiple threads on density estimate of the same sample point. Although it may introduce extra overhead, this is necessary for the parallel algorithm to obtain correct density estimates. By the completion of this kernel, inclusive density estimates at all sample points are computed. Execution configuration for launching this kernel was also determined using equation

(10) and equation (11). The computed inclusive densities were later used in the kernel for computing spatially adaptive bandwidths (this simple kernel is not presented here due to length limit).

[Figure 4 is about here]

Figure 4. Pseudo code of the kernel for computing inclusive density estimates at sample points.

Another kernel was implemented to compute the exclusive densities by subtracting the density contribution of a sample point to itself from the inclusive density at that point. The computed exclusive densities were later used in a kernel for computing likelihood of the sample points. These two simple kernels were not presented here due to length limit.

4.3.3 Kernel for computing density surface on GPU

Computing density estimates at each cell in the study area requires two nested traversals. The outer traversal goes through each cell in the study area. For each cell traversed, another inner traversal goes over all sample points to compute the density contribution from each sample point to the density at that cell (Figure 5). The computation associated with computing density at a cell is dispatched to a corresponding GPU thread.

[Figure 5 is about here]

Figure 5. Pseudo code of the kernel for computing density estimates over the study area.

Execution configuration for launching this kernel was determined in a similar way as using equation (10) and (11), with the only difference that one needs to replace n (i.e., number of points) with m (i.e., number of cells) in equation (11). Upon the completion of this kernel, computed densities at cells in device memory were copied back to host memory and were written to an external file by the CPU-side routine.

5. Experiments

5.1 Experiment design

5.1.1 Overall design

Experiments were designed to evaluate three aspects of the implemented *adaptive* KDE algorithm for spatial point pattern analysis: correctness, effectiveness of the optimizations, and efficiency and scalability of the GPU-parallel algorithm. For comprehensive evaluations, besides the GPU-parallel algorithm implemented as described in Section 4, a sequential version of the algorithm that runs on a single CPU core and an Open Multi-Processing (OpenMP)-parallel version of the algorithm that runs on multiple CPU cores were also implemented. Each algorithm version was implemented at three different optimization levels: no optimization (i.e., naïve implementation), partial optimization (avoiding re-computing edge correction factors), and full optimization (avoiding re-computing edge correction factors plus using kd-tree to speedup density estimation at sample points). Moreover, each algorithm version was augmented to support three modes of KDE: *fixed* KDE with bandwidth determined using 'rule-of-thumb' algorithm (Fotheringham et al. 2000, p149), *fixed* KDE with bandwidth determined using cross-validation based on maximum likelihood criterion (Brunsdon 1995), and *adaptive* KDE with spatially adaptive bandwidths determined using *Brunsdon's algorithm*.

Correctness of the parallel algorithms was tested by comparing their results to those of the corresponding sequential algorithms. Effectiveness of the optimizations was evaluated by comparing execution time (i.e., elapsed time, or wall clock time) of the optimized algorithms with that of the non-optimized algorithms. *Efficiency* of the GPU-parallel algorithm was evaluated by comparing execution time of the GPU-parallel algorithm with that of the sequential algorithm and the OpenMP-parallel algorithm. Acceleration factor (*AF*) was used as an indicator of how much performance improvement a target algorithm achieves compared against a base algorithm (Zhang et al. 2016; Tang et al. 2015):

$$AF = T_{base}/T_{target} \quad (12)$$

where T_{base} and T_{target} are the execution time of the base algorithm and the target algorithm. *Scalability* of the GPU-parallel algorithm was evaluated by testing performance of the GPU-parallel algorithm (in terms of AF) on point pattern analysis tasks of various problem size (e.g., different number of points and number of cells). Recorded execution time included execution time on all stages of the algorithm except I/O (i.e., read data from disk and write data to disk). The recorded execution time for the GPU-parallel algorithm included time spent on exchanging data between the host memory and the GPU device memory as that is part of the computation cost of the GPU-parallel algorithm. The recorded execution time thus allows fair and holistic comparison of the performance of different algorithms. Reported time was averaged on 10 runs of the algorithm. This should suffice to provide a relatively accurate estimation of the average execution time because devoted computing resources were used to run the experiments. However, it is prohibitively time-consuming to run some experiments many times as each run takes tens of hours. For such experiments (e.g., running the non-optimized sequential algorithm on the test data set in Section 5.3), 5 runs were conducted. The mean and standard deviation of the execution time of the multiple runs were reported for each experiment. Across all experiments, the ratio between the standard deviation and the mean execution time was smaller than 5%, indicating that the recorded execution time was relatively stable. Acceleration factor was computed based on the mean execution time. The OpenMP-parallel algorithms run on 32 CPU cores (i.e., threads) if not otherwise specified.

Moreover, the GPU-parallel algorithm was used to conduct spatial point pattern analysis on a real-world data set to demonstrate how the GPU-accelerated *adaptive* KDE approach can facilitate efficient spatial point pattern analysis in real-world applications.

5.1.2 Data sets

The California Redwood seedlings data set used to demonstrate the original *Brunsdon's algorithm* (Brunsdon 1995) was also employed to test correctness of the algorithms implemented in this article. The Redwood data set available from the '*spatstat'* R package (Baddeley et al. 2015) contains 62 California Redwood seedling locations (Figure 6a). The expected results of point pattern analysis on this data set are documented in Brunsdon (1995).

Effectiveness of the optimizations and efficiency and scalability of the GPU-parallel algorithm were evaluated on simulated random point patterns. A point pattern that is a realization of the Matérn Cluster Process was simulated using the *rMatClust* command in 'spatstat' R package (Baddeley et al. 2015). 1,010,518 points were generated within a unit square by setting the intensity of the Poisson process of cluster centers, radius of the clusters, and mean number of points per cluster to 200, 0.1 and 5,000 respectively (parameters for the *rMatClust* command). Point patterns of various sizes were generated by randomly sampling different number of points from this simulated point pattern (Figure 6b shows one example). The unit square study area was discretized into rasters of various numbers of cells by setting different cell sizes. Point patterns of various numbers of points in combination with study areas of various numbers of cells composed point pattern analysis tasks of various problem sizes. Scalability of the GPU-parallel algorithm was evaluated on these point pattern analysis tasks of various problem sizes.

The eBird checklists data set (Sullivan et al. 2009; Wood et al. 2011) was used to demonstrate a real-world application of the GPU-accelerated *adaptive* KDE approach for spatial point pattern analysis. eBird checklists are composed of records of locations where birders conducted birding activities. Records in the contiguous United States over June, July, and August in 2012 were extracted (eBrid 2016). After removing duplicate locations (e.g., locations with the same latitude, longitude), a point pattern consisting of 78,977 unique checklists locations was constructed (Figure 6c). The contiguous United States study area was discretized into a raster of 4,548 × 2,847 = 12,948,156 cells (1 km cell size).

[Figure 6 is about here]

Figure 6. Test data sets used in evaluation experiments. (a) The Redwood data set for testing correctness (n = 62). (b) One of the simulated point pattern (n = 50,000). (c) The eBird checklists data set (n = 78,977).

5.1.3 Computing environment

Experiments were conducted on a computing server running the CentOS 7.2 operating system. CPUs used for CPU-enabled computing are AMD Opteron 6274 quad eight-core processor with 2.0 GHz of clock frequency (80KB L1 cache, 2048KB L2 cache, 6144KB L3 cache) and 128 GB of memory. The GPU used for GPU-enabled computing is GeForce GTX 480 based on Fermi architecture (480 CUDA cores with 1.4 GHz of clock speed, 1.5GB of global memory, 177.4 GB per second memory bandwidth).

5.2 Correctness of the algorithms

The GPU-parallel algorithms and the sequential algorithms produced the same results of point pattern analysis on the Redwood data set. The cross-validated optimal fixed bandwidth and the (h, α) values for computing optimal adaptive bandwidths obtained using the GPU-parallel algorithms were fairly close to those reported in Brunsdon (1995). The slight differences were mostly attributed to the addition of edge correction in our implementations. The estimated density surfaces (Figure 7) were consistent with those reported in Brunsdon (1995). Obviously, with the 'rule-of-thumb' fixed bandwidth, the estimated density surface was largely over-smoothed and thus failed to show any local variation of tree density (Figure 7a). The density surface estimated with the cross-validated fixed optimal bandwidth did capture local variation of tree density (Figure 7b). But it had two limitations: a tendency of 'spillage' of density near the boundary of clusters of trees; and 'spiky' estimates in less densely clustered areas of trees. The density surface

estimated with spatially adaptive optimal bandwidths (Figure 7c) was most reasonable. There was a greater tendency for sharp edges on the density surface. In areas where a dense cluster of trees is neighbored by areas with no trees, smaller bandwidths were used to reduce the 'spillage' of density. In areas of sparse trees, larger bandwidths were used to avoid 'spiky' density estimates (Figure 7d). The *adaptive* KDE was the preferred approach for spatial point pattern analysis of the Redwood data set, which is consistent with Brunsdon (1995).

[Figure 7 is about here]

Figure 7. Spatial point pattern analysis on the Redwood data set. Density surface estimated with (a) 'rule-of-thumb' fixed bandwidth ($h = 0.123$), (b) cross-validated fixed optimal bandwidth ($h = 0.045$), and (c) spatially adaptive optimal bandwidths ($h = 0.035$, $\alpha = 1.47$). (d) The adaptive optimal bandwidths used to estimate density surface in (c).

5.3 Effectiveness of the optimizations

A data set composed of a point pattern of 50000 points within a unit square study area discretized into 400 × 400 = 160,000 cells was used to evaluate the effectiveness of the optimizations. Profiling of the sequential algorithms reveals that about 83% of the run time was spent on edge correction factors computation in the naïve implementation of the algorithm (no optimization). The percentage was reduced to about 11% in the partially optimized implementation (avoiding re-computing edge correction factors). In terms of execution time (Table 1), avoiding re-computing edge correction factors accelerated the sequential and parallel algorithms by a factor of about 6.

Avoiding re-computing edge correction factors and using kd-tree indexing to speed up density estimation at sample points significantly accelerated the sequential and parallel algorithms by a factor of 28 to 75. The optimizations brought more acceleration to the sequential algorithm than to the parallel algorithms. This might be a result of the additional synchronization overhead

introduced in implementing the optimizations for parallel algorithms (e.g., using atomic operations to update density estimates).

Table 1. Performance of the non-optimized and optimized adaptive KDE algorithms on the test data set (n=50,000, m=160,000) (execution time was averaged over 10 runs, except the sequential non-optimized algorithm averaged over 5 runs; time unit in seconds).

[Table 1 is about here]

Overall, the two optimizations effectively sped up the algorithms by a factor of tens and thus could greatly increase the capability of the adaptive KDE approach for point pattern analysis on very large data sets. The algorithms with full optimization were used in the remaining evaluations.

5.4 Efficiency and scalability of the GPU-parallel algorithm

5.4.1 Efficiency

5.4.1.1 Comparison with sequential algorithm and OpenMP-parallel algorithm

Using adaptive KDE for point pattern analysis on the test data set (n = 50,000, m = 160,000), it took 2.13 seconds for the GPU-parallel algorithm, 63.63 seconds for the OpenMP-parallel algorithm and 1459.58 seconds (about 24 minutes) for the sequential algorithm (Table 1). The GPU-parallel algorithm achieved an acceleration factor of 685.2 over the sequential algorithm and 29.9 over the OpenMP-parallel algorithm, respectively (Table 3). The GPU-parallel algorithm significantly accelerated the *adaptive* KDE approach for point pattern analysis more than the OpenMP-parallel algorithm did.

The OpenMP-parallel algorithm (running on 32 CPU cores) achieved a speedup ratio (i.e., AF of OpenMP algorithm over sequential algorithm) of only 22.9 over the sequential algorithm on the test data set (Table 3), which was far below the ideal theoretical speedup of 32. This was partly due to the overhead required to manage threads and shared resources for the OpenMP algorithm.

Moreover, the OpenMP-parallel algorithm did not scale very well with respect to the number of CPU cores utilized (Table 2; Figure 8). As the number of CPU cores utilized increased, the gap between the speedup ratio achieved by the OpenMP-parallel algorithm and the ideal linear speedup was widened (Figure 8a) and the parallel efficiency decreased from 1.0 down to around 0.75 (Figure 8b) resulting from expensive maintenance of coherent memories and caches across a large number of CPU cores, known as 'false sharing problem' in multiprocessor programming (Torrellas et al. 1994; Zhang et al. 2016). Thus, it is unrealistic to expect that the OpenMP-parallel algorithm could match the performance of the GPU-parallel algorithm by simply increasing the number of CPU cores.

Table 2. Performance of the OpenMP-parallel adaptive KDE algorithms on the test data set (n=50,000, m=160,000) using different number of CPU cores (execution time was averaged over 10 runs; parallel efficiency = speedup ratio / number of CPU cores; time unit in seconds).

[Table 2 is about here]

[Figure 8 is about here]

Figure 8. Performance of the OpenMP-parallel algorithm for adaptive KDE on the test data set (n = 50,000, m = 160,000): (a) speed up ratio and (b) parallel efficiency - defined as speedup ratio divided by number of CPU cores.

5.4.1.2 Comparison with fixed KDE

The GPU-parallel algorithm of the *adaptive* KDE approach for point pattern analysis was compared to that of the *fixed* KDE approach on the test data set (n = 50,000, m = 160,000) (Figure 9). The GPU-parallel algorithm of both approaches was also compared to that of the OpenMP-parallel algorithm (Table 3). The *adaptive* KDE approach for point pattern analysis was indeed much more computationally expensive than the *fixed* KDE approach. Using either the

fixed KDE or the *adaptive* KDE approach for point pattern analysis, the GPU-parallel algorithm accelerated point pattern analysis tasks by a factor of greater than 680 whilst the OpenMP-parallel algorithm sped up the tasks by a factor of less than 25. The GPU-parallel algorithm was more than 29 times faster than the OpenMP-parallel algorithm. It demonstrated that GPU-enabled algorithms not only significantly accelerated the *adaptive* KDE approach, but also could greatly speed up the general KDE approach for point pattern analysis.

[Figure 9 is about here]

Figure 9. Spatial point pattern analysis on the test data set (n=50,000, m=160,000). Density surface estimated with (a) 'rule-of-thumb' fixed bandwidth ($h = 0.025$), (b) cross-validated fixed optimal bandwidth (h = 0.017), and (c) spatially adaptive optimal bandwidths (h = 0.010, α = 1.088). The density surface estimated using *adaptive* KDE captured more local density variations.

Table 3. Performance of the GPU-parallel, OpenMP-parallel, and sequential algorithm to conduct point pattern analysis on the test data set (n = 50,000, m = 160,000) (execution time was averaged over 10 runs; time unit in seconds).

[Table 3 is about here]

5.4.1.3 Impact of block size

Execution configuration in kernel invocations has impacts on performance of GPU-parallel algorithms (NVIDIA 2016). As discussed in Section 4.3, the number of threads per block (i.e., block size) determines the execution configurations for launching kernels. The performance of the GPU-parallel algorithm for *adaptive* KDE was evaluated on the test data set (n = 50,000, m = 160,000) with different block sizes (Figure 10). The performance fluctuates and was poorer when block size was too large or too small. The best performance was reached using 256 threads per block.

[Figure 10 is about here]

Figure 10. The impact of number of threads per block on performance of the GPU-parallel algorithm for *adaptive* KDE on the test data set (n = 50,000, m = 160,000) (execution time was averaged over 10 runs; error bars indicate standard deviations).

It should be noted, however, the block size leading to best performance of a GPU-parallel algorithm is highly dependent upon the underlying GPU hardware, the nature of the algorithm, and the characteristics of data set. Thus, an optimal block size should be mostly determined experimentally on a case-by-case basis. In our case, 256 threads per block seemed to be a reasonable configuration that guaranteed a good performance of the GPU-parallel algorithm. This block size was thus used in determining execution configurations for the GPU-parallel algorithm in all experiments.

5.4.2 Scalability

5.4.2.1 Impact of the number of sample points

The impact of the number of sample points on performance of the GPU-parallel algorithm was evaluated by conducting point pattern analysis on the simulated point patterns of different sizes (Figure 11). As expected, execution time of the GPU-parallel algorithm increased as the number of points increased (Figure 11a). Execution time of the GPU-parallel algorithm was compared to that of the OpenMP-parallel algorithm to compute acceleration factors (comparing the sequential algorithm is unrealistic as it takes prohibitively long to complete the point pattern analysis tasks). Across all test data sets, the GPU-parallel algorithm was about 20 to 45 times faster than the OpenMP-parallel algorithm. The acceleration factor achieved by the GPU-parallel algorithm over the OpenMP-parallel algorithm decreased as the number of points increased (Figure 11b). The decreasing GPU-over-OpenMP acceleration factor might be caused by the higher synchronization overhead (e.g., *atomicAdd* operations) across a large number of GPU threads (the number of GPU

threads was the same as the number of points for the GPU-parallel algorithm; whilst there were always 32 CPU threads for the OpenMP-parallel algorithm). Yet the GPU-parallel algorithm was still more than 20 times faster than the OpenMP-parallel algorithm running on 32 CPU cores for point pattern analysis tasks with a large number of points.

[Figure 11 is about here]

Figure 11. Impact of number of points on the GPU-parallel algorithm for adaptive KDE. (a) Execution time of the GPU-parallel algorithm (averaged over 10 runs; error bars indicate standard deviations). (b) Acceleration factor of the GPU-parallel algorithm over the OpenMP-parallel algorithm. Experiments were run with study areas of various numbers of cells (i.e., 900 × 900 = 810,000, 1000 × 1000 = 1,000,000, and 1,100 × 1,100 = 1,210,000).

5.4.2.2 Impact of the number of cells

The impact of the number of cells on performance of the GPU-parallel algorithm was evaluated by conducting point pattern analysis on 500,000 point in study areas discretized into various numbers of cells (Figure 12). As expected, execution time of the GPU-parallel algorithm increased as the number of cells increased (Figure 12a). Across all test data sets, the GPU-parallel algorithm was about 37 to 50 times faster than the OpenMP-parallel algorithm. The acceleration factor achieved by the GPU-parallel algorithm over the OpenMP-parallel algorithm increased as the number of cells increased (Figure 12b), indicating that the GPU-parallel algorithm scales very well on point pattern analysis tasks of a large number of cells.

[Figure 12 is about here]

Figure 12. Impact of number of cells on the GPU-parallel algorithm for adaptive KDE. (a) Execution time of the GPU-parallel algorithm (averaged over 10 runs; error bars indicate standard deviations). (b) Acceleration factor of the GPU-parallel algorithm over the OpenMP-parallel

algorithm. Experiments were run with various numbers of points (i.e., 400,000, 500,000, and 600,000).

5.5 GPU-accelerated point pattern analysis on a real-world data set

The GPU-enabled algorithm was applied for point pattern analysis on the eBird checklists data set (n = 78,977, m = 12,948,156) using both the *adaptive* KDE approach and the *fixed* KDE approach (Figure 13). Density surface produced by *fixed* KDE with 'rule-of-thumb' bandwidth showed several large-scale high density birding hotspots over the United States. But it failed to reveal fine-scale density variations as the overly large bandwidth over-smoothed birding locations (Figure 13a and d). *Fixed* KDE with cross-validated optimal bandwidth produced a density surface that was telling of local density variations (Figure 13b and e). Yet the density surface produce by *adaptive* KDE was the most expressive among the three in terms of revealing fine-scale patterns in the data (Figure 13c and f). With spatially adaptive bandwidth, *adaptive* KDE was able to pinpoint fine-scale high density birding hotspots more precisely.

[Figure 13 is about here]

Figure 13. Point pattern analysis on the eBird checklists data set (n = 78,977, m = 12,948,156). Density surface estimated with (a) 'rule-of-thumb' fixed bandwidth (h = 84.4 km), (b) cross-validated fixed optimal bandwidth (h = 24.5 km), and (c) spatially adaptive optimal bandwidths (h = 4.2 km, α = 1.21). (d), (e), and (f) are the zoom-in maps of a small area on the east coast of United States.

The real-world study again demonstrated that the *adaptive* KDE approach for point pattern analysis was much more computationally intensive than the *fixed* KDE approach (Table 4). It took about 1.5 hours to complete point pattern analysis on the eBird data set using the OpenMP-parallel *adaptive* KDE algorithm (running on 32 CPU cores). Performing such an analysis using

the sequential algorithm would be extraordinarily time-consuming (it would take about 30 hours estimated based on a speedup ratio of 20 for OpenMP-parallel algorithm). Yet using the GPU-parallel *adaptive* KDE algorithm took only 5.8 minutes. For either *adaptive* KDE or *fixed* KDE, the GPU-parallel algorithm achieved an acceleration factor of greater than 15 over the OpenMP-parallel algorithm (an estimated acceleration factor of about 15 × 20 = 300 would be achieved over the sequential algorithm). Given the significant acceleration brought by the GPU-enabled algorithm, efficient point pattern analysis on large data sets such as the eBird checklists can be routinely conducted.

Table 4. Execution time of the algorithms for conducting point pattern analysis on the eBird checklists data set (n = 78,977, m = 12,948,156) using the *adaptive* KDE approach and the *fixed* KDE approach (execution time was averaged over 10 runs; time unit in seconds).

[Table 4 is about here]

6. Discussion

6.1 Impact of the *C* value

The value of C could impact the accuracy of the estimated density surface and the execution time of the algorithms (Section 3). To be consistent with the original Brunsdon's algorithm (Brunsdon 1995), C=3 was used for the experiments in this article. The impact of different C values on the results was examined by running the GPU-parallel adaptive KDE algorithm on the test data set (n = 50,000, m = 160,000) under different values of C. The density surfaces estimated under various C values were compared to that estimated without setting a distance threshold to exclude raster cells or sample point in computation (equivalent to setting a very large C value) (Figure 14a; this was treated as an accurate density surface that reveals the underlying point pattern). Results were shown on Figure 14.

[Figure 14 is about here]

Figure 14. Impact of C value on the estimated density surface and on performance of the algorithm. (a) Density surface estimated without using $C \times h$ as a distance threshold to exclude raster cells or sample points in computation. (b) through (f) show the density surfaces estimated using C=1, C=2, C=3, C=6, and C=10. The estimated parameters (i.e., h and α) and the corresponding execution time (average of 10 runs and the standard deviation) were also shown on each figure.

Under C=1 or C=2, the algorithm failed to estimate a density surface that could reveal the underlying point pattern. The algorithm either did not converge to a positive α value or did not find a proper α value after going through 30 iterations (the predefined maximum number of iterations). At a location that is h_i or $2 \times h_i$ distance from the center sample point, the density value (height of Gaussian kernel surface) is still 60.7% or 13.5% of the height at the center point (the ratio is 1.1% for $3 \times h_i$). The kernel mass over the h_i or $2 \times h_i$ radius circular neighborhood of the center point is only 39.3.9% or 86.5% of the total kernel mass (the ratio is 98.9% for $3 \times h_i$). Setting C=1 or C=2 introduces significant errors in computing edge correction factors and densities. Thus, the estimated density surfaces were highly inaccurate (the density surfaces on Figure 14b and c were very different from the accurate density surface on 14a).

Under C=3 or larger C values (e.g., C=6, C=10), the algorithm converged to a positive α value in fewer than 30 iterations and estimated similar density surfaces (Figure 14d, e, and f) that were consistent with the accurate density surface. Thus, one should always set a C value that is no smaller than 3 (Section 3.1). With a value of C greater than or equal to 3, the estimated density surface would better resemble the accurate one with an increasing C value, but the execution time of the algorithm increased significantly as well (Figure 14d, e, f, and a). It should be noted that the execution time under $C = 1$ was longer than that under $C = 3$ because the algorithm went

through a larger number of iterations (this is also why the execution time under $C = 2$ was longer than under $C = 6$). Overall, C=3 was indeed a reasonable trade-off between computation cost and accuracy, and could be used as the default setting (Brunsdon 1995). The GPU-enabled algorithm can effectively accelerate the adaptive KDE by a factor of tens or even hundreds compared to the sequential or OpenMP-parallel implementations given problems of the same sizes (Section 5.3). But still the GPU-enabled algorithm expectedly takes longer execution time on larger-size problems (Section 5.4). Thus, $C = 3$ is the recommended setting for large-size problems. In cases where execution time is less of a concern (e.g., small-size problems, or one is willing to wait longer), a C value that larger than 3 can be adopted to obtain more accurate density surfaces.

6.2 Cost associated with implementing the optimizations

There was computation cost associated with implementing the optimizations (Section 3), which includes computing distances from the sample points to the boundary, sorting the sample points based on the distances, and building k-d tree on the sample points. The distances were computed in parallel in the GPU-parallel algorithm. Sorting the points and building k-d tree were implemented with sequential routines. When performing adaptive KDE on the test data set (n = 50,000, m = 160,000), these computations took up 11% of the total execution time of the GPU-parallel algorithm (3.3% on sorting points including computing distances first). The percentage decreased on problems of larger sizes. These computations took up no more than 10% (about the same amount of time on sorting points and on building k-d tree) of the total execution time for experiments in Section 5.4.2, and only 1.4% (1.3% on sorting points) of the total execution time for the experiment in Section 5.5.

Nevertheless, experiment results have shown that the computation cost associated with implementing the optimizations were well paid off by the significant acceleration brought by the optimizations (Section 5.3; Table 1).

6.3 Impact of data exchange between host and GPU

Using the GPU-parallel adaptive KDE algorithm for point pattern analysis on the test data set (n = 50,000, m = 160,000), data exchange between the host memory and the GPU device memory took up only 0.03% of the total execution time (i.e., 0.685 milliseconds out of 2.1 seconds). On problems of larger sizes, the percentage further decreased to negligible (e.g., data exchange took up less than 0.01% of the total execution time for experiments in Section 5.4.2 and 5.5).

If the recorded execution time of the GPU-parallel algorithm did not include time spent on data exchange between host and GPU and it was used to compute the acceleration factor achieved by the GPU-parallel algorithm (equation 12), the resulting acceleration factor would be higher than those presented in Table 3 and 4. However, for all experiments reported in this article, data exchange time was included in the recorded execution time for the GPU-parallel algorithm to compute the acceleration factor. This is because data exchange is part of the computation cost for the GPU-parallel algorithm, and including data exchange time allows fair and holistic performance comparison between the GPU-parallel algorithm and other algorithms.

7. Conclusions

This article presented a GPU-accelerated *adaptive* KDE algorithm for efficient spatial point pattern analysis on large data sets. Optimizations were designed to reduce complexity and therefore speed up the algorithm for determining spatially adaptive optimal bandwidths for *adaptive* KDE. GPU computing was then exploited to further accelerate the optimized *adaptive* KDE algorithm for point pattern analysis. Computationally intensive tasks involved in bandwidths determination were dispatched to GPUs and conducted in a massively parallel fashion. Experimental results demonstrated that the proposed optimizations effectively sped up the algorithm by a factor of tens. The GPU-enabled algorithm accelerated point pattern analysis tasks on large data sets by a factor of *hundreds* compared to the sequential version of the algorithm.

Compared to an OpenMP-parallel version of the algorithm that leverages computing power on multi-CPUs, the GPU-accelerated algorithm was still *tens* of times faster. The GPU-accelerated algorithm scales reasonably well on large-scale point pattern analysis that involves a large number of points, over large areas, or at high spatial resolution.

Spatial big data are now becoming ubiquitous (Shekhar et al. 2012; Wu et al. 2014; Yang et al. 2016). As the volume of spatial data is increasingly growing, it is urgent to address the computational challenges associated with spatial big data analytics to keep pace with the ever-faster-growing data volume (Evans et al. 2013; Wu et al. 2014; Yang et al. 2016). Computational tools exploiting high-performance computing resources are under development to deal with the computational challenges posed by spatial big data processing and analysis (Wang 2010; Yang et al. 2010; Aji et al. 2013; Evans et al. 2013; Wang 2013; Pijanowski et al. 2014; Wu et al. 2014; Tang et al. 2015; Yang et al. 2016; Zhang et al. 2016). These computational tools are collectively establishing a geospatial computational toolbox that enables fast processing and analysis of spatial big data and could greatly facilitate geographic knowledge discovery from spatial big data.

Many spatial big data can be perceived as point event data and point pattern analysis as a mainstream analytical tool for revealing hidden patterns in spatial point data is paramount to support geographic knowledge discovery. Given the significant acceleration brought by the GPU-enabled algorithm presented in this article, *adaptive* KDE can be performed efficiently for point pattern analysis on spatial big point data. Point pattern analysis tasks that once were computationally prohibitive can now be conducted routinely to identify interesting patterns in a timely manner. The GPU-accelerated *adaptive* KDE approach contributes to the computational toolbox for spatial big data analytics.

Acknowledgements

The work reported here was supported by grants from National Natural Science Foundation of China (Project No.: 41431177), National Basic Research Program of China (Project No.: 2015CB954102), Natural Science Research Program of Jiangsu (14KJA170001), PAPD, National Key Technology Innovation Project for Water Pollution Control and Remediation (Project No.: 2013ZX07103006). Supports to A-Xing Zhu through the Vilas Associate Award, the Hammel Faculty Fellow Award, the Manasse Chair Professorship from the University of Wisconsin-Madison, and the "One-Thousand Talents" Program of China are greatly appreciated. We thank the anonymous reviewers whose comments helped improve this manuscript. Thanks to Dr. Dan Negrut, Mr. Ang Li, and Mr. Colin Vanden Heuvel at the Wisconsin Applied Computing Center (WACC) at the University of Wisconsin-Madison for introducing GPU programming and for providing computing platform to run experiments reported in this study, and to Mr. Jack John Keel for proof reading the manuscript.

References

Abramson, I.S., 1982. On bandwidth variation in kernel estimates-a square root law. *The annals of Statistics*, 10(4), pp.1217–1223.

Aji, A. et al., 2013. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11), pp.1009–1020.

Andrzejewski, W., Gramacki, A. & Gramacki, J., 2013. Graphics processing units in acceleration of bandwidth selection for kernel density estimation. *International Journal of Applied Mathematics and Computer Science*, 23(4), pp.869–885.

Baddeley, A. et al., 2015. Package "spatstat."

Bentley, J.L. & Friedman, J.H., 1979. Data Structures for Range Searching. *ACM Comput. Surv.*, 11(4), pp.397–409.

Breiman, L., Meisel, W. & Purcell, E., 1977. Variable kernel estimates of multivariate densities. *Technometrics*, 19(2), pp.135–144.

Brunsdon, C., 1995. Estimating probability surfaces for geographical point data: An adaptive kernel algorithm. *Computers & Geosciences*, 21(7), pp.877–894.

Burt, J.E., Barber, G.M. & Rigby, D.L., 2009a. *Elementary statistics for geographers*, Guilford Press.

Burt, J.E., Barber, G.M. & Rigby, D.L., 2009b. *Elementary statistics for geographers*, New York: Guilford Press.

Carlos, H. a et al., 2010. Density estimation and adaptive bandwidths: a primer for public health practitioners. *International journal of health geographics*, 9, p.39.

Diggle, P., 1985. A Kernel Method for Smoothing Point Process Data. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 34(2), pp.138–147.

eBrid, 2016. eBird Reference Dataset. Available at: http://www.ccs.neu.edu/home/mirek/papers/ebird-ref-data.pdf.

Epanechnikov, V., 1969. Non-Parametric Estimation of a Multivariate Probability Density. *Theory of Probability & Its Applications*, 14(1), pp.153–158.

Evans, M.R. et al., 2013. Enabling spatial big data via CyberGIS: challenges and opportunities. In S. Wang & M. Goodchild, eds. *CyberGIS: Fostering a New Wave of Geospatial Innovation and Discovery. Springer Book*. New York.

Fotheringham, A.S., Brunsdon, C. & Charlton, M., 2000. *Quantitative geography: perspectives on spatial data analysis*, Thousand Oaks, California: Sage.

Gatrell, A.C. et al., 1996. Spatial Point Pattern Analysis and Its Application in Geographical

Epidemiology. *Transactions of the Institute of British Geographers*, 21(1), pp.256–274.

Goodchild, M.F., 2007. Citizens as sensors: the world of volunteered geography. *Geojournal*, 69(4), pp.211–221.

Guan, Q. et al., 2016. A hybrid parallel cellular automata model for urban growth simulation over GPU/CPU heterogeneous architectures. *International Journal of Geographical Information Science*, 30(3), pp.494–514.

Guan, Q. & Clarke, K.C., 2010. A general-purpose parallel raster processing programming library test application using a geographic cellular automata model. *International Journal of Geographical Information Science*, 24(5), pp.695–722.

Hooke, R. & Jeeves, T.A., 1961. "Direct Search" Solution of Numerical and Statistical Problems. *J. ACM*, 8(2), pp.212–229.

Huang, Q. & Wong, D.W.S., 2015. Modeling and Visualizing Regular Human Mobility Patterns with Uncertainty : An Example Using Twitter Data Modeling and Visualizing Regular Human Mobility Patterns with Uncertainty : An Example Using Twitter Data. *Annals of the Association of American Geographers*, 105(6), pp.1179–1197.

Jones, M., 1990. Variable kernel density estimates and variable kernel density estimates. *Australian Journal of Statistics*, 32(March), pp.361–371.

Jones, M.C., Marron, J.S. & Sheather, S.J., 1996. Progress in data-based bandwidth selection for kernel density estimation. *Computational Statistics*, 11(3), pp.337–381.

Kakde, H.M., 2005. Range Searching using Kd Tree. *Florida State University*.

Kirk, D.B. & Hwu, W.W., 2012. *Programming massively parallel processors: a hands-on approach* 2nd ed., Newnes.

Law, R. et al., 2009. Ecoogical information from satial patterns of plants: Insights from point process theory. *Journal of Ecology*, 97(4), pp.616–628.

Longley, P.A. & Adnan, M., 2016. Geo-temporal Twitter demographics. *International Journal of Geographical Information Science*, 30(2), pp.369–389.

Luebke, D., 2008. CUDA: Scalable parallel programming for high-performance scientific computing. *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, Proceedings, ISBI*, pp.836–838.

Michailidis, P.D. & Margaritis, K.G., 2013. Accelerating Kernel Density Estimation on the GPU Using the CUDA Framework. *Applied Mathematical Sciences*, 7(30), pp.1447–1476.

Miecznikowski, J.C., Wang, D. & Hutson, A., 2010. Bootstrap MISE Estimators to Obtain Bandwidth for Kernel Density Estimation. *Communications in Statistics - Simulation and Computation*, 39(1986), pp.1455–1469.

Nickolls, J. et al., 2008. Scalable Parallel Programming with CUDA. *Queue*, 6(2), pp.40–53.

NVIDIA, 2016. CUDA C programming guide version 7.0. *NVIDIA Corporation, Santa Clara, CA*.

Osterman, A., Benedičič, L. & Ritoša, P., 2014. An IO-efficient parallel implementation of an R2 viewshed algorithm for large terrain maps on a CUDA GPU. *International Journal of Geographical Information Science*, 28(11), pp.2304–2327.

Pijanowski, B.C. et al., 2014. A big data urban growth simulation at a national scale: Configuring the GIS and neural network based Land Transformation Model to run in a High Performance Computing (HPC) environment. *Environmental Modelling & Software*, 51, pp.250–268.

Qin, C.-Z. et al., 2014. A strategy for raster-based geocomputation under different parallel

computing platforms. *International Journal of Geographical Information Science*, 28(11), pp.2127–2144.

Sain, S.R. et al., 1996. On Locally Adaptive Density Estimation. *Journal of the American Statistical Association*, 91(436), pp.1525–1534.

Shekhar, S. et al., 2012. 8. Spatial big-data challenges intersecting mobility and cloud computing. In *Proceedings of the Eleventh ACM International Workshop on Data Engineering for Wireless and Mobile Access,SIGMOD/PODS '12 International Conference on Management of Data Scottsdale, AZ, USA — May 20 - 24, 2012*. New York: ACM, pp. 1–6.

Shi, X., 2010. Selection of bandwidth type and adjustment side in kernel density estimation over inhomogeneous backgrounds. *International Journal of Geographical Information Science*, 24(5), pp.643–660.

Silverman, B.W., 1986. *Density Estimation for Statistics and Data Analysis*, London, UK: Chapman and Hall.

Sullivan, B.L. et al., 2009. eBird: A citizen-based bird observation network in the biological sciences. *Biological Conservation*, 142(10), pp.2282–2292.

Tang, W., 2013. Parallel construction of large circular cartograms using graphics processing units. *International Journal of Geographical Information Science*, 27(11), pp.2182–2206.

Tang, W., Feng, W. & Jia, M., 2015. Massively parallel spatial point pattern analysis: Ripley's K function accelerated using graphics processing units. *International Journal of Geographical Information Science*, 29(3), pp.412–439.

Torrellas, J., Lam, M.S. & Hennessy, J.L., 1994. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6), pp.651–663.

Wang, S., 2010. A CyberGIS framework for the synthesis of Cyberinfrastructure, GIS, and spatial

analysis. *Annals of the Association of American Geographers*, 100(February 2015), pp.535–557.

Wang, S., 2013. CyberGIS: blueprint for integrated and scalable geospatial software ecosystems. *International Journal of Geographical Information Science*, 27(11), pp.2119–2121.

Wood, C. et al., 2011. eBird: engaging birders in science and conservation. *PLoS Biology*, 9(12), p.e1001220.

Worton, B.J., 1989. Kernel Methods for Estimating the Utilization Distribution in Home-Range Studies. *Ecology*, 70(1), p.164.

Wright, D.J. & Wang, S., 2011. The emergence of spatial cyberinfrastructure. *Proceedings of the National Academy of Sciences of the United States of America*, 108(14), pp.5488–5491.

Wu, H., Zhang, T. & Gong, J., 2014. GeoComputation for Geospatial Big Data. *Transactions in GIS*, 18(S1), pp.1–2.

Xie, Z. & Yan, J., 2008. Kernel Density Estimation of traffic accidents in a network space. *Computers, Environment and Urban Systems*, 32(5), pp.396–406.

Yang, C. et al., 2016. Cloud computing and big data: Opportunities and challenges for innovation. *Computers, Environment and Urban Systems*.

Yang, C. et al., 2010. Geospatial Cyberinfrastructure: Past, present and future. *Computers, Environment and Urban Systems*, 34(4), pp.264–277.

Yang, C. et al., 2011. Spatial cloud computing: how can the geospatial sciences use and help shape cloud computing? *International Journal of Digital Earth*, 4(February 2015), pp.305–329.

Yang, C.P., 2016. Geospatial cloud computing and big data. *Computers, Environment and Urban*

Systems, p.22030.

Zhang, G. et al., 2016. Enabling point pattern analysis on spatial big data using cloud computing: Optimizing and accelerating Ripley's K function. *International Journal of Geographical Information Science*.

Zhang, J. & You, S., 2013. High-performance quadtree constructions on large-scale geospatial rasters using GPGPU parallel primitives. *International Journal of Geographical Information Science*, 27(11), pp.2207–2226.

Zhao, Y., Padmanabhan, A. & Wang, S., 2013. A parallel computing approach to viewshed analysis of large terrain data using graphics processing units. *International Journal of Geographical Information Science*, 27(2), pp.363–384.

Zhou, C. et al., 2016. A Parallel Scheme for Large-scale Polygon Rasterization on CUDA-enabled GPUs. *Transactions in GIS*.

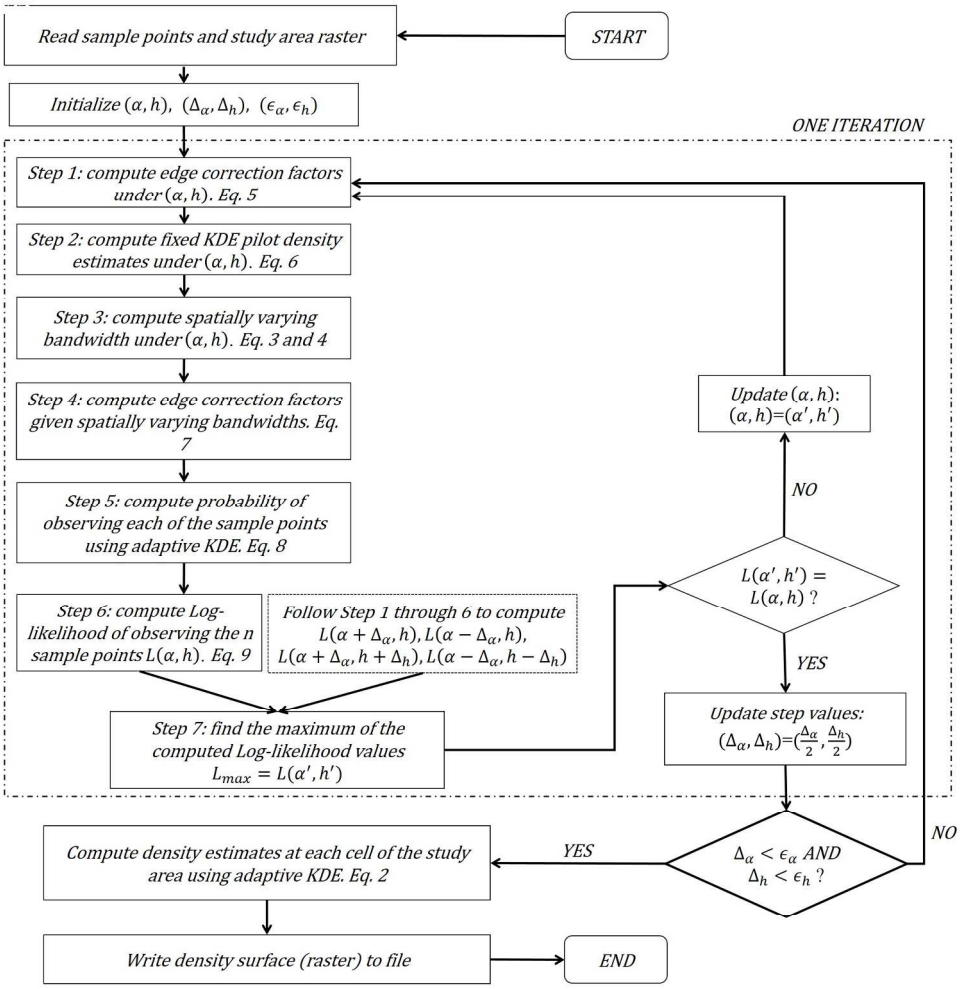

Figure 1. Flow chart of the sequential Brunsdon's algorithm for adaptive KDE.

189x194mm (300 x 300 DPI)

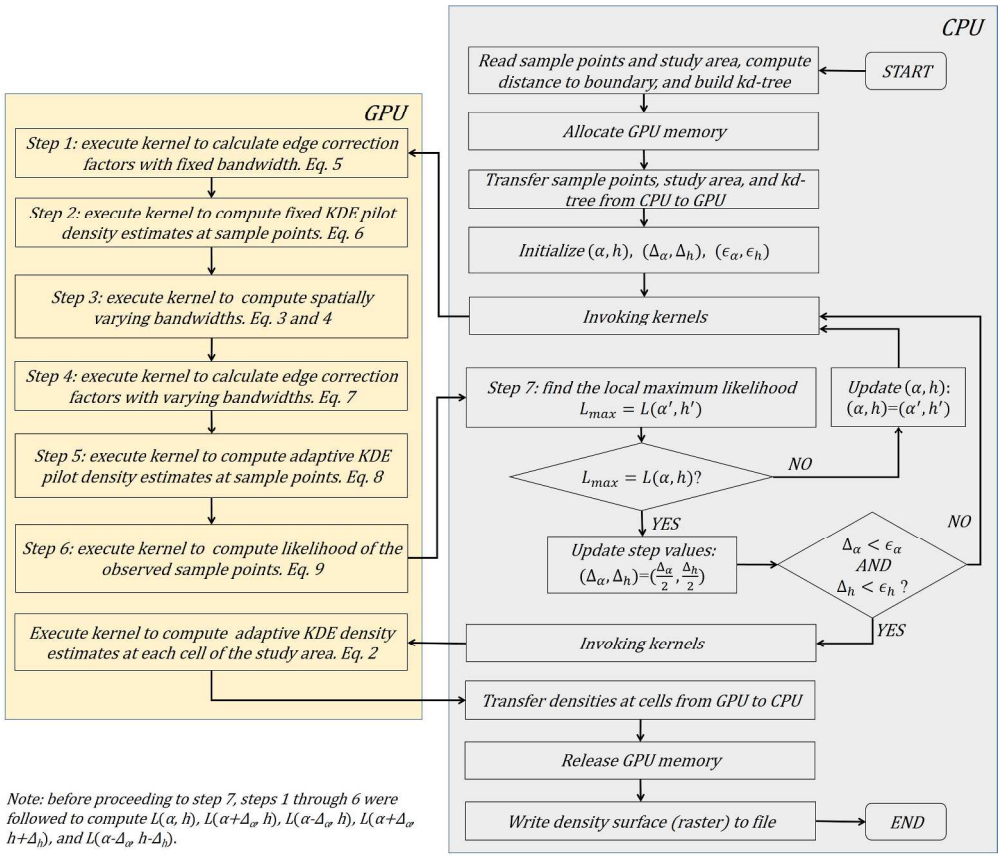

Figure 2. Flow chart of the GPU-enabled Brunsdon's algorithm for adaptive KDE.

242x206mm (300 x 300 DPI)

Algorithm 1: Compute edge correction factors for sample points

```
1  function CalcEdgeCorrectionFactors (dPoints, dHs, dRaster, dEdgefactors)
   Input  : Sample points dPoints
   Input  : Bandwidths for sample points dHs
   Input  : Study area raster dRaster
   Output: Edge correction factors for sample points dEdgefactors
2  tid = (blockIdx.y * gridDim.y + blockIdx.x) * blockDim.x + threadIdx.x;
3  h = dHs[tid];
4  cellSize = dRaster.cellSize;
5  if tid ≥ dPoints.numberOfPoints then
6  |  return;
7  end
8  if dPoints.distances[tid] ≥C*h then
9  |  dEdgefactors[tid] = 1.0;
10 |  return;
11 else
12 |  pX = dPoints.xCoordinates[tid];
13 |  pY = dPoints.yCoordinates[tid];
14 |  rowLower=YCOORD_TO_ROW(pY+C*h);
15 |  rowUpper=YCOORD_TO_ROW(pY-C*h);
16 |  colLower=XCOORD_TO_COL(pX-C*h);
17 |  colUpper=XCOORD_TO_COL(pX+C*h);
18 |  mass=0.0;
19 |  for row in rowLower : rowUpper do
20 |  |  for col in colLower : colUpper do
21 |  |  |  val = dRaster.elements[row][col];
22 |  |  |  if val ≠ dRaster.noDataValue then
23 |  |  |  |  cellX=COL_TO_XCOORD(col);
24 |  |  |  |  cellY=ROW_TO_YCOORD(row);
25 |  |  |  |  d=dDistance(pX,pY,cellX,cellY);
26 |  |  |  |  mass+=dGaussianKernel(h,d)*cellSize*cellSize;
27 |  |  |  end
28 |  |  end
29 |  end
30 |  dEdgefactors[tid] = 1.0/mass;
31 end
```

Figure 3. Pseudo code of the kernel for computing edge correction factors for sample points.

147x166mm (300 x 300 DPI)

Algorithm 2: Compute density estimates at sample points

```
1  function CalcDensityAtPoints (dKdtree, dPoints, dHs, dEdgefactors, dDensity)
   Input  : Kd-tree spatial indexing data structure dKdtree
   Input  : Sample points dPoints
   Input  : Bandwidths for sample points dHs
   Input  : Edge correction factors for sample points dEdgefactors
   Output: Density estimates (initialized with zeros) at sample points dDensity
2  tid = (blockIdx.y * gridDim.y + blockIdx.x) * blockDim.x + threadIdx.x;
3  h = dHs[tid];
4  cellSize = dRaster.cellSize;
5  if tid ≥ dPoints.numberOfPoints then
6  |  return;
7  end
8  pX = dPoints.xCoordinates[tid];
9  pY = dPoints.yCoordinates[tid];
10 h = dHs[tid];
11 e = dEdgefactors[tid];
12 query = Point(pX, pY);
13 range = C * h;
14 dRangeSearch(dKdtree, query, range, retNbrs, retIdxs, retDists);
15 for i in 0 : retNbrs-1 do
16 |  idx = retIdxs[i];
17 |  d = retDists[i];
18 |  g = dGaussianKernel(h, d) * e;
19 |  atomicAdd(dDensity[idx], g);
20 end
21
```

Figure 4. Pseudo code of the kernel for computing inclusive density estimates at sample points

139x125mm (300 x 300 DPI)

Algorithm 3: Compute density surface

```
1  function CalcDensitySurface (dPoints, dHs, dEdgefactors, dRaster)
   Input  : Sample points dPoints
   Input  : Bandwidths for sample points dHs
   Input  : Edge correction factors for sample points dEdgefactors
   Output: Study area raster dRaster
2  tid = (blockIdx.y * gridDim.y + blockIdx.x) * blockDim.x + threadIdx.x;
3  nRows=dRaster.nRows;
4  nCols=dRaster.nCols;
5  if tid ≥ nRows * nCols then
6  |  return;
7  end
8  row=tid/nCols;
9  col=tid-row*nCols;
10 noDataValue=dRaster.noDataValue;
11 if dRaster.elements[row][col]==noDataValue then
12 |  return;
13 end
14 cellX=COL_TO_XCOORD(col);
15 cellY=ROW_TO_YCOORD(row);
16 density=0.0;
17 for i in 0 : dPoints.numberOfPoints-1 do
18 |  pX = dPoints.xCoordinates[i];
19 |  pY = dPoints.yCoordinates[i];
20 |  h = dHs[i];
21 |  e = dEdgefactors[i];
22 |  d = dDistance(pX, pY, cellX, cellY);
23 |  density+ = dGaussianKernel(h, d) * e;
24 end
25 dRaster.elements[row][col] = density;
26
```

Figure 5. Pseudo code of the kernel for computing density estimates over the study area.

139x138mm (300 x 300 DPI)

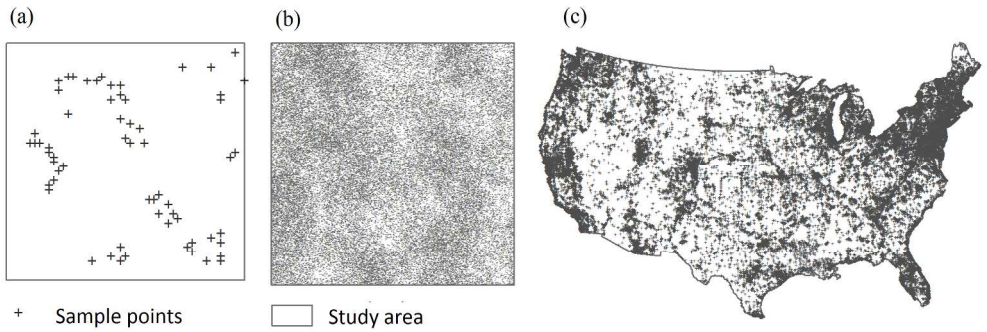

Figure 6. Test data sets used in evaluation experiments. (a) The Redwood data set for testing correctness (n = 62). (b) One of the simulated point pattern (n = 50,000). (c) The eBird checklists data set (n = 78,977).

245x85mm (300 x 300 DPI)

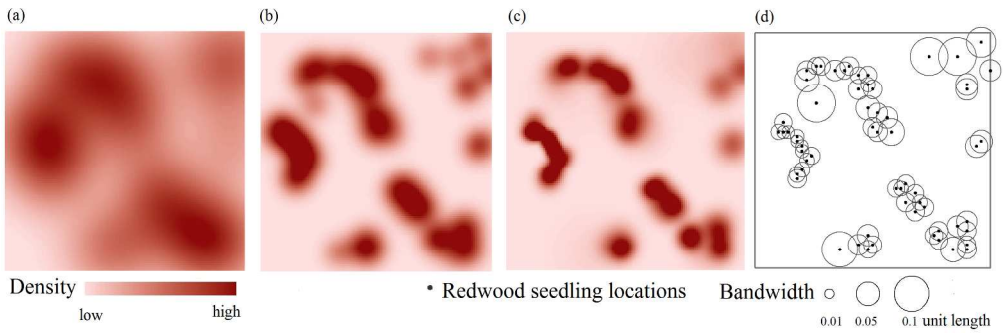

Figure 7. Spatial point pattern analysis on the Redwood data set. Density surface estimated with (a) 'rule-of-thumb' fixed bandwidth (h = 0.123), (b) cross-validated fixed optimal bandwidth (h=0.045), and (c) spatially adaptive optimal bandwidths (h = 0.035, α = 1.47). (d) The adaptive optimal bandwidths used to estimate density surface in (c).

249x81mm (300 x 300 DPI)

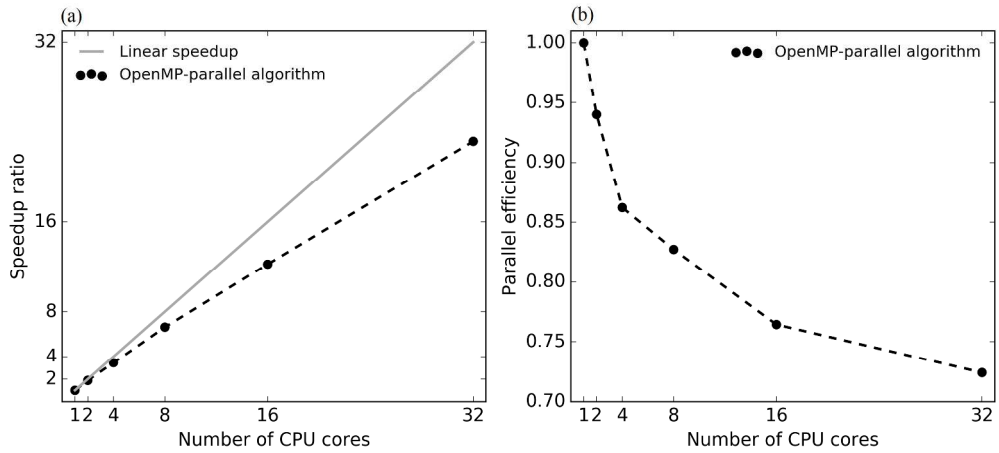

Figure 8. Performance of the OpenMP-parallel algorithm for adaptive KDE on the test data set (n = 50,000, m = 160,000): (a) speed up ratio and (b) parallel efficiency - defined as speedup ratio divided by number of CPU cores.

274x121mm (300 x 300 DPI)

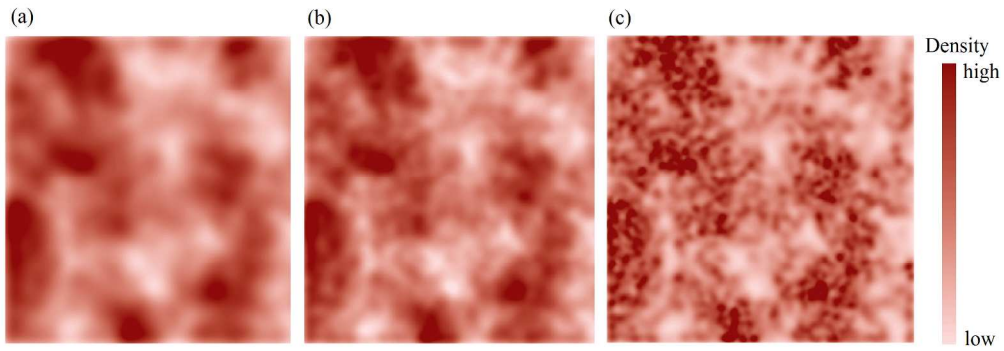

Figure 9. Spatial point pattern analysis on the test data set (n=50,000, m=160,000). Density surface estimated with (a) 'rule-of-thumb' fixed bandwidth (h=0.025), (b) cross-validated fixed optimal bandwidth (h = 0.017), and (c) spatially adaptive optimal bandwidths (h = 0.010, α = 1.088). The density surface estimated using adaptive KDE captured more local density variations.

245x85mm (300 x 300 DPI)

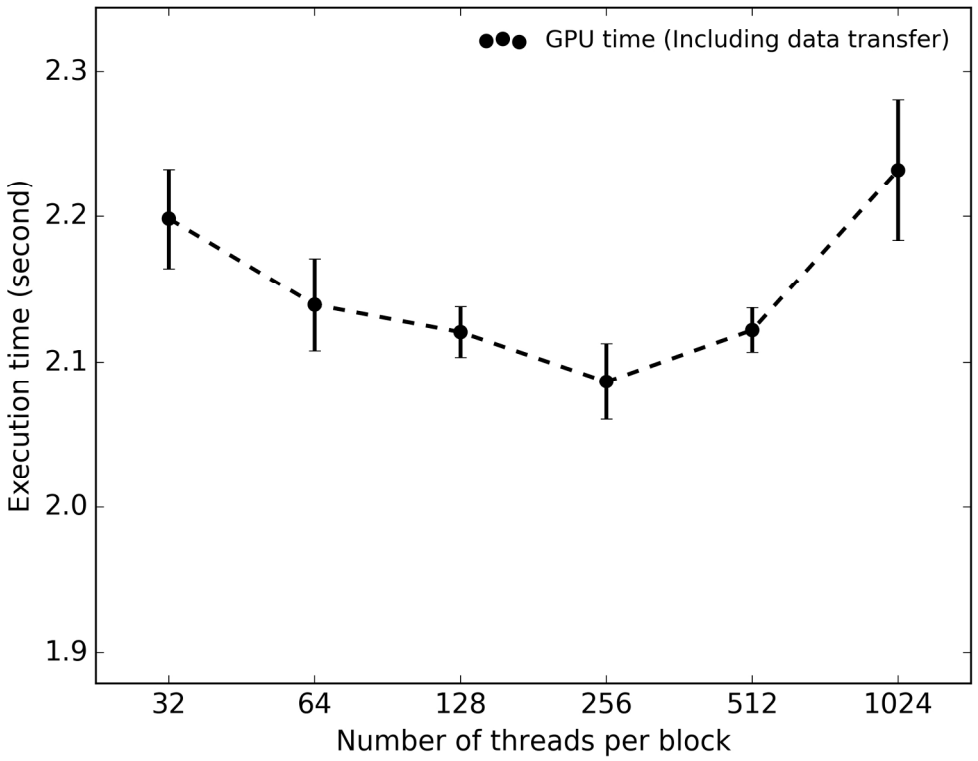

Figure 10. The impact of number of threads per block on performance of the GPU-parallel algorithm for adaptive KDE on the test data set (n = 50,000, m = 160,000) (execution time was averaged over 10 runs; error bars indicate standard deviations).

180x140mm (300 x 300 DPI)

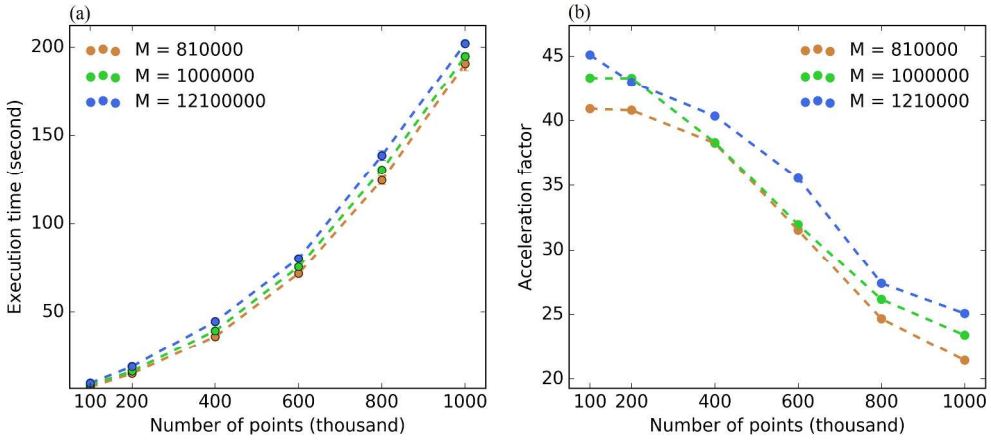

Figure 11. Impact of number of points on the GPU-parallel algorithm for adaptive KDE. (a) Execution time of the GPU-parallel algorithm (averaged over 10 runs; error bars indicate standard deviations). (b) Acceleration factor of the GPU-parallel algorithm over the OpenMP-parallel algorithm. Experiments were run with study areas of various numbers of cells (i.e., 900 × 900 = 810,000, 1000 × 1000 = 1,000,000, and 1,100 × 1,100 = 1,210,000).

279x121mm (300 x 300 DPI)

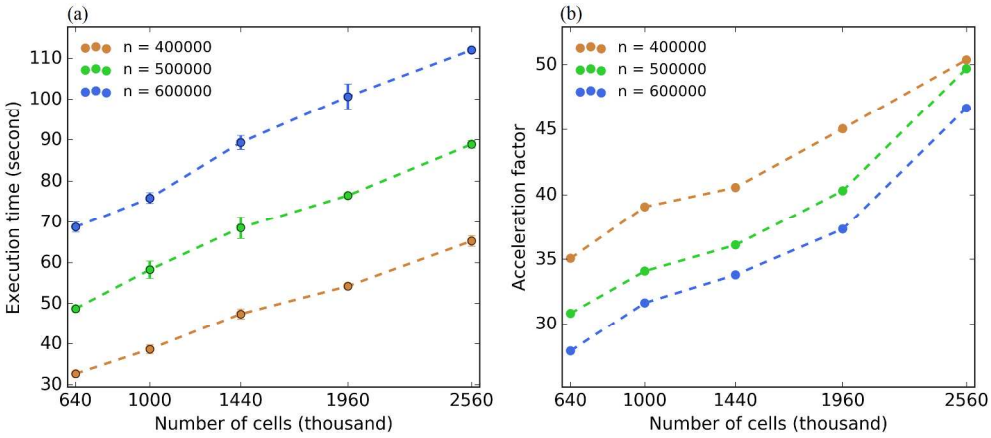

Figure 12. Impact of number of cells on the GPU-parallel algorithm for adaptive KDE. (a) Execution time of the GPU-parallel algorithm (averaged over 10 runs; error bars indicate standard deviations). (b) Acceleration factor of the GPU-parallel algorithm over the OpenMP-parallel algorithm. Experiments were run with various numbers of points (i.e., 400,000, 500,000, and 600,000).

282x120mm (300 x 300 DPI)

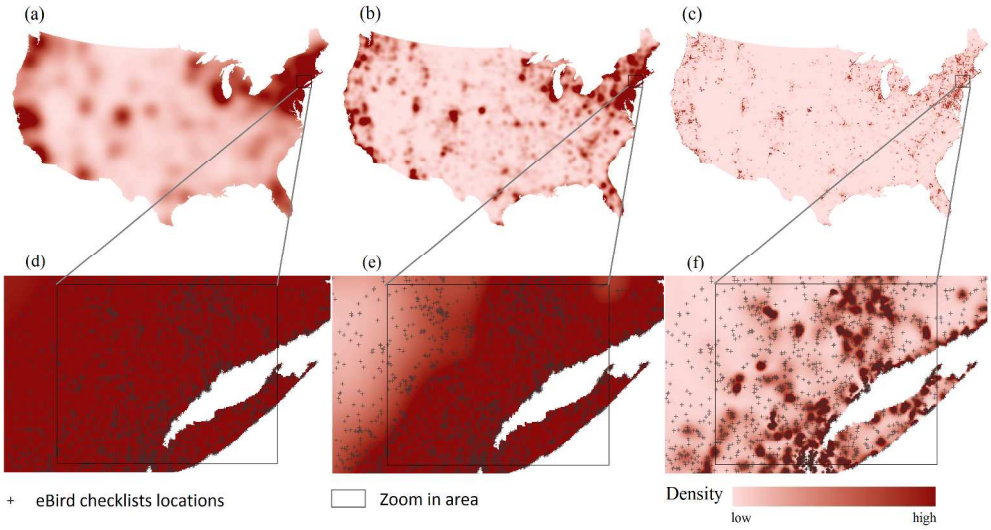

Figure 13. Point pattern analysis on the eBird checklists data set (n = 78,977, m = 12,948,156). Density surface estimated with (a) 'rule-of-thumb' fixed bandwidth (h=84.4 km), (b) cross-validated fixed optimal bandwidth (h = 24.5 km), and (c) spatially adaptive optimal bandwidths (h = 4.2 km, α = 1.21). (d), (e), and (f) are the zoom-in maps of a small area on the east coast of United States.

293x163mm (300 x 300 DPI)

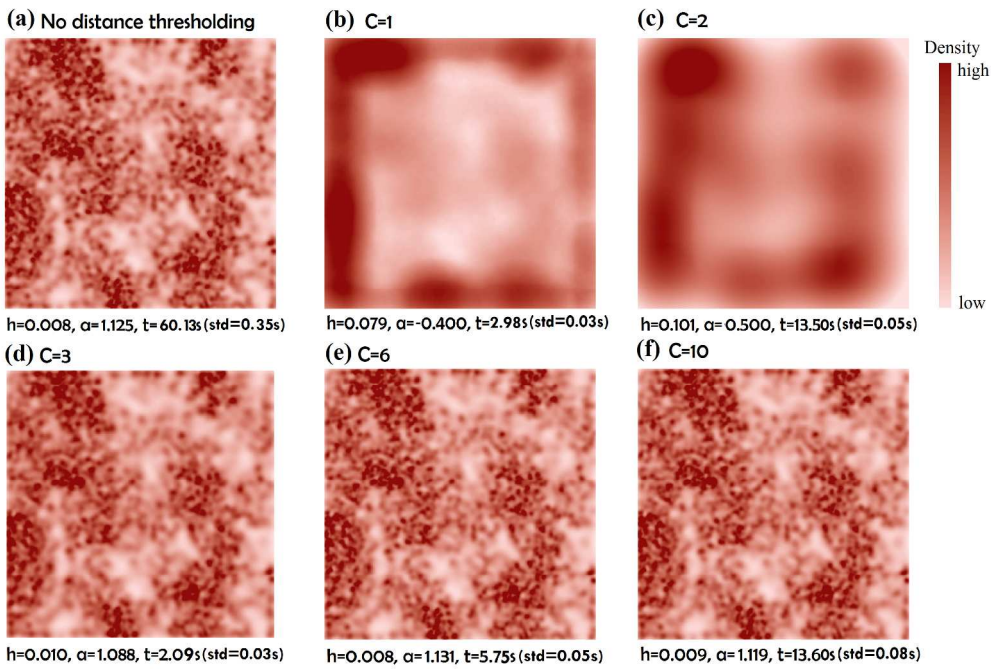

Figure 14. Impact of C value on the estimated density surface and on performance of the algorithm. (a) Density surface estimated without using C×h as a distance threshold to exclude raster cells or sample points in computation. (b) through (f) showed the density surfaces estimated using C=1, C=2, C=3, C=6, and C=10. The estimated parameters (i.e., h and α) and the corresponding execution time (average of 10 runs and the standard deviation) were also shown on each figure.

280x187mm (300 x 300 DPI)

Table 1. Performance of the non-optimized and optimized adaptive KDE algorithms on the test data set (n=50,000, m=160,000) (execution time was averaged over 10 runs, except the sequential non-optimized algorithm averaged over 5 runs; time unit in seconds).

	Algorithm		*No optimization*	*Partial optimization (avoiding re-computing edge correction factors)*	*Full optimization (avoiding re-computing edge correction factors and kd-tree indexing)*
Execution time	Sequential	Mean	109840.26	19348.03	1459.58
		Std	1046.94	439.21	41.87
	OpenMP-parallel	Mean	3882.55	641.97	63.63
		Std	8.97	0.17	1.66
	GPU-parallel	Mean	60.13	9.88	2.13
		Std	0.33	0.06	0.05
Acceleration factor	Sequential		1.0	5.7	75.3
	OpenMP-parallel		1.0	6.0	61.0
	GPU-parallel		1.0	6.1	28.2

Table 2. Performance of the OpenMP-parallel adaptive KDE algorithms on the test data set (n=50,000, m=160,000) using different number of CPU cores (execution time was averaged over 10 runs; parallel efficiency = speedup ratio / number of CPU cores; time unit in seconds).

Number of CPU cores		1 (sequential)	2	4	8	16	32
Execution time	Mean	1459.58	784.20	427.51	222.80	120.56	63.63
	Std	41.87	29.06	10.34	2.70	2.58	1.66
Speedup ratio		1.0	1.9	3.4	6.6	12.1	22.9
Parallel efficiency		1.00	0.93	0.85	0.82	0.76	0.72

Table 3. Performance of the GPU-parallel, OpenMP-parallel, and sequential algorithm to conduct point pattern analysis on the test data set (n = 50,000, m = 160,000) (execution time was averaged over 10 runs; time unit in seconds).

	Algorithm		*Fixed* KDE		*Adaptive* KDE
			Rule-of-thumb	Cross-validation	
Execution time	Sequential	Mean	435.54	967.08	1459.58
		Std	15.16	25.99	41.87
	OpenMP-parallel	Mean	18.50	40.56	63.63
		Std	0.01	0.26	1.66
	GPU-parallel	Mean	0.50	1.05	2.13
		Std	0.02	0.01	0.05
Acceleration factor	Sequential/OpenMP		23.5	23.8	22.9
	Sequential/GPU		871.1	921.0	685.2
	OpenMP/GPU		37.0	38.6	29.9

Table 4. Execution time of the algorithms for conducting point pattern analysis on the eBird checklists data set (n = 78,977, m = 12,948,156) using the *adaptive* KDE approach and the *fixed* KDE approach (execution time was averaged over 10 runs; time unit in seconds).

	Algorithm		*Fixed* KDE		*Adaptive* KDE
			Rule-of-thumb	Cross-validation	
Execution time	OpenMP-parallel	Mean	2223.29	3261.86	5245.69
		Std	107.25	25.08	175.35
	GPU-parallel	Mean	30.27	122.73	346.45
		Std	0.27	0.46	16.49
Acceleration	OpenMP/GPU		73.5	26.6	15.1